Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 105
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 105
Espoo 2006 HUT-TCS-A105

TECHNIQUES FOR SOLVING BOOLEAN EQUATION SYSTEMS

Misa Keinänen

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D’HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 105
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 105
Espoo 2006 HUT-TCS-A105

TECHNIQUES FOR SOLVING BOOLEAN EQUATION SYSTEMS

Misa Keinänen

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering, for public examination and debate in Auditorium T3 at Helsinki University of Technology (Espoo, Finland) on the 15th of December, 2006, at 12 o'clock noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FI-02015 TKK, FINLAND
Tel. +358 9 451 1
Fax. +358 9 451 3369
E-mail: lab@tcs.tkk.fi
URL: http://www.tcs.tkk.fi/

ISBN–13 978-951-22-8546-4
ISBN–10 951-22-8546-0
ISSN 1457-7615

Electronic Edition
2006

Abstract: Boolean equation systems are ordered sequences of Boolean equations decorated with least and greatest fixpoint operators. Boolean equation systems provide a useful framework for formal verification because various specification and verification problems, for instance, μ-calculus model checking can be represented as the problem of solving Boolean equation systems. The general problem of solving a Boolean equation system is a computationally hard task, and no polynomial time solution technique for the problem has been discovered so far. In this thesis, techniques for finding solutions to Boolean equation systems are studied and new methods for solving such systems are devised.

The thesis presents a general framework that allows for dividing Boolean equation systems into individual blocks and solving these blocks in isolation with special techniques. Three special techniques are presented, namely: (i) new specialized algorithms for disjunctive and conjunctive form Boolean equation systems, (ii) a new encoding of a general form Boolean equation system into answer set programming, and (iii) new encodings of a general form Boolean equation systems into satisfiability problems. The approaches (ii) and (iii) are motivated by the recent success of answer set programming solvers and satisfiability solvers in formal verification.

First, the thesis presents especially fast solution algorithms for disjunctive and conjunctive classes of Boolean equation systems. These special algorithms are useful because many practically relevant model checking problems can be represented as Boolean equation systems that are disjunctive or conjunctive. The new algorithms have been implemented and the performance of the algorithms has been compared experimentally on communication protocol verification examples.

Second, the thesis gives a translation of the problem of solving a general form Boolean equation system into the problem of finding a stable model of a logic program. The translation allows to use implementations of answer set programming solvers to solve Boolean equation systems. Experimental tests have been performed using the presented approach and these experiments indicate the usefulness of answer set programming in this problem domain.

Third, the thesis presents reductions from the problem of solving general form Boolean equation systems to the satisfiability problems of difference logic and propositional logic. The reductions allow to use implementations of satisfiability solvers to solve Boolean equation systems. The presented reductions have been implemented and it is shown via experiments that the new approach leads to practically efficient methods to solve general Boolean equation systems.

Keywords: answer set programming, Boolean equation systems, computer-aided verification, satisfiability problems

TIIVISTELMÄ: Boolen yhtälöryhmät ovat kiintopisteoperaattoreilla varustettuja Boolen yhtälöitä. Boolen yhtälöryhmät luovat hyödyllisen viitekehyksen tietokoneavusteiselle verifioinnille, sillä monet määrittely- ja verifiointiongelmat voidaan kuvata tällaisten kiintopisteyhtälöiden avulla. Työssä kehitetään uusia menetelmiä Boolen yhtälöryhmien ratkaisemiseen.

Työssä esitetään yleinen viitekehys Boolen yhtälöryhmien ratkaisemiseen, joka yksinkertaistaa ratkaisun laskemista jakamalla yhtälöryhmät yksinkertaisempiin aliongelmiin. Työssä esitetään kolme uutta mentelmää Boolen yhtälöryhmien ratkaisemiseen.

Konjunktiivisten ja disjunktiivisten Boolen yhtälöryhmien ratkaisemiseen kehitetään uusia algoritmeja, sekä esitetään näiden toteutukset ja suorituskykyjä koskevia koetuloksia.

Työssä kehitetään käännös Boolen yhtälöryhmän ratkaisemisesta logiikkaohjelman stabiilin mallin löytämiseen sekä menetelmän toimivuutta koskevia koetuloksia. Käännös mahdollistaa logiikkaohjelmointiympäristöjen toteutusten käytön Boolen yhtälöryhmien ratkaisemiseen. Koetulokset osoittavat rajoitepohjaisen logiikkaohjelmointiympäristön tehokkuuden Boolen yhtälöryhmien ratkaisemisessa.

Työssä kehitetään myös käännökset Boolen yhtälöryhmän ratkaisemisesta differenssilogiikan sekä lauselogiikan toteutuvuusongelmiin. Käännökset mahdollistavat toteutuvuustarkastimien käytön Boolen yhtälöryhmien ratkaisemiseen. Koetulokset osoittavat esitettyjen menetelmien tehokkuuden Boolen yhtälöryhmien ratkaisemisessa.

AVAINSANAT: Logiikkaohjelmointi, Boolen yhtälöryhmät, tietokoneavusteinen verifiointi, logiikan toteutuvuusongelmat

CONTENTS

List of Figures

List of Tables

PREFACE

This dissertation is the outcome of years of studies and research carried out at the Laboratory for Theoretical Computer Science of the Helsinki University of Technology (TKK) between 2001 and 2006. I am grateful to a number of people for contributing, in a way or another, to this work.

First of all, I would like to thank Professor Ilkka Niemelä, the supervising professor, for providing the possibility to join his research group at TKK. The encouraging research environment at the Laboratory for Theoretical Computer Science of TKK is much due to Niemelä. Also, I thank Dr. Keijo Heljanko, the thesis instructor, for substantial advice. Both Niemelä and Heljanko have been ideal instructors, never short of new ideas, always willing to discuss and comment my work.

I thank Professor Jan Friso Groote for various constructive ideas and successful collaboration. I am greatly indebted to Professor Wan Fokkink for the possibility to visit his research group at the national research institute for mathematics and computer science (CWI) in the Netherlands in 2003. Thanks to Prof. Fokkink, Dr. Jaco van de Pol and others for the discussions and the friendly research climate at the CWI. Dr. van de Pol made useful remarks on the draft of this dissertation.

I am grateful to Dr. Martin Lange for contributions to this work through collaboration, and for counselling the Encoding star ship on the BMC mission deep down in the omega-quadrant while encountering the ν. Equally well, I want to thank among others Dr. Tommi Junttila, Dr. Petteri Kaski, Dr. Orna Kupferman, Dr. Timo Latvala and Professor Angelika Mader for inspiring discussions along the years.

Both pre-examiners of the dissertation, Prof. H.R. Andersen and Dr. M. Jurdziński, are thanked for making useful comments and suggestions for improvements.

The financial supports of Helsinki Graduate School in Computer Science and Engineering, the Academy of Finland and the Emil Aaltonen Foundation are gratefully acknowledged.

Especially, I want to express my gratitude to my wife, family and friends for their invaluable, constant support.

1 INTRODUCTION

In this thesis, we study Boolean Equation Systems (BESs) [1, 70, 75, 101] These are ordered sequences of Boolean equations decorated with fixpoint signs. More precisely, a Boolean equation system consists of equations with Boolean variables as left-hand sides and positive propositional formulas as right-hand sides. In particular, we restrict the attention to solution techniques for Boolean equation systems. The research topic belongs to the area of formal verification but more specifically it addresses effective ways of solving systems of fixpoint equations. Our work is mainly inspired by the usefulness of Boolean equation systems in formal verification of computerized systems.

Computerized systems are present almost everywhere in a modern society. During a lifetime an average person migh use thousands of computerized devices. Typical examples of these devices include consumer electronics such as digital cameras, mobile phones, televisions, stereo equipment, and microwave ovens. In addition, hardware and software systems control airplanes, cars, elevators, medical devices, missiles, ships, spacecrafts, trains, chemical and nuclear power plants, and so on.

Computerized systems are built by human beings and, unfortunately, humans make mistakes. Both design errors and faulty implementation may lead hardware and software systems to behave in unexpected ways which in turn may lead to financial losses and even hazardous situations.

The main aim of *formal verification* is to verify that a computerized system satisfies a formal specification which describes the correct behaviour of the system. Motivated mainly by a need to ensure the correctness of hardware and software systems, an important discipline of formal verification emerged in the early 1980's, called model checking. Over the past decades, model checking has become a very active research area as well as a widely used verification technique in computer hardware and software industry.

In brief, *model checking* [17] is an automated method to check that a requirement holds for a model of a hardware or software system. A hardware or software system is typically modelled in a particular specification language. The correctness requirements are then specified as formulas in some temporal logic. As an intermediate step, a state space may be generated from the system specification. Essentially, the state space is a model which is simply a graph representing all the possible behaviours of the system under consideration. Then, a model checking algorithm decides whether or not the temporal logic formula holds for the model, i.e. the model checking algorithm either verifies or refutes the correcness property. In addition, witnesses or counter-examples can be provided, too.

Model checking has been applied successfully in different phases of the development of various systems. In practice, model checking has proved to be a particularly effective method to detect errors in early design phases of finite-state, concurrent systems such as microprocessors and many communication protocols. For instance, model checking has become an essential part of the system development cycle in the design of VLSI circuits (see, e.g., [14]). Model checking has also been applied, e.g., to assist the design and implementation of telecommunication systems software (see [50] for a survey).

One of the main obstacles to using model checking is its complexity. The complexity of model checking arises mainly from two sources:

I State space explosion phenomenon: the state space generated from the system specification is usually much (exponentially) larger than the system specification.

II Expressive logics have computationally complex model checking algorithms.

Therefore, much of the model checking research is centered around techniques which try to alleviate both of these shortcomings. In brief, ways to alleviate the state space explosion problem (I) include:

1. *On-the-fly model checking* [25] integrates the state space generation and model checking phases in order to detect counter-examples early.

2. *Symbolic model checking* [14, 9] represents the state space compactly by clever data structures.

3. *Partial order reduction* [40, 99] techniques ignore certain executions of the system because they are covered by other behaviours.

4. *Symmetry reductions* [18, 32] try to avoid building the entire state spaces based on the fact that many systems are highly symmetric.

5. *Abstraction* [19] methods remove details of the system behaviours and work with approximations of the state space.

In this thesis, rather than trying to alleviate the state space explosion problem we concentrate on (II). Consequently, we aim to devise techniques to better cope with expressive logics which have computationally complex model checking algorithms.

The μ-calculus [60] is an expressive logic for system verification, and many important features of system models can be expressed with the μ-calculus. In fact, most logics used in model checking can be encoded into the μ-calculus. For these reasons, the μ-calculus is a logic widely studied in the recent systems verification literature. It is well-known that the μ-calculus model checking problem is in the complexity class NP $\cap$ co-NP [34, 35] (and even in UP $\cap$ co-UP [53]). Yet, the computational complexity of the μ-calculus model checking problem is unresolved, and no polynomial time algorithm has been discovered so far.

Our goal in this thesis is to devise practically effective methods for μ-calculus model checking. Boolean equation systems provide a suitable framework for our task because μ-calculus model checking problem can be easily translated to Boolean equation systems (see, e.g., [4, 70, 75] or Section 2.4 for such translations). Representing μ-calculus formulas and system models as Boolean equation systems has proven to be a useful approach for implementing model checking and for obtaining improvements to this verification method.

In the following subsections, we will briefly discuss related work, state the main contributions of this thesis, and, finally, we will outline the general organization of the thesis.

1.1 Related Work

There is a large existing body of knowledge on the μ-calculus model checking problem albeit the computational complexity of the problem is yet unresolved. In particular, various effective model checking algorithms exist for syntactic fragments of the μ-calculus. To mention a few of them, Arnold and Crubille [3] present an algorithm for checking alternation depth 1 formulas of μ-calculus which is linear in the size of the model and quadratic in the size of the formula. Cleaveland and Steffen [23] improve this result by making the algorithm linear also in the size of the formula. Andersen [1], and similarly Vergauwen and Lewi [101], show how model checking alternation depth 1 formulas amounts to the evaluation of *Boolean graphs*, resulting also in linear time techniques for model checking alternation depth 1 formulas. Even more expressive subsets of the μ-calculus have been investigated by Bhat and Cleaveland [8] as well as Emerson et al. [34, 35]. They present polynomial time model checking algorithms for fragments L1 and L2 which may contain alternating fixpoint formulas. Notable algorithms for solving the general μ-calculus model checking problem include, for instance, [31, 69, 88, 67].

The notion of a *Boolean equation system* goes back at least to the work of Larsen [64], where he presents an early form of a Boolean equation system. Larsen gives a sound and complete proof system for Boolean equation systems consisting of minimal fixpoint equations. Larsen shows also how correctness questions of finite-state parallel systems can be solved in this framework. In the same way, Boolean equation systems are studied in detail, for example, by Vergauven and Lewi [101], and by Andersen and Vergauven [2].

In [70], Mader provides an extensive study of the properties of Boolean equation systems. She shows how the model checking problem of μ-calculus can be solved in terms of Boolean equation systems. In addition, Mader provides a proof system for solving general Boolean equation systems by means of algebraic manipulations. This leads to an iterative algorithm to solve general form Boolean equation systems called Gauß elimination. Mader shows that the algebraic approach is also applicable to solving infinite systems of equations, an extension of Boolean equation systems to infinite sequences of Boolean equations possibly involving infinite Boolean formulas. This leads to a model checking technique for infinite state spaces.

In [75], Mateescu describes solution algorithms for alternation-free Boolean equation system. The approach from [75] can be used for both bisimulation checking and for model checking of alternation-free μ-calculus on finite-state systems. Furthermore, in [74], Mateescu provides algorithms that can be used to compute counterexamples as well as diagnostic information explaining the solution computed to a given variable of a Boolean equation system.

In [61], Kumar and others apply answer set programming to solve Boolean equation systems. They propose to solve general form Boolean equation systems by translating them to propositional normal logic programs, and computing stable models which satisfy certain criteria of preference.

There is also a recent direction of research centered around an extension of Boolean equation systems with data. Such systems are often called parameterized Boolean equation systems and they are also known as first-order Boolean equation systems. In [43], Groote and Willemse show how a μ-calculus formula and a process algebraic specification, both involving data parameters, can be transformed into a parameterized Boolean equation system. In [45, 44], various solution methods for parameterized Boolean equation systems are studied. An advantage of this kind of approach is that it allows for dealing with the verification of infinite state systems.

Rather than providing a comprehensive list of work in the field with a special reference to Boolean equation systems, the above list of results shows that Boolean equation systems have been studied in some depth in the recent systems verification literature. Yet, the computational complexity of the problem of solving a general Boolean equation system is still unresolved, and there is a need to develop further solution methods that are efficient in practice.

It is worthwhile to mention that the μ-calculus model checking has been studied in other frameworks than Boolean equation systems as well. For instance, it is well-known that the model checking problem for the μ-calculus is equivalent to the non-emptiness problem of parity tree automata [34, 35]. In [6], another automata-theoretic approach to μ-calculus model checking is presented in terms of alternating parity automata. Also, the μ-calculus model checking problem has been treated in a game-theoretical setting. For instance, [92] presents a technique to solve the problem in terms of so-called model checking games. In particular, the problems of μ-calculus model checking and determining a winner of a parity game are equivalent [33]. Many authors have thus developed μ-calculus model checking techniques through parity games, see e.g. [54, 7, 81]. Finally, [21] and [93] present tableau-based model checking procedures for μ-calculus. Most of these alternative frameworks fall out of the scope of this thesis but it is important to keep in mind that one could usually switch to other equivalent formalisms as well.

1.2 Contribution of the Thesis

This thesis concentrates on the following topics. The work presents a general framework that allows for dividing Boolean equation systems into individual blocks and solving the blocks in isolation with special methods. The framework is based on two fundamental design decisions. The framework uses graph-theoretic techniques to efficiently build a block partitioning of a Boolean equation system. Then, the framework solves the resulting blocks using a customized solution method for each partition of the underlying Boolean equation system. This approach enables considerable optimization of the solution methods. Previously, a quite similar approach for equational systems has been proposed already in [22].

The thesis presents new solution methods for important subclasses of Boolean equation systems. In particular, we study solution methods for Boolean equation systems which are either in conjunctive or disjunctive form. This is motivated by the fact that many practically relevant μ-calculus formulas can be encoded as Boolean equation systems that consist of conjunctive and disjunctive blocks. For instance, the model checking problems for Hennessy

Milner Logic (HML) [49], Computation Tree Logic (CTL) [17], and many equivalence/preorder problems result in alternation-free Boolean equation systems in conjunctive/disjunctive forms. Moreover, encoding the L1 and L2 fragments [8, 34, 35] of μ-calculus (and similar subsets) or many fairness constraints as Boolean equation systems result in alternating Boolean equation systems with only disjunctive and conjunctive form blocks. For instance, Emerson et.al [34, 35] show that the fragment L2 is as expressive as the logic ECTL* given in [100], where ECTL* is the extended version of CTL* logic in which ω-regular experssions are used as path formulas. Hence, the problem of solving the conjunctive and disjunctive subclasses of Boolean equation systems is so important that developing special purpose solution techniques for these classes is worthwhile.

Previously, Mateescu [75] presented a solution algorithm for conjunctive and disjunctive Boolean equation systems. However, Mateescu's approach is restricted to alternation-free Boolean equation systems only. We are only aware of one sketch of an algorithm that is directed to alternating conjunctive and disjunctive form Boolean equation systems, namely Proposition 6.5 and 6.6 of [70]. In [70], $O(n^2)$ time and $O(n^2)$ space algorithms are provided where n is the size of the Boolean equation system.

We give two alternative algorithms for solving conjunctive and disjunctive Boolean equation systems. The first of these algorithms presented in this thesis takes time $O(n^2)$ where n is the size of the equation system. For all alternation depths $d > 1$ ($d \leq n$ always holds on Boolean equation systems), the second algorithm finds the solution using time $O(n \log d)$ in the worst case (for $d = 1$ the second algorithm takes time $O(n)$ in the worst case). Both algorithms have the space complexity $O(n)$. Thus, the new algorithms are theoretical improvements over the previous works in the setting of Boolean equation systems. Our first algorithm is based on Tarjan's depth-first search [95]. The other is essentially a variant of Tarjan's hierarchical clustering algorithm [96], and it is also a variant of a closely related algorithm by King, Kupferman and Vardi [59] in the realm of parity word automata. We have implemented the new algorithms and have done various computational experiments to show that the theoretical improvement also leads to practically efficient solution techniques. The algorithms are compared by experiments on communication protocol verification examples.

In addition, the thesis presents new solution techniques for solving general form Boolean equation systems for which no polynomial time solution algorithms are known to date.

Since the problem of solving a general Boolean equation system is in the complexity class NP $\cap$ co-NP [70], it should be possible to employ answer set programming solvers as effective proof engines to solve general Boolean equation systems. Initially, this kind of approach has been suggested in [61], and it is largely motivated by the success of answer set programming systems in solving various computationally hard problems. Inspired by the initial idea proposed in [61] we introduce a new mapping from Boolean equation systems to normal logic programs. Namely, we reduce the problem of solving general Boolean equation systems to computing stable models of normal logic programs. Our translation is such that it ensures the polynomial time complexity of solving both conjunctive and disjunctive alternating Boolean equation systems. A drawback of the approach from [61] is that typical answer set programming systems do not support the computation of answer sets satisfying the kind of preference criteria defined in [61]. In contrast, our translation only uses the kinds of rules that are already implemented in many answer set programming systems, thus allowing us to solve general form Boolean equation systems by answer set programming systems.

In order to obtain practically efficient solution methods for general Boolean equation systems, the thesis presents new reductions from the problem of solving general form Boolean equation systems to the satisfiability problems of difference logic and propositional logic (SAT). This part of the thesis is essentially based on a submitted manuscript [47] which extends the results in [63]. The reduction is first given into difference logic, i.e. SAT combined with the theory of integer differences, an instance of the SAT modulo theories (SMT) framework. In the second stage the integer variables and constraints of the difference logic encoding are replaced with a set of Boolean variables and constraints on them, giving rise to a pure SAT encoding of the problem. These kinds of reductions are motivated by the recent success of satisfiability solvers in formal verification, symbolic model checking in particular. The research hyphothesis is that we can employ the recent results of satisfiability solving techniques to significantly improve the methods to solve general form Boolean equation systems, thus leading to practically efficient model checking techniques for full μ-calculus. We are not aware of any previous attempts to do μ-calculus model checking through a reduction to difference logic satisfiability.

Finally, we have implemented the presented techniques and tested their performance on benchmark problems. Consequently, in this thesis we investigate experimentally the performance of the new techniques. In the experimental part of the work, we report the results and analysis of the extensive experiments that we have performed in order to evaluate and compare the techniques.

Parts of the contributions in the thesis have been published in the following publications:

[41] J.F. Groote and M. Keinänen. Solving Disjunctive/Conjunctive Boolean Equation Systems with Alternating Fixed Points. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 2988, pages 436 – 450, Springer Verlag, 2004.

[56] M. Keinänen. Obtaining Memory Efficient Solutions to Boolean Equation Systems. *Electronic Notes in Theoretical Computer Science*, 133: 175–191. Elsevier, 2005.

[57] M. Keinänen and I. Niemelä. Solving Alternating Boolean Equation Systems in Answer Set Programming. *Applications of Declarative Programming and Knowledge Management*, Lecture Notes in Artificial Intelligence 3392, pages 134–148, Springer Verlag, 2005.

[42] J.F. Groote and M. Keinänen. A Sub-quadratic Algorithm for Conjunctive and Disjunctive Boolean Equation Systems. In *Proceedings of 2nd International Colloquium on Theoretical Aspects of Computing,*

Lecture Notes in Computer Science 3722, pages 545–558, Springer Verlag, 2005.

[47] K. Heljanko, M. Keinänen, M. Lange and I. Niemelä. Solving Parity Games by a Reduction to SAT. Submitted manuscript.

1.3 Organization of the Thesis

The organization of this thesis is as follows.

Section 2 provides the needed background to read the thesis. Section 3 presents an overview of our general framework to solve a Boolean equation system. Section 4 discusses solution methods for alternation-free parts of Boolean equation systems. Section 5 describes algorithms for conjunctive and disjunctive cases of Boolean equation systems. Section 6 details a method to solve general form, alternating parts of Boolean equation systems with the answer set programming approach. Section 7 presents a technique to solve general form, alternating parts of Boolean equation systems by reductions to satisfiability problems. Section 8 describes and discusses experimental research results. Finally, section 9 presents conclusions, and suggests directions for future work.

2 BACKGROUND

This section presents some basic concepts that will be required in the following sections. The current section presents essentially an introduction to μ-calculus model checking with Boolean equation systems and reviews the answer set programming framework and defines the satisfiability problems.

2.1 Labelled Transition Systems as Formal Models

In the introduction, we outlined that in order to analyze the behaviour of a hardware or software system with formal methods we first need to construct a model of the system that is to be verified. As mentioned before, systems are typically modeled in some suitable modeling formalism, and a state space of the system may be obtained from the system specification. In this thesis, we do not consider in detail how the state spaces will actually be obtained from the system descriptions but instead our starting point is a simple model that captures the computations of systems, namely labelled transition systems (LTSs).

The formal models of systems that we will consider in this thesis are the following.

Definition 1 (Labelled transition systems) *Let $\mathcal{L}$ be a set of action labels. A labelled transition system $\mathcal{T}$ is a tuple $(S, \{\xrightarrow{a} \mid a \in \mathcal{L}\})$ where*

- *S is a (finite) set of states, and*
- *for every $a \in \mathcal{L}$, the relation $\xrightarrow{a} \subseteq S \times S$ is a transition relation.*

We will write $(s, t) \in \xrightarrow{a}$ as $s \xrightarrow{a} t$.

The intuitive idea here is that the behaviour of a system is modeled by a state transition graph consisting of nodes and labelled edges. The nodes represent possible states of the system. The edges, in turn, represent the state transitions of the system (for example events, input/output actions, internal steps, variable assignments etc.). Usually, model checking techniques assume this kind of a model of the system, and it can typically be obtained as an end-product of state space generation or compilation.

The formal model built from the system describes all possible behaviors and execution sequences of the system to be verified. Consequently, for each execution sequence of the system there is a corresponding path of a model which is simply a connected sequence of consecutive transitions of the labelled transition system $\mathcal{T}$. Given a formal model of the system to be verified, we wish to express various properties of system models in order to be able to reason about the correctness. Let us now define a specification language which allows us to express formally the properties of systems.

2.2 The μ-Calculus as a Specification Language

In this subsection, we will briefly give the basic definitions concerning the μ-calculus [60] and demonstrate how to use it as a language for expressing system properties. The μ-calculus is based on fixpoint computations [97], and a more detailed survey on this logic can be found in [13].

Syntax of the μ-Calculus

The μ-calculus [60] is an expressive logic for system verification, and most model checking logics can be encoded in the μ-calculus. Many important features of system models, such as liveness and fairness properties, can also be expressed with the logic. For these reasons, μ-calculus is a logic widely studied in the recent systems verification literature(see, e.g., [13] for a comprehensive survey of the μ-calculus).

We define the syntax of μ-calculus in positive normal form. Let $\mathcal{Z}$ be a set of recursion variables (indicated by $X, Y, Z \ldots$). Let $\mathcal{L}$ be a set of action labels (indicated by $a, b, c, \ldots$). Then, the set of μ-calculus formulas with respect to $\mathcal{Z}$ and $\mathcal{L}$ is defined by the grammar

$$\Phi ::= \ \bot \mid \top \mid X \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [a]\Phi \mid \langle a \rangle \Phi \mid \mu X.\Phi \mid \nu X.\Phi$$

where X and a range over $\mathcal{Z}$ and $\mathcal{L}$, respectively. As usual, for the above syntax we assume that the modal operators ($[a]$ and $\langle a \rangle$) have higher precedence than the Boolean connectives ($\wedge$ and $\vee$). Moreover, we assume that the least fixpoint operator μ and the greatest fixpoint operator ν have the lowest precedence.

In addition, we will make use of some extensions to the above syntax which are very standard in the literature. For instance, the notation $[-]\Phi$ is a shorthand for $\bigwedge_{a \in \mathcal{L}}[a]\Phi$, and the notation $[-b]\Phi$ is a shorthand for $\bigwedge_{a \in \mathcal{L}\setminus\{b\}}[a]\Phi$.

An important syntactic notion of the μ-calculus formulas is the *alternation depth*. The alternation depth of a formula can be defined as the number of alternations between least and greatest fixpoint operators occurring in the formula. For example, [80, 31] give definitions of the alternation depth.

Semantics of the μ-Calculus

Given a set $\mathcal{L}$ of action labels, formulas of the μ-calculus are interpreted relative to a labelled transition system $\mathcal{T} = (S, \{\xrightarrow{a} \mid a \in \mathcal{L}\})$.

A valuation function $\mathcal{V}$ assigns to every variable $X \in \mathcal{Z}$ a set of states $\mathcal{V}(X) \subseteq S$ meaning that variable X holds for all states in $\mathcal{V}(X)$. Let $\mathcal{V}[X/S']$ be the valuation which maps X to S' and otherwise agrees with valuation $\mathcal{V}$.

Then, the semantics of a μ-calculus formula Φ, relative to a transition system $\mathcal{T}$ and a valuation $\mathcal{V}$, is a set of states $||\Phi||^{\mathcal{T}}_{\mathcal{V}}$ which is defined inductively as follows:

$$\begin{aligned}
||\bot||^{\mathcal{T}}_{\mathcal{V}} &= \emptyset \\
||\top||^{\mathcal{T}}_{\mathcal{V}} &= S \\
||X||^{\mathcal{T}}_{\mathcal{V}} &= \mathcal{V}(X) \\
||\Phi_1 \wedge \Phi_2||^{\mathcal{T}}_{\mathcal{V}} &= ||\Phi_1||^{\mathcal{T}}_{\mathcal{V}} \cap ||\Phi_2||^{\mathcal{T}}_{\mathcal{V}} \\
||\Phi_1 \vee \Phi_2||^{\mathcal{T}}_{\mathcal{V}} &= ||\Phi_1||^{\mathcal{T}}_{\mathcal{V}} \cup ||\Phi_2||^{\mathcal{T}}_{\mathcal{V}} \\
||[a]\Phi||^{\mathcal{T}}_{\mathcal{V}} &= \{s \mid \forall t.s \xrightarrow{a} t \Rightarrow t \in ||\Phi||^{\mathcal{T}}_{\mathcal{V}}\} \\
||\langle a \rangle \Phi||^{\mathcal{T}}_{\mathcal{V}} &= \{s \mid \exists t.s \xrightarrow{a} t \wedge t \in ||\Phi||^{\mathcal{T}}_{\mathcal{V}}\} \\
||\mu X.\Phi||^{\mathcal{T}}_{\mathcal{V}} &= \bigcap\{S' \subseteq S \mid ||\Phi||^{\mathcal{T}}_{\mathcal{V}[X/S']} \subseteq S'\}
\end{aligned}$$

$$\|\nu X.\Phi\|^{\mathcal{T}}_{\mathcal{V}} = \bigcup\{S' \subseteq S \mid S' \subseteq \|\Phi\|^{\mathcal{T}}_{\mathcal{V}[X/S']}\}$$

Notice that the semantics of the least and the greatest fixpoint operators is a special case of Knaster–Tarski characterization of the least and the greatest fixpoints of a monotone operator in a complete lattice.

Given a μ-calculus formula Φ and a state s of a labelled transition system $\mathcal{T}$, state s satisfies Φ iff $s \in \|\Phi\|^{\mathcal{T}}_{\mathcal{V}}$; as usual, this is written as $\mathcal{T}, s \models \Phi$.

The *model checking problem* for μ-calculus can be stated as follows.

Definition 2 (μ-calculus model checking) *Given a μ-calculus formula Φ, and a state s of a labelled transition system $\mathcal{T}$, the μ-calculus model checking problem is to determine whether $\mathcal{T}, s \models \Phi$ holds.*

It is well-known that the μ-calculus model checking problem is in the complexity class NP $\cap$ co-NP. Emerson, Jutla, and Sistla [34, 35] show that the μ-calculus model cheking problem is in NP (and also by symmetry in co-NP). Jurdziński [53] shows that the problem of deciding a winner in parity games is in the complexity class UP$\cap$co-UP, and as a consequence of this fact it can be shown that the μ-calculus model checking problem is in UP$\cap$co-UP too (there is a polynomial time reduction from μ-calculus model checking to parity games, and vice versa).

The seminal result of Emerson and Lei [31] states that there is a model checking algorithm which checks a μ-calculus formula of size n and alternation depth d (here $d \leq n$) on a labelled transition system of size m in time $O(m \cdot n^d)$. This result has been improved to $O(m \cdot n^{\lceil d/2 \rceil + 1})$ by Long et.al [69]. Yet, the complexity of the μ-calculus model checking problem for the unrestricted logic is an open problem; no polynomial time algorithm has been discovered so far.

As mentioned before, various model checking algorithms exist for expressive subsets of the μ-calculus. Arnold and Crubille [3] present an algorithm for checking alternation depth 1 formulas of μ-calculus which is linear in the size of the model and quadratic in the size of the formula. Cleaveland and Steffen [23] improve this result by making the algorithm linear also in the size of the formula. Andersen [1], and similarly Vergauwen and Lewi [101], show how model checking alternation depth 1 formulas amounts to the evaluation of *Boolean graphs*, resulting also in linear time techniques for model checking alternation depth 1 formulas. Even more expressive subsets of μ-calculus have been investigated by Bhat and Cleaveland [8] as well as Emerson et al. [34, 35]. They present polynomial time model checking algorithms for fragments L1 and L2 of the μ-calculus.

The fragment L1 of the μ-calculus is the formulas formed by the following rules:

1. constants ($\bot$, $\top$) and variables ($X \in \mathcal{Z}$) are L1 formulas
2. if Φ_1 and Φ_2 are L1 formulas then $\Phi_1 \vee \Phi_2$ is a L1 formula
3. if Φ_1 is a L1 formula and $a \in \mathcal{L}$ then $\langle a \rangle \Phi$ is a L1 formula

Property	Formula
No deadlock can occur (i.e. in all states some action is enabled).	$\nu Z.(\langle - \rangle \top \wedge [-]Z)$
An *error* action does not occur along any execution path.	$\nu Z.([error]\bot \wedge [-]Z)$
A *send* action can always eventually be followed by a *receive* action.	$\nu X.([-]X \wedge [send](\mu Y.(\langle - \rangle Y \vee \langle receive \rangle \top)))$
There are no executions where a *request* action is enabled infinitely often but occurs only finitely often.	$\nu X.\mu Y.\nu Z.([request]X \wedge ([request]\bot \vee [-request]Y) \wedge [-request]Z)$

Figure 1: Examples of properties expressed in the μ-calculus.

4. if Φ_1 is a L1 formula and $X \in \mathcal{Z}$ then $\mu X.\Phi_1$ is a L1 formula

5. if Φ_1 is a L1 formula and $X \in \mathcal{Z}$ then $\nu X.\Phi_1$ is a L1 formula

We say that a variable $X \in \mathcal{Z}$ is a free variable in a formula Φ if there is an occurrence of X in Φ which is not in the scope of some μX or νX. A formula without any free variables is called a closed formula. The fragment L2 includes L1 and, in addition, allows formulas of the forms $\Phi_1 \wedge \Phi_2$ and $[a]\Phi_1$ provided that Φ_1 is a closed formula. In [34, 35], the fragment L2 is shown to be exactly as expressive as the logic ECTL* [100].

The μ-calculus allows to express very concisely a wide range of useful properties of systems. Figure 1 shows some typical examples of such properties encoded as the μ-calculus formulas (for a detailed survey of the use of fixpoint operators, see [13]). More examples of formulas expressing system properties will be given in Section 8.

2.3 Boolean Equation Systems

We will give in this subsection a short presentation of Boolean equation systems. Essentially, a Boolean equation system is a sequence of fixpoint equations over Boolean variables, with associated signs, μ and ν, specifying the polarity of the fixed points. The equations are of the form $\sigma x = \alpha$, where α is a positive Boolean expression. The sign, σ, is μ if the equation is a least fixed point equation and ν if it is a greatest fixed point equation. In the following subsections, we will first define positive Boolean expressions, and then define the syntax and semantics of Boolean equation systems.

Syntax of Boolean Equation Systems

Let $\mathcal{X} = \{x_1, x_2, ..., x_n\}$ be a set of Boolean variables. The set of *positive Boolean expressions* over $\mathcal{X}$ is denoted by $B(\mathcal{X})$, and is given by the grammar

$$\alpha ::= \ 0 \mid 1 \mid x_i \mid (\alpha_1 \wedge \alpha_2) \mid (\alpha_1 \vee \alpha_2)$$

where 0 stands for *false*, 1 stands for *true*, and $x_i \in \mathcal{X}$. In the positive

Boolean expressions, we often omit the parentheses when they are not needed to indicate the precedence.

A Boolean equation system (BES), denoted by $\mathcal{E}$, with variables from $\mathcal{X}$ is a sequence of Boolean equations defined in the following way.

Definition 3 (The equations of a Boolean equation system) *A Boolean equation is of the form $(\sigma_i x_i = \alpha_i)$, where $\sigma_i \in \{\mu, \nu\}$, $x_i \in \mathcal{X}$, and $\alpha_i \in B(\mathcal{X})$.*
A Boolean equation system consists of a sequence of Boolean equations

$$(\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2) \dots (\sigma_n x_n = \alpha_n)$$

where the left-hand sides of the equations are all different.

The priority ordering on variables and equations of a Boolean equation system is important for it ensures the existence of a unique solution. This ordering is reflected in an *alternation hierarchy* $\mathcal{H}$ of Boolean equation systems. Intuitively, alternation hierarchy is introduced in order to reflect the nestings of consequtive equations having the same fixpoint sign.

Definition 4 (Alternation hierarchy of a Boolean equation system) *Let*

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2) \dots (\sigma_n x_n = \alpha_n)$$

be a Boolean equation system. Alternation hierarchy *$\mathcal{H}$ of $\mathcal{E}$ is a mapping $\mathcal{H} : \{x_1, x_2, \dots, x_n\} \to \mathbb{N}$ given by:*

$$\mathcal{H}(x_1) = \begin{cases} 0 & \text{if } \sigma_1 = \nu, \\ 1 & \text{otherwise;} \end{cases}$$

and for all $(1 < i \leq n)$

$$\mathcal{H}(x_i) = \begin{cases} \mathcal{H}(x_{i-1}) & \text{if } \sigma_{i-1} = \sigma_i, \\ \mathcal{H}(x_{i-1}) + 1 & \text{otherwise.} \end{cases}$$

Notice that the alternation hierarchy is obtained by dividing the variables of the Boolean equation system into layers within which only fixpoint signs of the same kind occur. A related definition of hierarchical systems of equations is given in [88].

The notion of alternation hierarchy will play an essential role later in our encodings. We next consider a simple example demonstrating the notion of alternation hierarchy.

Example 5 *Let $\mathcal{X}$ be the set $\{x_1, x_2, x_3\}$ and assume we are given a sequence of Boolean equations*

$$\mathcal{E}_1 \equiv (\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee x_3)(\nu x_3 = x_3).$$

The alternation hierarchy $\mathcal{H}$ of $\mathcal{E}_1$ is given by $\mathcal{H}(x_1) = 0$, $\mathcal{H}(x_2) = 1$ and $\mathcal{H}(x_3) = 2$.

Before turning to the semantics of Boolean equation systems, let us first define some useful syntactic notions. In order to formally estimate the computational costs we need to define the *size* and the *alternation depth* of Boolean equation systems.

Definition 6 (The size of a Boolean equation system) *The size of a Boolean equation system*

$$(\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\dots(\sigma_n x_n = \alpha_n)$$

is

$$\sum_{i=1}^{n} 1 + |\alpha_i|$$

where $|\alpha_i|$ is the number of variable occurrences in α_i.

We have taken a definition of alternation depth based on the sequential occurrences of μ's and ν's in a Boolean equation system. More formally, the notion of alternation depth can be defined as follows.

Definition 7 (The alternation depth of a Boolean equation system) *Let*

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\dots(\sigma_n x_n = \alpha_n)$$

be a Boolean equation system. The alternation depth *of $\mathcal{E}$, denoted by $ad(\mathcal{E})$, is defined in the following way. If $\sigma_1 = \sigma_2 = \dots = \sigma_n$, then $\mathcal{E}$ is alternation-free and $ad(\mathcal{E}) = 1$. The system $\mathcal{E}$ is alternating, if it is not alternation-free. If $\mathcal{E}$ is alternating, then $ad(\mathcal{E}) = (1 + k)$ where k is the number of variables x_j $(1 \leq j < n)$ such that $\sigma_j \neq \sigma_{j+1}$.*

An alternative definition of alternation depth which abstracts from the syntactical appearance can be found in Definition 3.34 of [70]. The idea there is that to determine the alternation depth only chains of equations in a Boolean equation system must be followed that depend on each other.

Notice that for each equation system $\mathcal{E}$ with variables from $\mathcal{X}$ we have that $ad(\mathcal{E}) \leq |\mathcal{X}|$.

It is seen in Section2.5 that the notion of the alternation depth of a Boolean equation system is closely related to the notion of a *priority* in parity games.

As pointed out in [70] (see Proposition 3.31), for each system $\mathcal{E}$ there is another system $\mathcal{E}'$ in a *standard form* such that $\mathcal{E}'$ preserves the solution of $\mathcal{E}$ and has size linear in the size of $\mathcal{E}$.

Definition 8 (Standard form Boolean equation systems) *A Boolean equation system*

$$(\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\dots(\sigma_n x_n = \alpha_n)$$

is in standard form if, for $1 \leq i \leq n$, the right-hand side expression α_i has the form $y \circ z$ or y where $\circ \in \{\wedge, \vee\}$ and $y, z \in \{x_1, x_2, \dots, x_n\} \cup \{1, 0\}$.

The transformation from unrestricted Boolean equation systems to standard form systems is based on the fact that, for each Boolean subformula occurring in the right-hand side formula of a Boolean equation we can introduce new variables and equations such that the solution is preserved (for a more detailed description of this process, see [70]). For example, $(\sigma x = y \circ \alpha)$ with $\circ \in \{\wedge, \vee\}$ and $\sigma \in \{\mu, \nu\}$ we can always introduce a new variable z and replace the equation by two consecutive equations $(\sigma x = y \circ z)(\sigma z = \alpha)$.

In the sequel, we will restrict to standard form Boolean equation systems.

Local Semantics of Boolean Equation Systems

The semantical interpretation of Boolean equation systems is such that each system has a uniquely determined solution. Informally, the solution is a valuation assigning a constant value in $\{0, 1\}$ to variables occurring in the system. More precisely, the solution is a truth assignment to the variables $\{x_1, x_2, ..., x_n\}$ satisfying the fixed point equations as defined below in Definition 9 (see also, e.g., [1, 70]).

In particular, we are interested in the value of the left-most variable x_1, and we call this value the solution of a Boolean equation system. Such a local solution can be characterized in the following way.

Let α be a closed positive Boolean expression (i.e. without occurrences of variables in $\mathcal{X}$). Then α has a uniquely determined value in the set $\{0, 1\}$ which we denote by $\|\alpha\|$. This value of closed positive Boolean expressions is trivially defined by using the usual semantics for Boolean formulas.

We define a substitution for positive Boolean expressions. Given Boolean expressions $\alpha, \beta \in B(\mathcal{X})$, let $\alpha[x := \beta]$ denote the expression α where all occurrences of variable x are substituted by β simultaneously.

Similarly, we extend the definition of substitutions to Boolean equation systems in the following way. Let $\mathcal{E}$ be a Boolean equation system over $\mathcal{X}$, and let $x \in \mathcal{X}$ and $\alpha \in B(\mathcal{X})$. A substitution $\mathcal{E}[x := \alpha]$ means the operation where $[x := \alpha]$ is applied simultaneously to all right-hand sides of equations in $\mathcal{E}$. We suppose that substitution $\alpha[x := \alpha]$ has priority over $\mathcal{E}[x := \alpha]$.

The definition of the local solution is as follows.

Definition 9 (The local solution to a Boolean equation system) *The solution to a Boolean equation system $\mathcal{E}$, denoted by $[\![\mathcal{E}]\!]$, is a Boolean value inductively defined by*

$$[\![\mathcal{E}]\!] = \begin{cases} \|\alpha[x := b_\sigma]\| & \textit{if } \mathcal{E} \textit{ is of the form } (\sigma x = \alpha) \\ [\![\mathcal{E}'[x := \alpha[x := b_\sigma]]]\!] & \textit{if } \mathcal{E} \textit{ is of the form } \mathcal{E}'(\sigma x = \alpha) \end{cases}$$

where b_σ is 0 *when $\sigma = \mu$, and b_σ is* 1 *when $\sigma = \nu$.*

The following example illustrates the above definition of the solution.

Example 10 *Let $\mathcal{X}$ be the set $\{x_1, x_2, x_3\}$ and assume we are given a Boolean equation system*

$$\mathcal{E}_1 \equiv (\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee x_3)(\nu x_3 = x_3).$$

The local solution, $[\![\mathcal{E}_1]\!]$, of variable x_1 in $\mathcal{E}_1$ is given by
$[\![(\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee x_3)(\nu x_3 = x_3)]\!] =$
$[\![(\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee x_3)[x_3 := 1]]\!] =$
$[\![(\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee 1)]\!] =$
$[\![(\nu x_1 = x_2 \wedge x_1)[x_2 := (x_1 \vee 1)]]\!] =$
$[\![(\nu x_1 = (x_1 \vee 1) \wedge x_1)]\!] =$
$\|((1 \vee 1) \wedge 1)\| = 1$

The above definition of the semantics is local in the sense that it characterizes a value only for the least variable x_1. In contrast, the definition of a solution to a Boolean equation system defined in the next section gives a global solution, i.e. values for all variables.

Global Semantics of Boolean Equation Systems

As opposed to Definition 9, the global semantics of Boolean equation systems provides a uniquely determined solution to each variable of a Boolean equation system. According to the global semantics a solution is a valuation θ assigning a constant value in $\{0,1\}$ to all variables occurring in a Boolean equation system. We define here the global semantics as an alternative semantics of Boolean equation systems.

Let θ stand for a valuation which is a function $\theta : \mathcal{X} \to \{0,1\}$. Let $\theta[x{:=}a]$ denote the valuation that coincides with θ for all variables except x which has the value a.

Let $\|\alpha\|(\theta)$ denote the Boolean value of the positive Boolean formula α obtained by replacing each free variable x in α by $\theta(x)$ and evaluating the resulting closed formula. Here, we use a different notation than in the semantics of μ-calculus formulas to explicitly distinguish the two semantics.

The global definition of a solution to a Boolean equation system is given as follows (see also, e.g., Definition 3.3 in [70]). We use similar semantic notation $[\![\mathcal{E}]\!]$ for both local and global semantics because the local semantics in Definition 9 coincides with the following global semantics in Definition 11.

Definition 11 (The global solution to a Boolean equation system) *The global solution to a Boolean equation system $\mathcal{E}$ relative to valuation θ, denoted by $[\![\epsilon]\!]\theta$, is inductively defined as*

$$[\![\epsilon]\!]\theta = \theta$$

$$[\![(\sigma_i x_i = \alpha_i)\mathcal{E}]\!]\theta = \begin{cases} [\![\mathcal{E}]\!]\theta[x_i{:=}\mathit{MIN}(x_i,\alpha_i,\mathcal{E},\theta)] & \text{if}\, \sigma_i = \mu \\ [\![\mathcal{E}]\!]\theta[x_i{:=}\mathit{MAX}(x_i,\alpha_i,\mathcal{E},\theta)] & \text{if}\, \sigma_i = \nu \end{cases}$$

where

$$\mathit{MIN}(x_i,\alpha_i,\mathcal{E},\theta) = \min\{a \in \{0,1\} \mid \|\alpha_i\|([\![\mathcal{E}]\!]\theta[x_i{:=}a]) = a\}$$
$$\mathit{MAX}(x_i,\alpha_i,\mathcal{E},\theta) = \max\{a \in \{0,1\} \mid \|\alpha_i\|([\![\mathcal{E}]\!]\theta[x_i{:=}a]) = a\}$$

and ϵ denotes an empty Boolean equation system.

The above definition of a global solution to a Boolean equation system has quite a complex nature, as exemplified with a simple system below.

Example 12 *Let $\mathcal{X}$ be the set $\{x_1, x_2\}$ and assume we are given a Boolean equation system $\mathcal{E}_2 \equiv (\mu x_1 = x_2)(\nu x_2 = x_1)$. According to Definition 11, the solution to this system can be calculated as follows. Consider an arbitrary valuation θ_0 that maps $\mathcal{X}$ to $\{0,1\}$. First, we calculate*
$\theta_1 = \theta_0[x_1{:=}\mathit{MIN}(x_1, x_2, (\nu x_2 = x_1), \theta_0)]$.
Thus, we calculate
$\mathit{MIN}(x_1, x_2, (\nu x_2 = x_1), \theta_0) =$
$\min\{a \in \{0,1\} \mid \|x_2\|([\![(\nu x_2 = x_1)]\!]\theta_0[x_1{:=}a]) = a\}$
and within this
$[\![(\nu x_2 = x_1)]\!]\theta_0[x_1{:=}\, a]$.
Now, we have
$\min\{a \in \{0,1\} \mid a = a\} = \min\{0,1\} = 0$.
Thus, $\theta_1 = \theta_0[x_1{:=}0]$. Hence, the solution to $\mathcal{E}_2$ is:
$[\![(\mu x_1 = x_2)(\nu x_2 = x_1)]\!]\theta_0 =$

$[\![(\nu x_2 = x_1)]\!]\theta_0[x_1{:=}MIN(x_1, x_2, (\nu x_2 = x_1), \theta_0)] =$
$[\![(\nu x_2 = x_1)]\!]\theta_0[x_1{:=}0] =$
$[\![(\nu x_2 = x_1)]\!]\theta_1 =$
$[\![\epsilon]\!]\theta_1[x_2{:=}MAX(x_2, x_1, \epsilon, \theta_1)] =$
$\theta_1[x_2{:=}MAX(x_2, x_1, \epsilon, \theta_1)]$.
Here, we have $MAX(x_2, x_1, \epsilon, \theta_1) =$
$\max\{a \in \{0,1\} \mid \|x_1\|([\![\epsilon]\!]\theta_1[x_2{:=}a]) = a\} =$
$\max\{a \in \{0,1\} \mid \|0\| = a\} =$
$\max\{a \in \{0,1\} \mid 0 = a\} =$
$\max\{0\} = 0$.
Therefore, the global solution is is given by $\theta_2 = \theta_1[x_2{:=}0]$. *In other words, the solution to both variables* x_1 *and* x_2 *is* 0.

When applied to non-trivial Boolean equation systems, i.e. to systems involving more than two simple equations, the above global definition of the semantics is quite tedious to apply by hand. Therefore, in this thesis we have mainly used the local semantics given in Definition 9.

It will be seen in Section 3 that the local semantics in Definition 9 can be used to find the global solutions as well. We have the following proposition.

Proposition 13 *Given a Boolean equation system* $\mathcal{E}$, *let* $[\![\mathcal{E}]\!]$ *be the local solution to* $\mathcal{E}$ *and let* θ *be the global solution to* $\mathcal{E}$. *Then, the following are equivalent:*

1. $[\![\mathcal{E}]\!] = 1$;
2. $\theta(x_1) = 1$.

There are also alternative characterizations of the solution to a Boolean equation system which provide more insight, for instance, Proposition 3.6 in [70] and Definition 1.4.10 in [4].

Graph Representation of Boolean Equation Systems
Given a Boolean equation system we can define a variable *dependency graph* similar to a *Boolean graph* in [1] which provides a representation of the dependencies between the variables.

Definition 14 (Dependency graph) *Let* $\mathcal{E}$ *be a standard form Boolean equation system*

$$(\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n).$$

The dependency graph of $\mathcal{E}$ *is a directed graph* $G_{\mathcal{E}} = (V, E, \ell)$ *where*

- $V = \{i \mid 1 \leq i \leq n\} \cup \{\bot, \top\}$ *is the set of nodes;*
- $E \subseteq V \times V$ *is the set of edges such that for all equations* $(\sigma_i x_i = \alpha_i)$
  - $(i, j) \in E$, *if a variable* $x_j \in \alpha_i$;
  - $(i, \bot) \in E$, *if* 0 *occurs in* α_i;
  - $(i, \top) \in E$, *if* 1 *occurs in* α_i;
  - $(\bot, \bot), (\top, \top) \in E$;

- *$\ell : V \to \{\mu, \nu\}$ is the node labelling defined by $\ell(i) = \sigma_i$ for $1 \leq i \leq n$, $\ell(\bot) = \mu$, and $\ell(\top) = \nu$.*

Observe that in the definition above the sink nodes with self-loops, $\bot$ and $\top$, represent the constants 0 and 1. The nodes are numbers which gives them a linear order. This ordering on nodes will be used later on in the thesis to give a characterisation of the solution. The linear ordering on nodes is extended to $\bot$ and $\top$ by putting them highest up in the ordering. The ordering between $\bot$ and $\top$ is assumed to be $\bot < \top$ (although it is irrelevant). We often omit the labelling function ℓ from the dependency graphs when it is not of particular importance.

We now define some graph-theoretic notions concerning dependency graphs of Boolean equation systems which will be used throughout this thesis. To be precise, these notions will be singled out as separate definitions because there exist several variants of definitions for the notions in the literature.

Definition 15 (Paths of dependency graphs) *A path of length k from a node i to a node j in a dependency graph $G_{\mathcal{E}} = (V, E, \ell)$ is a sequence $(v_0, v_1, v_2, ..., v_k)$ of nodes such that $i = v_0$, $j = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, ..., k$. The path contains the nodes $v_0, v_1, v_2, ..., v_k$.*

Definition 16 (Reachability) *A node j is reachable from node i in a dependency graph $G_{\mathcal{E}}$, if there is a path in $G_{\mathcal{E}}$ from i to j.*

Definition 17 (Cycles) *A path $(v_0, v_1, v_2, ..., v_k)$ appearing in a dependency graph $G_{\mathcal{E}}$ is a cycle if $v_0 = v_k$ and it is of length $k \geq 2$.*

Based on these standard concepts, we may introduce some additional terms.

We say that a variable x_i *depends on* variable x_j in a Boolean equation system $\mathcal{E}$, if the dependency graph $G_{\mathcal{E}}$ of $\mathcal{E}$ contains a path from node i to node j.

It is said that two variables $x_i, x_j \in \mathcal{X}$ are *mutually dependent*, if x_i depends on x_j and vice versa. In general, it is said that a Boolean equation system is *alternation-free*, if for all pairs of variables $x_i, x_j \in \mathcal{X}$ it holds that x_i and x_j being mutually dependent implies that $\sigma_i = \sigma_j$ holds. Otherwise, the Boolean equation system is said to be *alternating*.

An important notion, which will be used in our mapping from Boolean equation systems to normal logic programs, is self-dependency. We say that a variable x_i is *self-dependent*, if x_i depends on itself in such a way that no variable x_j with $j < i$ occurs in this chain of dependencies. More precisely, the notion of self-dependency can be defined in the following way.

Definition 18 *Given a Boolean equation system $\mathcal{E}$, let $G_{\mathcal{E}} = (V, E, \ell)$ be its dependency graph and $k \in V$. We define the graph $G{\restriction}k = (V, E{\restriction}k, \ell)$ by taking*

- *$E{\restriction}k = \{\langle i, j\rangle \in E \mid i \geq k \text{ and } j \geq k\}$.*

The variable x_k is said to be self-dependent in the system $\mathcal{E}$, if node k is reachable from itself in the graph $G{\restriction}k$.

$$\begin{array}{l}(\nu x_1 = x_2 \wedge x_1)\\(\mu x_2 = x_1 \vee x_3)\\(\nu x_3 = x_3)\end{array}$$

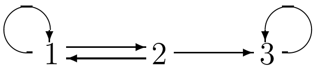


Figure 2: The dependency graph of Boolean equation system $\mathcal{E}_1$ in Example 10.

As with dependency graphs, we often omit the labelling function ℓ from a restricted graph $G{\restriction}k$ when it is not of particular importance. Let us consider a simple example below.

Example 19 *Consider the Boolean equation system $\mathcal{E}_1$ of Example 10. The dependency graph of $\mathcal{E}_1$ is depicted in Figure 2. The system $\mathcal{E}_1$ is in standard form and is alternating, because it contains alternating fixed points with mutually dependent variables having different signs, like x_1 and x_2 with $\sigma_1 \neq \sigma_2$. Notice that two variables are mutually dependent when they appear on a same cycle in the dependency graph. The variables x_1 and x_3 of $\mathcal{E}_1$ are self-dependent, but x_2 is not as $G{\restriction}2 = (\{1, 2, 3\}, \{(2, 3), (3, 3)\})$ contains no loop from node 2 to itself.*

Finally, we define maximal *strongly connected components* of a dependency graph. This definition will be needed in partitioning a Boolean equation system into blocks as explained in Section 3.

Definition 20 (Strongly connected components) *A strongly connected component (SCC) in a graph $G = (V, E, \ell)$ is a set of nodes $W \subseteq V$ such that, for all nodes $k, m \in W$, m is reachable from k in E. A strongly connected component W is called maximal, if there does not exist a larger set $W' \subseteq V$ of nodes such that $W \subset W'$ and W' is a strongly connected component. A maximal strongly connected component is called trivial, if it consists of one vertex $v \in V$, and there is no edge $(v, v) \in E$. A maximal strongly connected component is non-trivial, if it is not trivial.*

Most of the above graph-theoretic notions and definitions are very standard in the literature.

2.4 μ-Calculus Model Checking with Boolean Equation Systems

As mentioned before, instead of treating μ-calculus expressions together with their semantics we prefer to work with the more flexible formalism of Boolean equation systems. Boolean equation systems provide a useful framework for studying verification problems of finite-state concurrent systems because μ-calculus expressions can easily be translated into this simple formalism. A pleasant feature of Boolean equation systems is that they give a simple representation of the μ-calculus model checking problem.

Φ	$(\Phi)_s$	Φ	$(\Phi)_s$
$\bot$	0	$\top$	1
$\Phi_1 \vee \Phi_2$	$(\Phi_1)_s \vee (\Phi_2)_s$	$\Phi_1 \wedge \Phi_2$	$(\Phi_1)_s \wedge (\Phi_2)_s$
$\langle a \rangle \Phi$	$\bigvee_{s \xrightarrow{a} t} (\Phi)_t$	$[a]\Phi$	$\bigwedge_{s \xrightarrow{a} t} (\Phi)_t$
X	x_s	$\sigma X.\Phi_1$	$\sigma x_s = (\Phi_1)_s$

Figure 3: The translation from a μ-calculus formula to a Boolean equation system.

In this subsection, we demonstrate the *standard translation* from a μ-calculus formula and a labelled transition system to a Boolean equation system as defined in [75]. Similar translations serving the same purpose are presented, for example, in [1, 4, 70].

The transformation maps a μ-calculus formula Φ and a transition system $\mathcal{T}$ to a Boolean equation system by treating (state, variable) pairs as Boolean variables. Informal idea of the translation is to strip away the linearization of the μ-calculus formula Φ imposed by text, and then map the μ-calculus expression Φ to Boolean expressions at respective states of the transition system $\mathcal{T}$. More precisely, the translation proceeds as follows.

First, additional fresh variables may be introduced at appropriate places of Φ to ensure that in every subformula $\sigma X.\Phi'$ of Φ with $\sigma \in \{\mu, \nu\}$, Φ' contains a single Boolean or modal operator. This may be done in order to obtain only disjunctive or conjunctive formulas in the right-hand side Boolean expressions of the resulting Boolean equation system but is not necessary for the translation.

Then, a sequence of equations is created for each closed fixed point subformula $\sigma X.\Phi'$ of Φ. Each closed fixed point subformula $\sigma X.\Phi'$ is translated into a sequence $(\sigma X_s (\Phi')_s)_{s \in S}$ of equations where variables X_s express that state s satisfies variable X and the right-hand side Boolean formulas are obtained using the translation in Figure 3.

By using this technique, the size of the Boolean equation system resulting from the transformation is at most $O(m \times n)$ where m is the length of a formula and n is the size of a transition system. Also, there exists a polynomial mapping from a Boolean equation system to a μ-calculus formula and a labelled transition system (see Theorem 5.2 in [70]).

The following example illustrates the standard translation from μ-calculus to Boolean equation systems.

Example 21 *Consider the following μ-calculus formula*

$$\nu Z.([-]Z \wedge \langle - \rangle \top)$$

which expresses the freedom of deadlocks property. Consider the following labelled transition system

$$\mathcal{T} = (\{1, 2, 3, 4\}, \{1 \xrightarrow{a} 2, 1 \xrightarrow{a} 4, 2 \xrightarrow{a} 3, 3 \xrightarrow{a} 2\})$$

depicted in Figure 4. We demonstrate how to construct the corresponding Boolean equation system shown in Figure 4.

Clearly, the given formula corresponds to the closed fixed point formula

$$\nu Z.([a]Z \wedge \langle a \rangle \top)$$

without the shorthands $[-]$ *and* $\langle - \rangle$. *This is first translated into a sequence of equations, for all* $s \in S$,

$$(\nu\, z_s = ([a]z \wedge \langle a \rangle \top)_s)$$

with one equation for each state $s \in S$.

Next, using the translation shown in Figure 3 to obtain the right-hand side Boolean formulas, we get the sequence of equations, for all $s \in S$:

$$(\nu\, z_s = ([a]z)_s \wedge (\langle a \rangle \top)_s)$$

This sequence is translated to the Boolean equations, for all $s \in S$,

$$(\nu\, z_s = (\bigwedge_{s \xrightarrow{a} s'} z_{s'}) \;\wedge\; (\bigvee_{s \xrightarrow{a} s'} (\top)_{s'}))$$

where each variable z_s *expresses that state* s *satisfies variable* Z *of the* μ-*calculus formula.*

Given the labelled transition system $\mathcal{T}$ *in Figure 4, the above equations translate to the following sequence of Boolean equations:*

$$\begin{array}{l} (\nu z_1 = (z_2 \wedge z_4) \wedge (1 \vee 1)) \\ (\nu z_2 = z_3 \wedge 1) \\ (\nu z_3 = z_2 \wedge 1) \\ (\nu z_4 = 1 \wedge 0) \end{array}$$

Notice in this step that an empty disjunction is written as 0 *and an empty conjunction is written as* 1, *in particular in the last equation.*

By renaming the variables we get the Boolean equation system over $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$

$$\begin{array}{l} (\nu x_1 = (x_2 \wedge x_4) \wedge (1 \vee 1)) \\ (\nu x_2 = x_3 \wedge 1) \\ (\nu x_3 = x_2 \wedge 1) \\ (\nu x_4 = 1 \wedge 0) \end{array}$$

which corresponds to the given model checking problem.

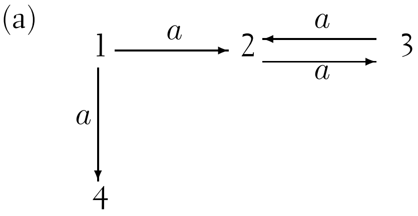
It can be seen that the global solution to the Boolean equation system resulting from the above translation is $\theta[x_1 := 0, x_2 := 1, x_3 := 1, x_4 := 0]$. *As expected, the formula*

$$\nu Z.([-]Z \wedge \langle - \rangle \top)$$

holds only in states 2 *and* 3 *as the solution to corresponding variables* x_2 *and* x_3 *is* 1.

Additional examples of the mapping will be given in Section 8.

In addition to representing μ-calculus model checking, Boolean equation systems give a useful formalism to encode other problems encountered in a



(b)

$$\nu Z.([-]Z \wedge \langle - \rangle \top)$$

(c)

$$\begin{array}{l}(\nu x_1 = (x_2 \wedge x_4) \wedge (1 \vee 1))\\ (\nu x_2 = x_3 \wedge 1)\\ (\nu x_3 = x_2 \wedge 1)\\ (\nu x_4 = 1 \wedge 0)\end{array}$$

Figure 4: Example labelled transition system (a), μ-calculus formula for deadlock freedom (b) and corresponding Boolean equation system (c).

wide range of problem domains. For instance, such problem domains include automatic program analysis (e.g. abstract interpretation of functional and logic programming languages [37]), and formal verification of concurrent programs (e.g. equivalence checking [2, 23, 64, 75] and partial order reduction [82]). In addition, Boolean equation systems could also be applied as a useful formalism in synthesis [17]. For instance, both related formalisms of the μ-calculus and the parity games are already being used in the realm of synthesis (for example, see [62, 5]). In general, Boolean equation system solvers can be used as general purpose tools targeted to handle all these kinds of problems, but in this thesis we will mainly concentrate on μ-calculus model checking.

2.5 Boolean Equation Systems and Parity Games

Recall from Section 1.1 that the μ-calculus model checking has been studied in other frameworks than Boolean equation systems too. Importantly, the model checking problem for the μ-calculus is equivalent to the problem of determining a winner in a parity game [33]. We refer the reader to Chapter 4 in [4] for a detailed presentation of parity games.

In this subsection, we review parity games and demonstrate the connection between Boolean equation systems and parity games. We use this connection to prove some useful results for the purposes ot this thesis. Namely, in Section 6 and Section 7 we re-use results, which are originally shown for parity games, to develop new solution techniques for Boolean equation systems.

A parity game is a tuple $G = (V, E, v_0, \Omega)$ where (V, E) is a finite, directed graph and V is partitioned into two sets $V_\exists$ and $V_\forall$, $v_0 \in V$ is the starting node, and $\Omega : V \to \mathbb{N}$ is a priority function. Furthermore, the game G is assumed to be total, i.e. for every $v \in V$ there is a $w \in V$ with $(v, w) \in E$. Without loss of generality, we assume that the starting node v_0 is always one of the nodes with the smallest priority, i.e. for every $v \in V$ such that $v_0 \neq v$, $\Omega(v) \geq \Omega(v_0)$.

A *play* of G is an infinite path $\pi = v_0 v_1 v_2 \ldots$ through G starting in v_0. It is constructed in the following way. Given a node $v \in V_x$ with $x \in \{\exists, \forall\}$, player x chooses a $w \in V$ with $(v, w) \in E$ and the construction of the play continues with w.

Given a play $\pi = v_0 v_1 \ldots$ let $\inf \pi = \{v \in V \mid$ there are infinitely many $i \in \mathbb{N}$ s.t. $v = v_i\}$. Player $\exists$ wins the play $\pi = v_0 v_1 \ldots$ if $\min\{\Omega(v) \mid v \in \inf \pi\}$ is even. If it is odd then player $\forall$ wins the play π.

A strategy for player x is a function $\sigma_x : V_x \rightarrow V$ that tells player x which choice to make depending on the current construction of a play.

A strategy σ_x for player x is called a *winning strategy*, if x wins every play by using σ_x, no matter how the other player plays.

Given a parity game G and a strategy $\sigma_\exists$ for player $\exists$ we write $G|_\sigma$ for the parity game that is induced by $\sigma_\exists$ on G. Formally, $G|_\sigma = (V, E \cap (V_\forall \times V \cup \{(v, \sigma_\exists(v)) \mid v \in V_\exists\}), v_0, \Omega)$. Note that $G|_\sigma$ is indeed a subgame of G, i.e. every play π in $G|_\sigma$ with winner x is also a play in G that is won by player x.

The problem of solving a parity game $G = (V, E, v_0, \Omega)$ is to determine whether or not player $\exists$ has a winning strategy for G.

Solving a parity game is closely related to the problem of solving other infinite games, for instance mean pay-off games [53, 103]. Solving a parity game is equivalent to the model checking problem for the modal μ-calculus [33].

From the results in [34, 35, 33], it follows already that solving a parity game is also one of the problems in the complexity class NP $\cap$ co-NP. Furthermore, the problem of solving a parity game is known to be in the class UP $\cap$ co-UP [53]. Several algorithms for solving parity games have been suggested but, at the time of writing this thesis, none of them provably runs in deterministic polynomial time.

The algorithms for solving a parity game include, for instance, a strategy improvement algorithm suggested by Jurdziński and Vöge [102]. An implementation of the strategy improvement algorithm is presented in [87]. A randomised and subexponential algorithm for solving parity games is due to Björklund, Sandberg, and Vorobyov [15]. An algorithm with a good asymptotic time complexity is Jurdziński's *small progress measures* procedure [54]. It is exponential in the number of odd priorities occurring in the game, i.e. approximately in the half of the maximal priority. An implementation of the small progress measure algorithm is presented in [36]. Recently, Jurdziński, Paterson and Zwick have suggested the first deterministic and subexponential ($n^{O(\sqrt{n})}$ with n the number of nodes in the game) algorithm for solving parity games [55].

For the purposes of this thesis, we now demonstrate the connections between parity games and Boolean equation systems. In this way, we can use the results for parity games in the setting of Boolean equation systems. In particular, in Section 7 we give an encoding of Boolean equation systems into propositional satisfiability which is based on results behind the Jurdziński's small progress measure algorithm [54].

The problem of finding a solution to a Boolean equation system can be shown to be equivalent to the problem of determining a winner in a parity game. Now, we give summarised arguments for this result.

For an original proof of the following result see, e.g., Theorem 8.9 of [70].

Theorem 22 *Let $G = (V, E, v_0, \Omega)$ be a parity game. Then, a Boolean equation system $\mathcal{E}$ can be constructed from G such that Player $\exists$ has a winning strategy in G if and only if $[\![\mathcal{E}]\!] = 1$.*

Proof:
Given a parity game $G = (V, E, v_0, \Omega)$ we construct a corresponding Boolean equation system $\mathcal{E}$. The set of variables $\mathcal{X}$ of the Boolean equation system

is V. The equations of the Boolean equation system are obtained in the following way:

- for all $v \in V_\exists$ s.t. $\Omega(v)$ is even, there is an equation $(\nu v = \bigvee_{(v,w)\in E} w)$,
- for all $v \in V_\forall$ s.t. $\Omega(v)$ is even, there is an equation $(\nu v = \bigwedge_{(v,w)\in E} w)$,
- for all $v \in V_\exists$ s.t. $\Omega(v)$ is odd, there is an equation $(\mu v = \bigvee_{(v,w)\in E} w)$, and
- for all $v \in V_\forall$ s.t. $\Omega(v)$ is odd, there is an equation $(\mu v = \bigwedge_{(v,w)\in E} w)$.

Finally, we need to order the equations. The first equation should be the one corresponding to the starting node of the parity game. For every other $s, t \in V$: if $\Omega(s) < \Omega(t)$, then the equation with variable s as its left-hand side variable must be before the equation with t as the left-hand side variable. Notice that the alternation hierarchy obtained is such that, for all $v \in V$, $\mathcal{H}(v) = \Omega(v)$.

It remains to show that Player $\exists$ has a winning strategy in G if and only if $[\![\mathcal{E}]\!] = 1$.

First, we consider the proof from right to left. Suppose $[\![\mathcal{E}]\!] = 1$. By Proposition 3.36 in [70], there is a conjunctive subsystem $\mathcal{E}'$ of $\mathcal{E}$ (see Def. 48) with the solution $[\![\mathcal{E}']\!] = 1$. Alternatively, this also follows from Lemma 49 in this thesis which gives a strenghtened result of Proposition 3.36 in [70]. Let $G' = (V', E', \ell')$ be the dependency graph of $\mathcal{E}'$. We construct a strategy $\sigma_\exists$ for Player $\exists$ as follows. Take a function $\sigma_\exists : V_\exists \to V$ such that

- if $i \in V_\exists$ and $(i, j) \in E'$, then $\sigma_\exists(i) = j$.

It can be shown that $\sigma_\exists$ is a winning strategy for Player $\exists$ by noticing that the graph G' does not contain any infinite path starting from node 1 where the smallest node occurring infinitely often has the label μ. For instance, this follows from Lemma 41 in Section 5.1 which gives a graph theoretic characterisation of a solution to a conjunctive Boolean equation system. The subgraph $G|_\sigma$ induced by $\sigma_\exists$ on G is isomorfic to the graph (V', E'). Thus, as the order of the equations in the construction of $\mathcal{E}$ follows the priority ordering given by Ω, it follows that $\sigma_\exists$ is a winning strategy for Player $\exists$.

The other direction (from left to right) follows by a similar argument where, for example, the dual Lemmas 50 and 40 can be used. □

There is also a mapping from Boolean equation systems to parity games. For a classical proof of the following result, see Theorem 8.7 of [70].

Theorem 23 *Let $\mathcal{E}$ be a Boolean equation system. A parity game $G = (V, E, v_0, \Omega)$ can be constructed from $\mathcal{E}$ such that player $\exists$ has a winning strategy in G if and only if $[\![\mathcal{E}]\!] = 1$.*

Proof:
Given a standard form Boolean equation system

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2) \ldots (\sigma_n x_n = \alpha_n)$$

with the alternation hierarchy $\mathcal{H}$, we construct a corresponding parity game $G = (V, E, v_0, \Omega)$ such that $[\![\mathcal{E}]\!] = 1$ if and only if player $\exists$ has a winning strategy in G.

Here, we assume that the constant symbols 1 and 0 do not occur in the right-hand side formulas of $\mathcal{E}$. Notice that the constants 1 and 0 can be easily removed from the equations of $\mathcal{E}$ by using the simplification rules given in Section 3.3 such that there remains only equations of the forms $(\sigma_i x_i = x_j)$, $(\sigma_i x_i = x_j \vee x_k)$, $(\sigma_i x_i = x_j \wedge x_k)$, $(\nu x_i = 0)$, and $(\mu x_i = 1)$. Notice that all equations of the forms $(\sigma_i x_i = 1)$ and $(\sigma_i x_i = 0)$ can be easily represented without any constants, e.g. by the equations $(\nu x_i = x_i)$ and $(\mu x_i = x_i)$. Therefore, we may restrict here to consider Boolean equation systems without any constants.

Let $V = \{v_1, v_2, \ldots, v_n\}$. Here, for each variable x_i there is a corresponding node v_i in the game. The set of nodes V is partitioned into two sets $V_\exists$ and $V_\forall$ as follows. For all $1 \leq i \leq n$, if equation α_i is conjunctive then $v_i \in V_\forall$. For all $1 \leq i \leq n$, if equation α_i is disjunctive then $v_i \in V_\exists$. The starting node (i.e. v_0) of the game is v_1. The set of edges $E \subseteq V \times V$ is constructed as follows:

- for all $1 \leq i \leq n$, if a variable x_j appears in α_i then $(v_i, v_j) \in E$.

Finally, the priority function Ω is defined as follows. For all $1 \leq i \leq n$, let $\Omega(v_i) = \mathcal{H}(x_i)$.

As the above construction is the reverse mapping of the one given in the proof of Theorem 22, it follows that Player $\exists$ has a winning strategy in G if and only if $[\![\mathcal{E}]\!] = 1$. □

Notice that, through the above equivalence between parity games and Boolean equation systems, every algorithm for solving parity games can also be seen as an algorithm for solving Boolean equation systems, and vice versa.

Next, we turn to issues concerning logic programs and answer set programming.

2.6 Normal Logic Programs

In this thesis, we will use normal logic programs with the stable model semantics [39] for encoding and solving Boolean equation systems. Therefore, in this subsection we provide a brief introduction to normal logic programs and stable model semantics.

The definitions in this subsection appeared also in [57], and they are very standard. A complete description of these topics and notions can be found, for instance, in [27].

Normal logic programs consist of *rules* of the form

$$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n. \qquad (1)$$

where each $a, b_1, \ldots, b_m, c_1, \ldots, c_n$ is a ground atom. In the normal rule above, a is called the *head* of the rule and $b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$ its *body*.

Given a logic program, its *models* are sets of ground atoms. A set of atoms Δ is said to satisfy an atom a if $a \in \Delta$ and a negative literal not a if $a \notin \Delta$. A rule r of the form (1) is satisfied by Δ if the head a is satisfied whenever every body literal $b_1, \ldots, b_m$, not $c_1, \ldots,$ not c_n is satisfied by Δ and a program Π is satisfied by Δ if each rule in Π is satisfied by Δ.

An essential concept here is a *stable model*. Stable models of a program are sets of ground atoms which satisfy all the rules of the program and are justified by the rules. This is captured using the concept of a *reduct*. As usually, for a program Π and a set of atoms Δ, the reduct Π^Δ can be defined by

$$\Pi^\Delta = \{a \leftarrow b_1, \ldots, b_m. \mid a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n. \in \Pi,$$
$$\{c_1, \ldots, c_n\} \cap \Delta = \emptyset\}$$

That is, a reduct Π^Δ does not contain any negative literals and, therefore, has a unique subset minimal set of atoms satisfying it. This leads to the following definition of stable models.

Definition 24 (Stable models of a logic program) *A set of atoms Δ is called a stable model of a program Π iff Δ is the unique minimal set of atoms satisfying Π^Δ.*

Notice in the above definition that, intuitively, calculating a unique minimal model satisfying a reduct amounts to computing a least fixpoint of a set of rules.

The problem of determining the existence of a stable model of a normal logic program is NP-complete [72].

In the following, we consider a series of examples to illustrate the intuitive idea behind the stable model semantics of logic programs.

Example 25 *Let $\{a, b\}$ be the set of ground atoms. Consider the program:*

$$a \leftarrow \textit{not } b.$$
$$b \leftarrow \textit{not } a.$$

This program has two stable models, namely $\{a\}$ and $\{b\}$. Here, we may either assume ***not b*** *in order to deduce the stable model $\{a\}$ or we may assume* ***not a*** *to deduce the stable model $\{b\}$. However, note that assuming both negative premises would lead to a contradiction; thus, we cannot deduce the stable model $\{\}$ for this program by assuming both* **not a** *and* **not b**. *Note that this is a way to encode a choice between atoms a and b.*

Example 26 *Let $\{a, b, c, d\}$ be the set of ground atoms. Consider the program:*

$$a \leftarrow a.$$
$$b \leftarrow c, d.$$
$$c \leftarrow d.$$
$$d.$$

The above program has only one stable model which is the set $\{b, c, d\}$. The atom c can be deduced from the fact d, and the atom b is included in the stable model because both c and d are included. Notice that the atom a is not included in the stable model because we cannot use positive assumption a to deduce what is to be included in a model.

In the course of this thesis, we will use two extensions which serve as shorthands for normal rules. We will use so-called *integrity constraints*. Integrity constraints are simply rules

$$\leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n. \tag{2}$$

with an empty head. Such a constraint can be seen as a compact shorthand for a rule

$$f \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n, \text{not } f.$$

where f is a new atom.

Notice that a stable model Δ satisfies an integrity constraint (2) only if at least one of its body literals is not satisfied by Δ.

Finally, for expressing the choice of selecting exactly one atom from two possibilities we will make use of *choose-1-of-2 rules* on the left which correspond to the normal rules on the right:

$$1\ \{a_1, a_2\}\ 1. \qquad\qquad a_1 \leftarrow \text{not } a_2. \quad a_2 \leftarrow \text{not } a_1. \quad \leftarrow a_1, a_2.$$

Choose-1-of-2 rules are a simple subclass of cardinality constraint rules presented in [90].

In what follows, we will present an answer set programming based approach for solving alternating Boolean equation systems. In this approach a problem is solved by devising a mapping from a problem instance to a logic program so that models of the program provide the answers to the problem instance [66, 73, 78].

In Section 6, we will define such a mapping from alternating Boolean equation systems to logic programs. This provides a basis for a new solution technique for alternating Boolean equation systems.

2.7 Difference Logic

In this thesis, we will present encodings of Boolean equation systems into difference logic [71, 79], i.e. propositional logic combined with the theory of integer differences. Therefore, this subsection provides a brief introduction to difference logic and its satisfiability problem.

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of Boolean variables and $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$ a set of integer variables. The set of atomic formulas of difference logic consists of propositions in $\mathcal{P}$ and integer constraints of the forms $(v_i \geq v_j)$ and $(v_i > v_j)$ with $v_i, v_j \in \mathcal{V}$. The set $\mathcal{F}$ of all difference logic formulas is the smallest set containing the atomic formulas which is closed under negation and conjunction:

- if $\phi \in \mathcal{F}$, then $\neg\phi \in \mathcal{F}$, and
- if $\phi \in \mathcal{F}$ and $\psi \in \mathcal{F}$, then $(\phi \wedge \psi) \in \mathcal{F}$.

The Boolean connectives $\vee, \rightarrow, \leftrightarrow$ are defined in the usual way in terms of $\neg$ and $\wedge$. Our logic is actually a proper subset of the standard definition of difference logic over integers which allows integer constraints of the form $(v_i + k \geq v_j)$, where k is an arbitrary integer constant, see e.g., [79].

A $(\mathcal{P}, \mathcal{V})$ valuation consists of two functions $\beta : \mathcal{P} \rightarrow \{\bot, \top\}$ and $\beta : \mathcal{V} \rightarrow \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers. Notice the name β is overloaded here. The valuation is extended to all formulas in $\mathcal{F}$ by defining $\beta(v_i \geq v_j) = \top$ iff $\beta(v_i) \geq \beta(v_j)$, $\beta(v_i > v_j) = \top$ iff $\beta(v_i) > \beta(v_j)$, and applying the usual rules for the Boolean connectives. A formula $\phi \in \mathcal{F}$ is satisfied by a valuation iff $\beta(\phi) = \top$, and it is satisfiable if there exists a satisfying valuation. Given a formula $\phi \in \mathcal{F}$, the satisfiability problem is to decide whether or not ϕ is satisfiable.

Theorem 27 [71] *The satisfiability problem for difference logic is NP-complete.*

Proof:
NP-hardness follows directly from the fact that our logic subsumes propositional logic and membership in NP from the fact that the full difference logic is in NP, see e.g., [71]. □

The satisfiability problem for propositional logic (SAT) is simply the problem of deciding whether or not a difference logic formula without any integer variables is satisfiable, i.e. SAT is a special case of the difference logic satisfiability problem. SAT is considered an important problem in many disciplines and it was the first problem shown to be NP-complete [24].

There is a special case of SAT that can be solved in linear time in the size of a formula, called HornSAT [29]. We briefly define the HornSAT problem here because there is a close connection between HornSAT and the problem of solving alternation-free Boolean equation systems to be discussed in Section 4. A *positive literal* is a Boolean variable $P_i \in \mathcal{P}$ and a *negative literal* is the negation $\neg P_i$ of a Boolean variable $P_i \in \mathcal{P}$. A *Horn clause* is a disjunction of literals, with at most one positive literal. A *Horn formula* is a conjunction of Horn clauses. HornSAT is the problem of deciding whether or not there is a valuation under which a Horn formula evaluates to $\top$. For instance, Dowling and Gallier [29] give a linear-time algorithm for solving HornSAT.

3 A GENERAL PROCEDURE TO SOLVE BOOLEAN EQUATION SYSTEMS

In this section, we introduce an overall approach to solve a Boolean equation system. We list some important properties of Boolean equation systems which allow for dividing them into blocks. After a brief discussion on partitioning, we give an overview of the most important types of blocks that may result in block partitioning. Then, we present an algorithm required to solve a general Boolean equation system using the approach. This section serves mainly as preliminaries to subsequent sections. Its purpose is to give a general idea of how a Boolean equation system can be solved by first partitioning it into blocks and then solving the individual blocks with specific, customized procedures.

3.1 Partitioning Boolean Equation Systems

The variables of a standard form Boolean equation system can be partitioned in *blocks* such that any two distinct variables belong to the same block iff they are mutually dependent. Consequently, each block consists of such variables whose nodes reside on the same maximal strongly connected component of the corresponding dependency graph.

The dependency relation among variables extends to blocks such that block B_i depends on another block B_j if some variable occurring in block B_i depends on another variable in block B_j. Below we have a simple example of such a partitioning on a Boolean equation system from a previous example.

Example 28 *Consider again the Boolean equation system $\mathcal{E}_1$ of Example 10. This system can be divided into two blocks, $B_1 = \{x_1, x_2\}$ and $B_2 = \{x_3\}$ such that the block B_1 depends on the block B_2. Consequently, the block B_1 is highest up in the block ordering, and block B_2 is the lowest block.*

Finding the blocks of a Boolean equation system can be done in linear time using the dependency graph with any algorithm suitable to detect maximal strongly connected components in directed graphs, for instance, those from [89, 95].

To summarize, a given Boolean equation system can trivially be partitioned into individual blocks via the following steps:

- construct a dependency graph of the Boolean equation system at hand;
- compute all maximal strongly connected components of the dependency graph;
- the set of blocks of the Boolean equation system is simply the set of maximal strongly connected components from the previous step.

Notice that the block ordering can be determined in linear time as well, namely by simply applying standard depth-first search algorithm to find the topological ordering among the blocks. Thus, the algorithm by Tarjan [95] gives the topological ordering for the blocks as a side result when calculating the blocks of a Boolean equation systems.

In general, this kind of partitioning is done as a preprocessing phase in our solution technique. The advantage of our approach is that we can use customized, optimized procedures to solve the individual blocks. In the following sections, we will present various routines and techniques to solve individual blocks in isolation.

Let us have a look at what kinds of blocks may result in the partitioning.

3.2 Types of Blocks of a Boolean Equation System

Recall the definition of a trivial maximal strongly connected component given in Definition 20. A *trivial* block of a Boolean equation system is such a block whose maximal strongly connected component (in the corresponding dependency graph) is trivial. Solutions to variables appearing in trivial blocks are solely determined on the basis of other blocks. Therefore, in what follows we will only be dealing with non-trivial blocks.

There are mainly two classes of non-trivial blocks of a Boolean equation system, namely *alternation-free* and *alternating* blocks. Alternating blocks can further be divided into *disjunctive*, *conjunctive*, and *general* blocks. Let us have a closer look at each of them in turn.

Alternation-Free Blocks

All variables of an alternation-free block have the same sign, either μ or ν. In the former case the block is said to be *minimal* and in the latter case *maximal*.

Alternation-free blocks are especially important because encoding the model checking problem of alternation-free μ-calculus as Boolean equation systems leads to systems with alternation-free blocks only. Therefore, for instance, the model checking problems for Hennessy-Milner logic (HML) [49], Computation Tree Logic (CTL) [17], and many equivalence/preorder checking problems result in alternation-free Boolean equation systems with alternation-free blocks only (see for instance [75]).

In Section 4, we will review solution methods for alternation-free blocks. It will be seen that such blocks can be solved in linear time in the size of the block.

Conjunctive and Disjunctive Blocks with Alternation

Important subclasses of alternating blocks are both conjunctive and disjunctive blocks with alternation. A conjunctive block with alternation consists of such a portion of a Boolean equation system, whose defining equations have different fixpoint signs, but all right-hand side expressions are conjunctive. Similarly, a disjunctive block with alternation consists of such a portion of a Boolean equation system, whose defining equations have different fixpoint signs, but all right-hand side expressions are disjunctive.

Many practically relevant μ-calculus formulas (actually virtually all of them) can be encoded as Boolean equation systems that have only conjunctive or disjunctive blocks with alternation. For instance, encoding the model checking problems for the fragments L1 and L2 of the μ-calculus [8, 34, 35] (and similar subsets) as Boolean equation systems result in alternating systems which are either in conjunctive or disjunctive form. Hence, the problem of solving conjunctive and disjunctive blocks of Boolean equation systems is so important that developing special purpose solution techniques for these classes is worthwhile.

In Section 5, we will study solution methods for conjunctive and disjunctive blocks with alternation. It will be seen that such blocks can be solved in quadratic, and even sub-quadratic, time in the size of the block.

General Alternating Blocks

In a general alternating block of a Boolean equation system, there are variables with both fixpoint signs μ and ν. Moreover, the right-hand side expressions are arbitrary in the sense that both conjunctions and disjunctions may appear as right-hand side formulas. This is the most general form of a Boolean equation system.

From a practical point of view, alternating blocks are rare as they do not occur very frequently in Boolean equation systems arising in the context of verification. Many alternating, general form Boolean equation systems that can be found from the literature are theoretical constructions (see, e.g., [12] for such examples).

But, from a theoretical point of view, solving an alternating, general form Boolean equation system is an interesting challenge. The problem is known to be in the complexity class NP ∩ co-NP [70] (and can be shown to be even in UP ∩ co-UP). Furthermore, it is widely believed that a polynomial time algorithm for the problem may exist but the best known algorithms to date are exponential in the alternation depth of the Boolean equation system.

In Sections 6 and 7, we will propose new approaches to solve alternating blocks of a Boolean equation system which are based on answer set programming and satisfiability solving.

3.3 General Solution Algorithm for Boolean Equation Systems

In Mader [70], there are two useful lemmas which allow to solve all blocks of standard form Boolean equation systems one at a time. As our solution method and proofs are based on these, we restate them here.

Lemma 29 (Lemma 6.2 of [70]**)** *Let $\mathcal{E}$ be a Boolean equation system*

$$(\sigma_1 x_1 = \alpha_1) \dots (\sigma_i x_i = \alpha_i) \dots (\sigma_n x_n = \alpha_n)$$

with equation $(\sigma_i x_i = \alpha_i)$, for $1 \leq i \leq n$. Let α_i' be exactly the same Boolean expression as α_i, except that all occurrences of x_i in α_i are substituted with 1 if $\sigma_i = \nu$, and with 0 if $\sigma_i = \mu$. Then, $\mathcal{E}$ has the same solution as the Boolean equation system

$$(\sigma_1 x_1 = \alpha_1) \dots (\sigma_i x_i = \alpha_i') \dots (\sigma_n x_n = \alpha_n).$$

Lemma 30 (Lemma 6.3 of [70]**)** *Let $\mathcal{E}$ be a Boolean equation system*

$$(\sigma_1 x_1 = \alpha_1) \dots (\sigma_i x_i = \alpha_i) \dots (\sigma_j x_j = \alpha_j) \dots (\sigma_n x_n = \alpha_n)$$

with two distinct equations $(\sigma_i x_i = \alpha_i)$ and $(\sigma_j x_j = \alpha_j)$, for $1 \leq i < j \leq n$. Let α_i' be exactly the same Boolean expression as α_i except that all *occurrences of x_j are substituted with expression α_j. Then, $\mathcal{E}$ has the same solution as the Boolean equation system*

$$(\sigma_1 x_1 = \alpha_1) \dots (\sigma_i x_i = \alpha'_i) \dots (\sigma_j x_j = \alpha_j) \dots (\sigma_n x_n = \alpha_n).$$

The basic idea of our approach is that we can start to find solutions to the variables in the last block, setting them to 1 or 0. Using Lemma 30 we can substitute the solutions for variables in blocks higher up the ordering.

The following simplification rules

- $(\phi \wedge 1) \mapsto \phi$
- $(\phi \wedge 0) \mapsto 0$
- $(\phi \vee 1) \mapsto 1$
- $(\phi \vee 0) \mapsto \phi$

can be used to simplify the equations and the resulting equation system has the same solution. The rules allow to remove each occurrence of 1 and 0 in the right-hand side of equations, except if the right-hand side becomes equal to 1 or 0, in which case yet another equation has been solved. By recursively applying these steps all non-trivial occurrences of 1 and 0 can be removed from the equations and the resulting Boolean equation system is in standard form.

Note that each substitution and simplification step reduces the number of occurrences of variables or the size of a right-hand side, and therefore, only a linear number (in the size of the equation system) of such reductions are applicable.

After solving all variables in a block, and simplifying subsequent blocks a suitable solution routine can be applied to the blocks higher up in the ordering iteratively solving them all. In this way, we can solve all blocks one at a time.

This approach leads to the following strategy to solve a general Boolean equation system $\mathcal{E}$ which is also discussed in [57, 41]. Previously, a quite similar algorithm for equational systems has been given in [22].

Algorithm 1 *The general solution algorithm for Boolean equation systems*

1. *Build the dependency graph $G_{\mathcal{E}}$ of $\mathcal{E}$.*
2. *Divide the system $\mathcal{E}$ into blocks by calculating the maximal strongly connected components of $G_{\mathcal{E}}$.*
3. *Topologically sort $G_{\mathcal{E}}$ into blocks $B_1, B_2, \dots, B_m$; here blocks are numbered so that if a block B_i depends on a block B_j then $i < j$.*
4. *Beginning with B_m, process each block B_i in turn by performing the following steps:*
    (a) *Generate a subsystem $\mathcal{E}'$ of $\mathcal{E}$ containing all equations of $\mathcal{E}$ whose left-hand sides are from B_i. These equations are modified by replacing each occurrence of all variables x_j outside the block B_i by a constant 0 or 1 (according to the already known solution to x_j), and then propagating the constants using the rules to simplify the equations of $\mathcal{E}'$.*

(b) *Solve the variables of the resulting subsystem $\mathcal{E}'$ with a suitable subroutine:*

   i. *if $\mathcal{E}'$ is alternation-free, use algorithms from Section 4;*

   ii. *if $\mathcal{E}'$ is disjunctive or conjunctive, use algorithms from Section 5;*

   iii. *if $\mathcal{E}'$ is general, use algorithms from Sections 6 and 7.*

The correctness of this procedure follows directly from the above lemmas, and from the correctness of the subroutines for various block types.

Theorem 31 *Given a general form Boolean equation system $\mathcal{E}$, the general solution procedure correctly computes the solution to $\mathcal{E}$.*

Proof:
The algorithm computes the solution block-wise. According to Lemma 30 it is safe to substitute already known values to blocks higher up in the ordering, and it is safe to simplify the right-hand side formulas with the simplification rules among a single block. Consequently, the general procedure is correct, assuming that all subroutines to solve the generated subsystems are correct.

□

Notice that all steps $1-3$ and step 4 (a) can be performed in linear time in the size of the underlying Boolean equation system. Thus, the complexity of the general procedure depends naturally on the costs of the subroutines in step 4 (b).

Here, we give a simple example to demonstrate how the above algorithm works on a Boolean equation system from previous examples.

Example 32 *Consider again the Boolean equation system $\mathcal{E}_1$ from Example 10. In step 1, the algorithm builds the dependency graph of the system which is depicted in Figure 2. In step 2, the algorithm divides the system in blocks, as explained in Example 28, resulting in two blocks $B_1 = \{x_1, x_2\}$ and $B_2 = \{x_3\}$ being identified. In step 3, the algorithm topologically sorts these blocks which simply results in the block ordering B_1, B_2. Accordingly, in step 4 the algorithm first solves the block B_2, and then solves the block B_1 in the following way. The block B_2 is alternation-free, and will be solved by using appropriate techniques, in step 4. (b) i. The solution to the only variable x_3 in block B_2 is seen to be 1. Thus, in step 4 (a), to solve block B_1 we generate the subsystem*

$$(\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee 1)$$

and simplify these equations according to the simplification rules. The propagation of the constant 1 in the second equation leads to a more simple system of equations

$$(\nu x_1 = x_1)(\mu x_2 = 1).$$

As this subsystem can be seen to be conjunctive (or disjunctive), the block B_1 can be solved by using appropriate techniques in step 4. (b) ii.

Notice that the above general solution algorithm could be optimized in various ways. For instance in step 4. (b) iii, one could build again the dependency graph after simplification of the equations. This may allow to solve (at least parts of) the general block using more efficient subroutines in steps 4. (b) i and 4. (b) ii. Furthermore, one could try to delay the execution of the most expensive step 4. (b) iii as much as possible, solving the special blocks first. The local solution may be found with this kind of strategy quite early, possibly even before executing step 4. (b) iii at all.

In the following sections, we will present the subroutines and techniques to solve individual blocks in isolation.

4 SOLVING MINIMAL AND MAXIMAL BLOCKS

In this section, we will discuss methods to solve alternation-free blocks of Boolean equation systems. There are two types of blocks that can be alternation-free, namely minimal and maximal. All equations of a minimal block have the fixpoint sign μ, and, dually, all equations of a maximal block have the sign ν. We will begin by exploring well-known linear-time techniques to solve alternation-free Boolean equation systems. Then, we will discuss a logic programming approach to solve alternation-free blocks. In the literature, there are are several efficient methods to solve alternation-free Boolean equation systems. Therefore, we will be brief in this section.

4.1 Algorithms for Alternation-Free Systems

The problem of solving an alternation-free Boolean equation system is a relatively easy task. Indeed, many solution algorithms can be found from the literature which are directed to this class and require only linear time and space in the size of an alternation-free system.

For instance, Andersen [1] presents an efficient linear-time algorithm for finding a global solution to Boolean graphs which correspond to minimal and maximal blocks of a Boolean equation system (see Fig. 1 on p.12 in [1]). In particular, this algorithm is useful to find the global solutions as defined in Definition 11.

In addition, very simple linear-time algorithms to solve a Boolean equation system, whose all equations have the same fixpoint sign, can be found from [68] (see, e.g., Fig. 2 on p. 5). Recall the definition of HornSAT in Section 2.7. In [68], there are also linear-time reductions between alternation-free Boolean equation systems and HornSAT, the problem of Horn formula satisfiability. The techniques described in [68] are local in the sense that they only give the local solutions as defined in Definition 9.

More recently, additional linear-time algorithms for alternation-free Boolean equation systems have been presented in [75]. Very similar algorithms can be found from [76], too. The algorithms from [75, 76] can only be used to find the local solution as defined in Definition 9.

The above algorithms could be applied to solve minimal and maximal blocks in our setting. As they are well-known algorithms, we do not consider them in detail here. Instead, in the following section, we show how minimal and maximal blocks of a Boolean equation system can be solved with an alternative approach based on logic programming techniques.

4.2 Minimal and Maximal Blocks as Logic Programs

A useful way to solve minimal and maximal blocks of a Boolean equation system is through a logic programming approach. Such an approach to solve Boolean equation systems was first proposed in [61]. In brief, it is suggested in [61] that Boolean equation systems can be solved by translating them to propositional normal logic programs, and computing stable models which satisfy certain criteria of preference.

In particular, it is suggested in [61] that alternation-free Boolean equation systems can be mapped to *stratified* logic programs, which can be directly solved in linear time, preserving the linear-time complexity of solving alternation-free Boolean equation systems. Unfortunately, [61] does not provide a complete translation but only sketches an informal idea via a few examples. However, the same kind of idea based on logic programming approach can efficiently be applied to solve minimal and maximal blocks in our setting as well.

Minimal and maximal blocks of Boolean equation systems can be easily seen as equivalent to propositional logic programs where every clause body is a negation-free Boolean formula. Such programs have unique stable models which can be calculated in linear-time (in the size of programs), for instance by employing the algorithm for HornSAT from [29].

Consider a standard form, minimal block of a Boolean equation system. This block itself can be seen as a standard form Boolean equation system, call it $\mathcal{E}$. We construct a logic program $\Pi(\mathcal{E})$ which captures the *global* solution to $\mathcal{E}$. Suppose that all non-trivial occurrences of constants 1 and 0 are removed from the equations of $\mathcal{E}$ by using the simplification rules given in Section 3.3, i.e. there are only equations of the forms $(\mu x_i = 1)$, $(\mu x_i = 0)$, $(\mu x_i = x_j)$, $(\mu x_i = x_j \vee x_k)$ and $(\mu x_i = x_j \wedge x_k)$.

The idea is that $\Pi(\mathcal{E})$ is a propositional normal logic program which has size linear in the size of $\mathcal{E}$ and where every clause body is negation-free. Suppose $\mathcal{E}$ has variables $\{x_1, x_2, \ldots, x_n\}$. The logic program $\Pi(\mathcal{E})$ we derive is over ground atoms $\{p_1, p_2, \ldots, p_n\}$.

For each equation $(\sigma_i x_i = \alpha_i)$ of $\mathcal{E}$, the program $\Pi(\mathcal{E})$ contains the rules:

$$p_i \leftarrow p_j. \qquad \text{if } \alpha_i = x_j \tag{3}$$

$$p_i \leftarrow p_j, p_k. \qquad \text{if } \alpha_i = x_j \wedge x_k \tag{4}$$

$$p_i \leftarrow p_j.\ p_i \leftarrow p_k. \qquad \text{if } \alpha_i = x_j \vee x_k \tag{5}$$

$$p_i. \qquad \text{if } \alpha_i = 1 \tag{6}$$

Notice that there is no rule for equations where the right-hand side formulas are of the form $\alpha_i = 0$ because they do not need to be translated at all.

The intuitive idea of the above translation is that for a variable x_i of $\mathcal{E}$, the solution to x_i is 1 if and only if the unique stable model of $\Pi(\mathcal{E})$ contains the corresponding atom p_i. The correctness of the translation is easy to establish.

Theorem 33 *Let $\mathcal{E}$ be a standard form Boolean equation system where all fixpoint signs are minimal, and let x_i be any variable of $\mathcal{E}$. Then, the solution to x_i is 1 iff $\Pi(\mathcal{E})$ has a stable model which contains the ground atom p_i.*

Proof:
Immediate from Definition 11, Definition 24, and by the construction of $\Pi(\mathcal{E})$. □

By Theorem 33, a minimal block of a Boolean equation system can now be solved by first converting the equation system into a corresponding logic program, then calculating the unique stable model of the program, and finally checking the resulting stable model for the containment of atoms.

Next, we demonstrate the above translation from minimal Boolean equation systems to propositional normal logic programs.

Example 34 *Consider the Boolean equation system $\mathcal{E}$ below:*

$$(\mu x_1 = x_3)(\mu x_2 = 1)(\mu x_3 = x_4 \vee x_5)(\mu x_4 = x_2 \wedge x_1)(\mu x_5 = x_1)(\mu x_6 = x_2).$$

The corresponding program $\Pi(\mathcal{E})$ over ground atoms $\{p_1, p_2, \ldots, p_6\}$ consists of the rules:

$$\begin{array}{l} p_1 \leftarrow p_3. \\ p_2. \\ p_3 \leftarrow p_4. \quad\quad p_3 \leftarrow p_5. \\ p_4 \leftarrow p_2, p_1. \\ p_5 \leftarrow p_1. \\ p_6 \leftarrow p_2. \end{array}$$

The stable model of program $\Pi(\mathcal{E})$ is $\{p_2, p_6\}$. As expected, by Theorem 33, the only variables of $\mathcal{E}$ with solution 1 are x_2 and x_6. For variables x_1, x_3, x_4, and x_5, the solution is 0 because the corresponding atoms p_1, p_3, p_4, p_5 are not contained in the stable model of the program $\Pi(\mathcal{E})$.

By duality, a method for obtaining the global solutions for maximal blocks via stable model computation proceeds in the very same way. For instance, the dual case (i.e. the case for maximal blocks) can be solved by complementing a given system and using the same translation as for minimal blocks. In the following, we will demonstrate how to solve maximal blocks using this approach.

The complementation for Boolean equation systems can be defined as in Definition 35. This definition is an instance of the well-known duality principle for Boolean logic. Therefore, *dual* could be used as a term here instead of *complement*.

Definition 35 (The complementation of a Boolean equation system) *The complement of a Boolean equation system*

$$(\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

is another Boolean equation system

$$(\overline{\sigma_1} x_1 = \overline{\alpha_1})(\overline{\sigma_2} x_2 = \overline{\alpha_2})\ldots(\overline{\sigma_n} x_n = \overline{\alpha_n})$$

where $\overline{\sigma_i}$ is defined by

$$\overline{\sigma_i} = \begin{cases} \nu & \text{if } \sigma_i = \mu \\ \mu & \text{if } \sigma_i = \nu \end{cases}$$

and $\overline{\alpha_i}$ is defined inductively as follows:

$$\begin{array}{l} \overline{0} = 1 \\ \overline{1} = 0 \\ \overline{x_i} = x_i \\ \overline{\alpha_j \wedge \alpha_k} = \overline{\alpha_j} \vee \overline{\alpha_k} \\ \overline{\alpha_j \vee \alpha_k} = \overline{\alpha_j} \wedge \overline{\alpha_k} \end{array}$$

Here, $x_i \in \mathcal{X}$ and $\alpha_j, \alpha_k \in B(\mathcal{X})$.

The complementation of a Boolean equation system preserves the solution in the following sense.

Lemma 36 (Lemma 3.35 of [70]**)** *Let $\mathcal{E}$ be a Boolean equation system and let $\overline{\mathcal{E}}$ be the complement of $\mathcal{E}$. Then, for each variable x_i of $\mathcal{E}$, the solution to x_i in $\mathcal{E}$ is 1 iff the solution to x_i in $\overline{\mathcal{E}}$ is 0.*

The complementation is very useful concept in most of the proofs concerning Boolean equation systems because, as a simple consequence of Lemma 36, many properties of Boolean equation systems have dual properties as well. Therefore, it is usually sufficient to give only one half of a proof of a property, and the other half immediately follows by a symmetric, dual argument.

For instance, the above fact explains why a maximal block of a Boolean equation system can be solved by complementing the block, and then using exactly the same solution method as for minimal blocks. To see this, consider the following example as an application of Lemma 36.

Example 37 *Consider the Boolean equation system $\mathcal{E}$ below, with only maximal equations:*

$$(\nu x_1 = x_2 \wedge x_3)(\nu x_2 = x_3 \vee x_4)(\nu x_3 = x_2 \vee x_4)(\nu x_4 = 0).$$

In order to solve system $\mathcal{E}$, we first take its complement $\overline{\mathcal{E}}$ given below:

$$(\mu x_1 = x_2 \vee x_3)(\mu x_2 = x_3 \wedge x_4)(\mu x_3 = x_2 \wedge x_4)(\mu x_4 = 1).$$

Then, we compute the unique stable model of the logic program $\Pi(\overline{\mathcal{E}})$ over ground atoms $\{p_1, p_2, \ldots, p_4\}$ which consists of the rules:

$$\begin{array}{l} p_1 \leftarrow p_2. \quad p_1 \leftarrow p_3. \\ p_2 \leftarrow p_3, p_4. \\ p_3 \leftarrow p_2, p_4. \\ p_4. \end{array}$$

The only stable model of program $\Pi(\overline{\mathcal{E}})$ is $\{p_4\}$. By Theorem 33, the only variable of $\overline{\mathcal{E}}$ with solution 1 is x_4. The solution is 0 to variables x_1, x_2, x_3 of $\overline{\mathcal{E}}$ because the corresponding atoms p_1, p_2, p_3 are not contained in the stable model of $\Pi(\overline{\mathcal{E}})$. By Lemma 36, the solution is 1 to variables x_1, x_2, x_3 of $\mathcal{E}$, and the solution is 0 to variable x_4 of $\mathcal{E}$.

5 SOLVING CONJUNCTIVE AND DISJUNCTIVE BLOCKS WITH ALTERNATION

In this section, we examine *conjunctive* and *disjunctive* fragments of Boolean equation systems. Many practically relevant properties of systems can be expressed by means of fixpoint formulas that lead to Boolean equations in either conjunctive or disjunctive forms (for instance, see Section 8.1). It is therefore interesting to develop specific solution techniques for disjunctive and conjunctive blocks with alternation. We define two different algorithms specially designed to solve these kinds of blocks. The main results by the thesis author presented in this section are from [41, 42].

We first introduce basic properties concerning conjunctive and disjunctive Boolean equation systems. Then, we present two distinct algorithms for solving conjunctive and disjunctive blocks based on the properties. We also deal with the correctness and complexity of the algorithms.

5.1 Properties of Conjunctive and Disjunctive Blocks

A Boolean equation system is called *disjunctive* if no conjunction symbol appears in its right-hand side expressions. In the same way, a Boolean equation system is called *conjunctive* if no disjunction symbol appears in its right-hand side expressions. Consequently, we define conjunctive and disjunctive Boolean equation systems in the following way.

Definition 38 *Let $(\sigma_i x_i = \alpha_i)$ be an equation of a standard form Boolean equation system. We call this equation disjunctive if no conjunction symbol $\wedge$ appears in α_i. Let $\mathcal{E}$ be a standard form Boolean equation system. We call $\mathcal{E}$ disjunctive iff each equation in $\mathcal{E}$ is disjunctive.*

The dual case below is similar.

Definition 39 *Let $(\sigma_i x_i = \alpha_i)$ be an equation of a standard form Boolean equation system. We call this equation conjunctive if no disjunction symbol $\vee$ appears in α_i. Let $\mathcal{E}$ be a standard form Boolean equation system. We call $\mathcal{E}$ conjunctive iff each equation in $\mathcal{E}$ is conjunctive.*

Clearly, the above definitions can be applied to blocks of a Boolean equation system too, and we will accordingly speak of disjunctive and conjunctive blocks.

We have the following useful lemma. In this lemma, recall that the the definition of the dependency graph is such that the sink nodes with self-loops, $\perp$ and $\top$, represent the constants 0 and 1, and that these sink nodes are assumed to be highest up in the ordering on the nodes.

Lemma 40 *Let $\mathcal{E}$ be a standard form, disjunctive Boolean equation system*

$$(\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2) \ldots (\sigma_n x_n = \alpha_n)$$

and let $G = (V, E, \ell)$ be the dependency graph of $\mathcal{E}$. Let $[\![\mathcal{E}]\!]$ be the local solution to $\mathcal{E}$. Then the following are equivalent:

1. $[\![\mathcal{E}]\!] = 1$

2. *$\exists j \in V$ with $\ell(j) = \nu$ such that:*

   (a) *j is reachable from node 1 in G, and*

   (b) *G contains a cycle of which the lowest index of a node on this cycle is j.*

Proof:
First we show that (2) implies (1).

If j lies on a cycle with all nodes larger than j, then there is a path

$$(j, k_1, k_2, \ldots, k_n, j)$$

in graph G such that, for $1 \leq i \leq n$, $j < k_i$ holds. So there is a sub-equation system of $\mathcal{E}$ that looks as follows:

$$\begin{array}{lcl}(\nu x_j & = & \alpha_j) \\ & \vdots & \\ (\sigma_{k_1} x_{k_1} & = & \alpha_{k_1}) \\ (\sigma_{k_2} x_{k_2} & = & \alpha_{k_2}) \\ & \vdots & \\ (\sigma_{k_n} x_{k_n} & = & \alpha_{k_n})\end{array}$$

Using Lemma 30 we can rewrite the Boolean equation system $\mathcal{E}$ to an equivalent one by replacing the equation $\nu x_j = \alpha_j$ by $\nu x_j = \beta_j$ where β_j is exactly the same Boolean expression as α_j except that, for $1 \leq i \leq n$, all occurrences of x_{k_i} are substituted with expression α_{k_i}. Now note that the right hand side β_j of equation $\nu x_j = \beta_j$ contains only disjunctions and the variable x_j at least once. Hence, by Lemma 29 the equation reduces to $\nu x_j = 1$. As node j is reachable from node 1 in dependency graph G, the equation $\sigma_1 x_1 = \alpha_1$ can similarly be replaced by $\sigma_1 x_1 = 1$. Hence, for the solution $[\![\mathcal{E}]\!]$ of $\mathcal{E}$, it holds that $[\![\mathcal{E}]\!] = 1$.

Now we prove that (1) implies (2) by contraposition. So, assume that there is no node j with $\ell(j) = \nu$ that is reachable from node 1 such that j is on a cycle with only higher numbered nodes.

The proof proceeds by induction on $n-k$ and we show that $\mathcal{E}$ is equivalent to the Boolean equation system where equations

$$(\sigma_{k+1} x_{k+1} = \alpha_{k+1}) \ldots (\sigma_n x_n = \alpha_n)$$

whose nodes $k+1, \ldots, n$ are reachable from 1 have been replaced by

$$(\sigma_{k+1} x_{k+1} = \beta_{k+1}) \ldots (\sigma_n x_n = \beta_n)$$

where all β_l are disjunctions of 0 and variables that stem from $x_1, \ldots, x_k$. If the inductive proof is finished, the lemma is also proven: consider the case where $n - k = n$. This says that $\mathcal{E}$ is equivalent to a Boolean equation system where all right hand sides of equations, on which x_1 depends, are equal to constant 0. So, for the solution $[\![\mathcal{E}]\!]$ of $\mathcal{E}$ it holds that $[\![\mathcal{E}]\!] = 0$.

For $n - k = 0$ the induction hypothesis obviously holds. In particular constant 1 cannot occur in the right hand side of any equation on which x_1 depends. So, consider some $n - k$ for which the induction hypothesis holds.

We show that it also holds for $n - k + 1$. So, we must show that, if equation $\sigma_k x_k = \alpha_k$ is such that x_1 depends on x_k, then it can be replaced by an equation $\sigma_k x_k = \beta_k$ where in β_k only variables chosen from $x_1, \ldots, x_{k-1}$ and constant 0 can occur.

As k is reachable from 1, all variables x_l occurring in α_k are such that x_1 depends on x_l. By the induction hypothesis the equations $\sigma_l x_l = \alpha_l$ for $l > k$ have been replaced by $\sigma_l x_l = \beta_l$ where in β_l only 0 and variables from $x_1, \ldots, x_k$ occur. Using Lemma 30 such variables x_l can be replaced by β_l and hence, α_k is replaced by γ_k in which 0 and variables from $x_1, \ldots, x_k$ can occur.

What remains to be done is to remove x_k from γ_k assuming x_k occurs in γ_k. This can be done as follows. Suppose $\sigma_k = \nu$. Then, as x_k occurs in γ_k, there must be a path in the dependency graph G to a node l' with $l' \geq k$ such that x_k appears in $\alpha_{l'}$. But this means that the dependency graph has a cycle on which k is the lowest value. This contradicts the assumption. So, it cannot be that $\sigma_k = \nu$, and thus $\sigma_k = \mu$. Now using Lemma 29 the variable x_k in α_k can be replaced by 0. □

Also, a dual property holds for conjunctive Boolean equation systems.

Lemma 41 *Let $\mathcal{E}$ be a standard form, conjunctive Boolean equation system*

$$(\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2) \ldots (\sigma_n x_n = \alpha_n)$$

and let $G = (V, E, \ell)$ be the dependency graph of $\mathcal{E}$. Let $[\![\mathcal{E}]\!]$ be the local solution to $\mathcal{E}$. Then the following are equivalent:

1. *$[\![\mathcal{E}]\!] = 0$*
2. *$\exists j \in V$ with $\ell(j) = \mu$ such that:*
   - (a) *j is reachable from node 1 in G, and*
   - (b) *G contains a cycle of which the lowest index of a node on this cycle is j.*

Proof:
In the same way as for Lemma 40. □

One can see that, as a consequence of Lemma 40 and Lemma 41, all variables in disjunctive or conjunctive blocks of a Boolean equation system have the same solutions. We may thus solve all variables of a conjunctive or a disjunctive block by simply computing the solution to the the smallest variable appearing in the block.

Furthermore, since a block of a Boolean equation system consists of a single maximal strongly connected component of the corresponding dependency graph, we may assume that all nodes in the dependency graph of the block are reachable from the smallest node. Therefore, in order to solve a conjunctive or a disjunctive block of a Boolean equation system, the condition that needs to be checked is whether there is a cycle in the dependency graph of the block where the lowest numbered node has label μ (or ν respectively). In the following subsections we define algorithms that perform this task.

Obviously, based on the above observations one can define specialized algorithms that are likely to perform better in practice than the more general algorithms for general form Boolean equation systems.

It is worthwhile to observe that the problem of solving a conjunctive/disjunctive Boolean equation system is equivalent to the problem of solving a one player parity game. Moreover, notice that parity word automata [59] can be shown to be equivalent to both of these formalism.

5.2 Depth-First Search Based Algorithm

There is a very simple algorithm based on depth-first search [96] on directed graphs which can be used to solve conjunctive and disjunctive blocks of a Boolean equation system. The algorithm is discussed in [41] and we present it here in a slightly simplified form. In particular, we give an algorithm to solve a disjunctive block, the conjunctive case is dual and goes along exactly the same lines. Notice that, with an algorithm for the disjunctive case, the conjunctive case can be solved through complementation (see Definition 35 and Lemma 36).

Given a dependency graph $G = (V, E, \ell)$ and a node $i \in V$ with $\ell(i) = \nu$, we define a predicate

$$NuLoop(G, i)$$

to be true iff the subgraph $G \upharpoonright i$ of G contains a cycle $(i, v_0, v_1 ..., i)$ such that $\min\{i, v_0, v_1 ..., i\} = i$. Recall from Definition 18 that $G \upharpoonright i$ is G restricted to nodes $j \geq i$.

Obviously, given a dependency graph $G = (V, E, \ell)$ of a disjunctive block and a node $i \in V$ with $\ell(i) = \nu$, deciding whether $NuLoop(G, i)$ holds reduces to the task of computing the reachability of node i from itself in the subgraph $G \upharpoonright i$ of G. Note that this can be done by a standard depth-first search algorithm in time and space $O(|V| + |E|)$. Assuming such a subroutine to decide $NuLoop(G, i)$, we can resolve a disjunctive block of a Boolean equation systems as follows.

We define the algorithm *SolveDisjunctive*(G) where $G = (V, E, \ell)$ is a dependency graph of a disjunctive block of a Boolean equation system. The algorithm *SolveDisjunctive* calculates whether there is a node k in G such that $\ell(k) = \nu$ and k is the smallest node on some cycle of G. The algorithm consists of the following steps:

Algorithm 2 *The algorithm to solve disjunctive form Boolean equations systems*

1. *For all nodes $i \in V$ such that $\ell(i) = \nu$:*
    - *If $NuLoop(G, i)$ holds, then report "solution to the smallest variable is 1" and STOP.*
2. *Report "solution to the smallest variable is 0".*

It is not difficult to see that the algorithm is correct.

Theorem 42 (Correctness) *The algorithm SolveDisjunctive works correctly on any disjunctive block of a Boolean equation system.*

Proof:
If the algorithm reports that "solution to smallest variable is 1" then

$$NuLoop(G, i)$$

holds for some $i \in V$, and G contains at least one cycle of which the lowest index of a node on this cycle is i, where $\ell(i) = \nu$. By Lemma 40, the solution to smallest variable is 1. If the algorithm reports "solution to smallest variable is 0", then there does not exist a node $i \in V$ with $\ell(i) = \nu$ such that $NuLoop(G, i)$ holds. By Lemma 40, the solution to smallest variable is 0. □

This approach is well suited for many Boolean equation systems. Since the algorithm performs standard depth-first search as a subroutine, which can detect cycles even before the whole graph has been traversed, the algorithm may find the solution by searching only a small portion of the dependency graph. In many cases, this leads to an early detection of the solution.

A disadvantage of the above approach is that, in the worst case, it requires quadratic time in the size of an input dependency graph. For instance, using an adjacency-list representation of dependency graphs, the time complexity of this algorithm is not worse than quadratic.

Theorem 43 *Let $G = (V, E, \ell)$ be a dependency graph of a disjunctive block with $|V| = n$ and $|E| = m$. The algorithm SolveDisjunctive requires time $O(n \cdot (n + m))$ to solve G.*

Proof:
The algorithm calls function $NuLoop$ at most n times and each call takes time $O(n + m)$. □

Note that the space complexity of *SolveDisjunctive*(G) is linear in the size of the input dependency graph, i.e. $O(|G|)$.

Next, we give an example which demonstrates that the above algorithm may take quadratic time. (Here, we use the notation $\Omega(f(n))$ for asymptotic lower bound; verbally read as "of the order *at least* $f(n)$".)

Theorem 44 *For all $n \in \mathbb{N}$ such that $n \geq 4$, there is a disjunctive Boolean equation system with dependency graph of size $O(n)$, on which the solution algorithm SolveDisjunctive takes time $\Omega(n^2)$.*

Proof:
We define a family of Boolean equation systems $\mathcal{E}_n$, for all even $n \in \mathbb{N}$ s.t. $n \geq 4$. For some even $n \in \mathbb{N}$ s.t. $n \geq 4$, consider the Boolean equation system:

$$(\mu x_1 = x_2)$$

$$(\nu x_2 = x_1 \vee x_3)$$

$$(\mu x_3 = x_1 \vee x_4)$$

$$\vdots$$

$$(\mu x_{n-1} = x_1 \vee x_{n-1})$$

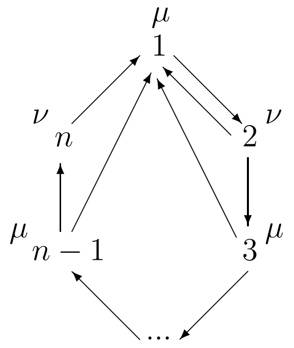


Figure 5: A worst-case example for the algorithm based on depth-first search.

$$(\nu x_n = x_1)$$

The above equation system is disjunctive, and the solution to variable x_1 is 0. Consider the dependency graph G of this system depicted in Figure 5. Clearly, in order to solve the block with the *SolveDisjunctive* algorithm, we need always at least

$$|G{\restriction}2| + |G{\restriction}4| + \cdots + |G{\restriction}n|$$

steps, i.e. $\Omega(n^2)$ time is needed to solve the Boolean equation system. □

Most algorithms for solving conjunctive and disjunctive Boolean equation systems, including those from [41, 70], take at least quadratic time in the size of a Boolean equation system in the worst case. But, for large Boolean equations, which are typically encountered in model checking of realistic systems, these algorithms might lead to unpleasant running times.

Therefore, the next subsection presents an especially fast algorithm for finding a solution to a Boolean equation system in either conjunctive or disjunctive form.

5.3 An Algorithm Based on Hierarchical Clustering

The contribution of this subsection is to present an especially fast algorithm for finding a solution to a Boolean equation system in either conjunctive or disjunctive form. Given such a system with size e and the alternation depth $d > 1$, the algorithm finds the solution using time $O(e \log d)$ in the worst case (for $d = 1$, the algorithm takes time $O(e)$ in the worst case). This improves the quadratic upper bound and should make the verification of a large class of fixpoint expressions more tractable.

Essentially, the algorithm is a new variant of King, Kupferman and Vardi [59] who give a similar algorithm in the realm of parity word automata. As both algorithms are based on ideas in [96] we therefore attribute explicitly the original idea to Tarjan by talking about hierarchical clustering.

Tarjan [96] presents a hierarchical clustering algorithm for constructing a strong component decomposition tree for a directed weighted graph. Tarjan's clustering algorithm is an off-line, partially dynamic algorithm which is heavily based on three well-known techniques: binary search, divide-and-conquer, and graph theoretic technique for finding strongly connected components [95].

It turns out that the ideas behind the hierarchical clustering algorithm are also suited to solve conjunctive and disjunctive Boolean equation systems. In the following subsection, we provide such an algorithm to solve disjunctive Boolean equation systems. The conjunctive case is dual and can be solved via complementation too (recall Definition 35 and Lemma 36).

Essentially, our algorithm and the one in [59] are both distinguished from the original hierarchical clustering algorithm [96] in the following way. The algorithm in [96] takes a weighted directed graph as its input and performs computations with this weighted graph; in contrast, both [59] and our variant make computations with directed labelled graphs. In addition, Tarjan's original algorithm in [96] always requires the asymptotically worst case time but this is not the case for [59] and our variant.

Our algorithm is distinguished from the existing algorithm [59] as follows:

- The algorithm in [59] may continue its computation even when the solution is already known. Our algorithm instead terminates as soon as the solution has been found, and thus avoids unnecessary computations. This is due to a minor fault in the procedure solve in Section 3 of [59]. More precisely, for the correctness of the procedure solve in [59] it is crucial that STOP (lines 1 and 5 in [59]) is interpreted as stop the current recursive call, i.e. do not stop the whole execution of the algorithm but this particular call to solve. But, this means that, if the solution is found in line 5 of [59] and YES is reported, then the recursion stack may still contain several calls that will be executed, and the algorithm [59] incorrectly continues its computation although the solution has been already found.

- Neither implementation-level description nor implementation of the algorithm is provided in [59]. In contrast, we have implemented our algorithm and we provide extensive experimental results which demonstrate the actual performance of the algorithm (see Section 8.1).

- In the algorithm [59], the order of the recursive calls is fixed but we noticed that the recursion order is not relevant for the correctness of the algorithms. Therefore, we have studied various recursion orders in order to obtain optimized versions of the algorithm (again, see Section 8.1).

Basic Definitions

Before presenting the algorithm, we define a few useful notions that will be needed.

Definition 45 *Let*

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

*be a Boolean equation system. An index j is a ν-*starting point *of $\mathcal{E}$ if $\sigma_j = \nu$, and either $j = 1$ or $\sigma_{j-1} = \mu$. If j is a ν-starting point then the ν-*segment of j *are those indices* $j, j+1, \ldots, j+k$ *such that* $\sigma_{j+i} = \nu$ $(0 \leq i \leq k)$ *and either* $j+k = n$ *or* $\sigma_{j+k+1} = \mu$.

In other words, all ν-variables that have the same $\mathcal{H}$ label in the alternation hierarchy of a Boolean equation system belong to the same ν-segment. Note that the alternation depth of a Boolean equation system is twice the number of ν-starting points of a Boolean equation system minus 0, 1 or 2, depending on whether or not there are initial and trailing μ's.

Recall Definition 18 where we define dependency graphs restricted to only certain subgraphs. In the same way, given a dependency graph $G = (V, E, \ell)$ and $k \in V$, let the restricted graph $G{\restriction}k = (V, E{\restriction}k, \ell)$ where

- $E{\restriction}k = \{\langle i, j\rangle \in E \mid i \geq k \text{ and } j \geq k\}$.

In general, we may assume that all nodes in a dependency graph $G = (V, E, \ell)$ of a disjunctive Boolean equation system are reachable from node 1 because the nodes that are not reachable from node 1 do not affect the solution.

Furthermore, we may assume that the dependency graph of a disjunctive Boolean equation system does not contain any self-loops (i.e. an edge from a node to itself) because such edges can trivially be removed from dependency graphs preserving the solution. To see this notice that, for all edges $(i, i) \in E$ s.t. $\sigma_i = \mu$, we can simply remove the edge (i, i) from E and the solution is preserved. On the other hand, if an edge $(i, i) \in E$ with $\sigma_i = \nu$ exists, then it holds that $[\![\mathcal{E}]\!] = 1$ and $\mathcal{E}$ is already solved.

Thus, by Lemma 40, the essential condition that needs to be checked is whether there is a cycle in the dependency graph of which the lowest numbered node has label ν. The following algorithm performs this task in subquadratic time.

The MinNuLoop Algorithm
To apply the algorithm on a Boolean equation system $\mathcal{E}$ with n equations, *MinNuLoop*(k, n, G) must be executed where G is the dependency graph of $\mathcal{E}$ and k is the first ν-starting point of $\mathcal{E}$. If such a starting point does not exist, then it holds that $[\![\mathcal{E}]\!] = 0$ and $\mathcal{E}$ is already solved.

Algorithm. We define the algorithm *MinNuLoop*(k_1, k_2, G) where k_1 and k_2 are indices such that $k_1 \leq k_2$, $G = (V, E, \ell)$ is a dependency graph, $\ell(k_1) = \nu$ and $|E| \geq |V|$. The algorithm *MinNuLoop* calculates whether there is an index k with $k_1 \leq k \leq k_2$, $\ell(k) = \nu$ and k is the smallest node on some cycle of G. The algorithm consists of the following steps:

1 Let s be the number of ν-starting points on $k_1, \ldots, k_2$. Let k_3 be the index of the $\lceil \frac{1}{2}s \rceil$-th ν-starting point on $k_1, \ldots, k_2$. Calculate the maximal strongly connected components of $G{\restriction}k_3$. Check whether any node on the ν-segment of k_3 resides in a non-trivial strongly connected component. If so, report "Found" and stop the execution of the algorithm. In the following steps, let $C(k)$ represent the strongly connected component of $G{\restriction}k_3$ containing node $k \in V$, and let $\min C(k)$ be the smallest member of the set $C(k)$.

2 Here and in 5 below we check nodes in the range $k_1, \ldots, k_3 - 2$. Calculate the graph $G' = (V', E', \ell')$ by

$$\begin{aligned}
&V' = \{\min C(i) \mid i \in V \text{ and } \exists j. \langle i, j\rangle \in E \text{ and } C(i) \neq C(j)\},\\
&E' = \{\langle \min C(i), \min C(j)\rangle \in V' \times V' \mid \langle i, j\rangle \in E,\ C(i) \neq C(j)\},\\
&\ell'(\min C(i)) = \begin{cases} \ell(i) & \text{if } C(i) \text{ is trivial,}\\ \mu & \text{otherwise.}\end{cases}
\end{aligned}$$

3 Let k_4 be the smallest index of a ν-starting point larger than k_3. We check nodes in the range $k_4, \ldots, k_2$ (see also item 6). Calculate the graph $G'' = (V'', E'', \ell)$ by

$$\begin{aligned}
&V'' = \{i \in V \mid C(i) \text{ is not trivial}\} \text{ and}\\
&E'' = \{\langle i, j\rangle \in V'' \times V'' \mid \langle i, j\rangle \in E \text{ and } C(i) = C(j)\}.
\end{aligned}$$

4 Forget G.

5 If $k_1 \leq k_3 - 2$, execute *MinNuLoop*$(k_1, k_3 - 2, G')$.

6 If $k_4 \leq k_2$, execute *MinNuLoop*(k_4, k_2, G'').

Notice that the algorithm *MinNuLoop* splits up the input dependency graph G into two graphs G' and G'', and then recurs on these new graphs. More precisely, in steps 2 and 5, the graph G' is a condensed version of the graph G, in which nodes belonging to the same strongly connected component of $G\restriction k_3$ (calculated in step 1) are compressed into a single node. In steps 3 and 6, the graph G'' is a subgraph of G, in which edges connecting different strongly connected components of $G\restriction k_3$ (calculated in step 1) are removed. Intuitively, the correctness of the algorithm can be explained in the following way.

On the one hand, when investigating whether some of the nodes in the range $k_1, \ldots, k_3 - 2$ is the smallest ν-labelled node on a cycle, the internal structures of non-trivial strongly connected components of $G\restriction k_3$ are irrelevant, and can therefore be safely collapsed in the construction of $G' = (V', E', \ell')$ in step 2. In other words, it suffices that for each strongly connected component calculated in step 1 we include a single representative node in G'. In addition, it suffices to consider as edges of G' exactly the representatives for those edges of E that bridge the strongly connected components of $G\restriction k_3$. Furthermore, nodes in V' without outgoing edges cannot contribute to any cycles, and can therefore be removed. Note that as all nodes without outgoing edges are removed from G', the precondition that $|E'| \geq |V'|$ to invoke *MinNuLoop* is met.

On the other hand, when investigating whether some node in the range $k_4, \ldots, k_2$ is the smallest ν-labelled node on a cycle, we do not need to consider edges that connect nodes belonging to different strongly connected components of $G\restriction k_3$. Indeed, such edges cannot participate in any cycle whose smallest index is in the range $k_4, \ldots, k_2$.

We now turn to give the correctness proof and the complexity analysis of the algorithm; a detailed example demonstrating how the algorithm works will be given later on.

Correctness and Complexity of MinNuLoop
Since the algorithm *MinNuLoop* is closely related to Tarjan's clustering algorithm, its correctness and complexity can be seen along the lines set out in [96, 59]. However, as mentioned in the beginning of Subsection 5.3, our presentation significantly differs from [96] and [59]. Therefore, we give here the correctness and complexity arguments in detail.

Theorem 46 *The MinNuLoop*(k_1, k_2, G) *stops reporting "Found" iff a cycle with a minimal* ν*-labelled node in the range* $k_1, \ldots, k_2$ *exists in* G.

Proof:
As the *MinNuLoop*(k_1, k_2, G) is a divide and conquer algorithm, we prove it correct by first showing that the cases where there are 0 ν-segments are correctly solved. Then, we notice that the *MinNuLoop*(k_1, k_2, G) divides a problem with A ν-segments into three parts of (i) 1 ν-segment, (ii) $\lfloor \frac{1}{2}(A - 1) \rfloor$ ν-segments, and (iii) $\lceil \frac{1}{2}(A - 1) \rceil$ ν-segments. By repeatedly doing this recursively, any initial number of ν-segments will eventually be split into only cases concerning 0 ν-segments. Therefore, if all the cases (i)-(iii) are correctly handled, then the correctness of the whole algorithm follows.

First, assume the number of ν-segments in the range $k_1, \ldots, k_2$ is 0. In this case, clearly no cycle with a minimal ν-labelled node in the range $k_1, \ldots, k_2$ exists in G. In step 1, the algorithm calculates that the number of ν-starting points in the range $k_1, \ldots, k_2$ is 0, and the graph $G \upharpoonright k_3$ will not be constructed at all. In steps 2-6, the algorithm does not execute any additional recursive calls. Thus, nothing is correctly reported in steps 1-6.

Let us next consider all the above three cases (i)-(iii) in turn.

The case (i) is treated in step 1 of the *MinNuLoop*(k_1, k_2, G) where it is checked whether any node in the ν-segment of k_3 is the smallest ν-labelled node on a cycle. Clearly, in step 1 the algorithm reports correctly "Found" and stops if and only if some node on the ν-segment of k_3 resides in a non-trivial maximal strongly connected component of $G \upharpoonright k_3$.

The case (ii) is treated in steps 2 and 5 where it is investigated whether some of the nodes in the range $k_1, \ldots, k_3 - 2$ (node $k_3 - 1$ is μ-labelled) is the smallest ν-labelled node on a cycle. Here, the internal structures of non-trivial strongly connected components of $G \upharpoonright k_3$ are irrelevant, and can therefore be safely collapsed. Thus, it suffices that all strongly connected components calculated in step 1 occur as compressed nodes of G'. In addition, we can take as edges of G' exactly those edges of E that bridge the strongly connected components of $G \upharpoonright k_3$. Furthermore, nodes in V' without outgoing edges cannot contribute to cycles and can therefore be removed. Thus, it is seen that G contains a cycle with a minimal ν-labelled node appearing in the range $k_1, \ldots, k_3 - 2$ if and only if G' contains a cycle with a minimal ν-labelled node appearing in the range $k_1, \ldots, k_3 - 2$. Therefore, the case (ii) is correctly handled.

Finally, the case (iii) is treated in steps 3 and 6 where it is investigated whether some node in the range $k_4, \ldots, k_2$ is the smallest ν-labelled node on a cycle. Clearly, here we do not need to consider edges that connect nodes belonging to different strongly connected components of $G \upharpoonright k_3$ because such edges cannot participate in any cycle whose smallest index is in the range $k_4, \ldots, k_2$. Thus, it is seen that G contains a cycle with a minimal ν-labelled node appearing in the range $k_4, \ldots, k_2$ iff G'' calculated in step 3 contains exactly the same cycle. Therefore, the case (iii) is correctly handled. □

We now estimate the worst-case time complexity of the *MinNuLoop* algorithm.

Theorem 47 *The time complexity of the algorithm MinNuLoop(k_1, k_2, G) is $O(|E| \log A)$ where $G = (V, E, \ell)$ and A is the number of ν-starting points on $k_1, \ldots, k_2$ ($2 \cdot A$ is approximately the alternation depth of the Boolean equation system that corresponds to G).*

Proof:
Let us analyse the asymptotic running time of each step in turn.

Notice that the precondition to invoke *MinNuLoop* is $|V| \leq |E|$. Then, in step 1 of *MinNuLoop* it takes $O(|E|)$ to determine k_3. Furthermore, calculating $G\lceil k_3$, the strongly connected components, and checking whether any node on the ν-segment of k_3 resides on a non-trivial strongly connected component requires time $O(|E|)$. In step 2, the graph G' is constructed. This can clearly be done in time $O(|E|)$. In step 3, it takes $O(|E|)$ time to determine k_4 and to construct the graph G''. In step 4, the removal of G takes time $O(|E|)$. Step 5 requires time $O(1)$, plus possibly the time for the recursive call *MinNuLoop*$(k_1, k_3 - 2, G')$ to solve the subproblem defined by the range $k_1, \ldots, k_3 - 2$ and the graph G' constructed in step 2. Step 6 takes time $O(1)$, plus possibly the time for the recursive call *MinNuLoop*(k_4, k_2, G'') to solve the subproblem defined by the range $k_4, \ldots, k_2$ and the intermediate graph G'' constructed in step 3.

Thus, summarizing the above analysis, computing the steps 1-6 requires time $O(|E|)$ plus the time to solve at most two recursive calls to solve the subproblems.

To estimate the total cost of the *MinNuLoop* algorithm, a crucial observation is that for each edge $(i, j) \in E$ at most one edge shows up in either E' or E'', depending on whether or not $C(i) = C(j)$ holds. This means that $|E'| + |E''| \leq |E|$. Furthermore, if the number of ν-starting points in $k_1, \ldots, k_2$ is A, then there are $\lfloor \frac{1}{2}(A-1) \rfloor$ ν-starting points on $k_1, \ldots, k_3 - 1$ and $\lceil \frac{1}{2}(A-1) \rceil$ ν-starting points on $k_4, \ldots, k_2$.

Let us define a notion of recursion depth for the *MinNuLoop* algorithm in the following way. For the initial call of the *MinNuLoop* with the original problem, the recursion depth is 0. For each call of the *MinNuLoop*, the recursion depth for the next recursive calls of *MinNuLoop* done in steps 5-6 to solve the subproblems is the recursion depth of the current call plus 1. Thus, if the number of ν-starting points on $k_1, \ldots, k_2$ in the original problem is A, then clearly the maximal recursion depth is $O(\log A)$. Obviously, for all $i \in \mathbb{N}$, the time required to calculate the steps 1-6 of all the recursive calls with the recursion depth i is $O(|E|)$ (each edge of the original graph is represented in at most one subproblem at the same recursion level). As the maximum recursion depth is $O(\log A)$ and it takes time $O(|E|)$ to solve all the subproblems at the same recursion level, the total complexity of the *MinNuLoop* is

$$O(\log A) \cdot O(|E|) = O(|E| \log A).$$

□

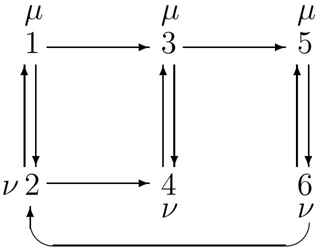


Figure 6: The dependency graph of the example Boolean equation system.

The time complexity for solving a Boolean equation system also contains the generation of the dependency graph, and it is easily seen to be $O(e)$ with e the size of the Boolean equation system.

The space complexity of *MinNuLoop*(k_1, k_2, G) is $O(|E|)$. In order to see this suffices to note that the graphs constructed in step 2 and 3 are together smaller (or of equal size) than the graph G, which is thrown away in step 4. Therefore, the memory footage is only reduced while executing the *MinNuLoop* algorithm. As generating the dependency graph also takes linear space, solving a disjunctive Boolean equation system also takes linear space.

A Detailed Example

We now demonstrate how the *MinNuLoop* algorithm works by giving a detailed example execution. Consider a Boolean equation system

$$(\mu x_1 = x_2 \vee x_3)$$
$$(\nu x_2 = x_1 \vee x_4)$$
$$(\mu x_3 = x_4 \vee x_5)$$
$$(\nu x_4 = x_3)$$
$$(\mu x_5 = x_6)$$
$$(\nu x_6 = x_5 \vee x_2)$$

which is disjunctive and consists of a single alternating block. The dependency graph G of this Boolean equation system is depicted in Figure 6.

In order to solve this block we must call *MinNuLoop*$(2, 6, G)$, because 2 is the smallest ν-starting point on $1, \ldots, 6$.

First, in step 1 it is determined that there are three ν-starting points on $2, \ldots, 6$, namely 2, 4 and 6. Thus, $s = 3$ and the $\lceil \frac{1}{2} \times 3 \rceil$-th ν-starting point on $2, \ldots, 6$ is 4. Therefore, we calculate the strongly connected components of the restricted graph $G\lceil 4$ shown in Figure 7. One can see that the trivial

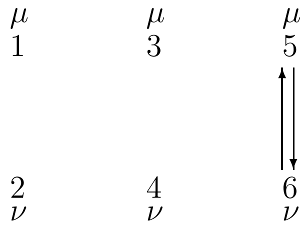


Figure 7: The restricted graph $G{\upharpoonright}4$ calculated in step 1.

strongly connected components of the graph $G{\upharpoonright}4$ are $\{1\}$, $\{2\}$, $\{3\}$ and $\{4\}$. In addition to this, there is one non-trivial strongly connected component in the graph $G{\upharpoonright}4$, namely $\{5, 6\}$.

Secondly, in step 1 the algorithm detects that no node on the ν-segment of 4 resides in a non-trivial strongly connected component of the graph $G{\upharpoonright}4$. Therefore, the algorithm proceeds to step 2.

In step 2, the algorithm builds a new graph G' from G. Now, the nodes of G' are the smallest nodes of each strongly connected component of $G{\upharpoonright}4$, and the edges of G' represent those edges of G that bridge the strongly connected components of $G{\upharpoonright}4$. The resulting graph G' is depicted in Figure 8.

In step 3, the algorithm detects that the smallest index of a ν-starting point

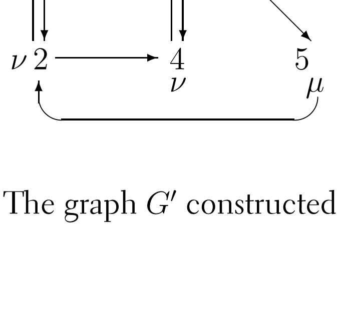


Figure 8: The graph G' constructed in step 2.

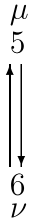


Figure 9: The graph G'' constructed in step 3.

larger than 4 is 6. Furtermore, in step 3 the algorithm builds another graph G'' from G. Recall that, intuitively, the nodes of G'' are such nodes of G which reside on the non-trivial strongly connected components of the restricted graph $G{\restriction}4$. The edges of G'' are such edges of G whose nodes belong to the same strongly connected component of the graph $G{\restriction}4$. Consequently, the resulting graph G'' can be depicted as shown in Figure 9.

Then, in step 4 we simply forget the original graph G which appeared as a parameter in the first call of the function *MinNuLoop*.

In step 5, we check that the condition $k_1 \leq k_3 - 2$ holds, as $2 \leq 2$, and the algorithm recurs by calling $\mathit{MinNuLoop}(2, 2, G')$.

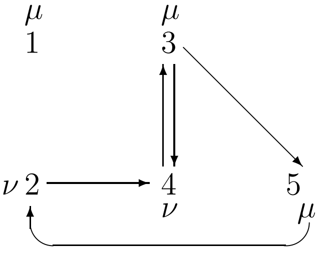


Figure 10: The restricted graph $G'{\restriction}2$ calculated in step 1 deeper in the recursion depth.

In step 6, we check that the nodes in the range $6, \ldots, 6$ by calling recursively $\mathit{MinNuLoop}(6, 6, G'')$, because the condition $k_4 \leq k_2$ holds as $6 \leq 6$.

The recursive call of the function *MinNuLoop* done in step 5 above, i.e. $\mathit{MinNuLoop}(2, 2, G')$, can be explained as follows. Again, in step 1 the algorithm determines the number of ν-starting points. There is only one ν-starting point on $2, \ldots, 2$, namely 2 itself. So, since the $\lceil \frac{1}{2} \times 1 \rceil$-th ν-starting point on $2, \ldots, 2$ is 2, the algorithm calculates the strongly connected components of the restricted graph $G' \upharpoonright 2$. This restricted graph $G' \upharpoonright 2$ is depicted in Figure 10.

One can see that the only trivial strongly connected component of the above graph $G' \upharpoonright 2$ is $\{1\}$. In addition, there is one non-trivial strongly connected component in the graph $G' \upharpoonright 2$, namely $\{2, 3, 4, 5\}$.

Finally, in step 1 the algorithm detects that there is a node on the ν-segment of 2 that resides in a non-trivial strongly connected component of the graph $G' \upharpoonright 2$, namely node 2 in component $\{2, 3, 4, 5\}$. For this reason, the algorithm reports "Found" and stops the execution.

The result is exactly what one might expect; the solution to the smallest variable x_1 of the given Boolean equation system is 1.

5.4 Discussion

In this section, we have seen two distinct ways to solve conjunctive and disjunctive Boolean equation systems. Both of these two algorithms take into account the specific structures of conjunctive and disjunctive Boolean equation systems to solve them efficiently. It is expected that these kinds of specific algorithms give better performance than general algorithms which are designed to solve general form Boolean equation systems lacking the same specific structure.

However, it is worthwhile to notice that the results presented in [2] combined with [4] allow for solving conjunctive/disjunctive Boolean equation systems in quadratic time as well. For example, one may first transform a conjunctive/disjunctive system of size e to a system of size e^2 but having alternation depth two only (notice that this is possible via a result in [4] for mapping a parity word automaton to a quadratic size Büchi automaton). Finally, apply the linear time algorithm of [2] for the alternation depth two Boolean equation systems.

It was seen that the algorithm based on hierarchical clustering has better asymptotic running time than the depth-first search based algorithm. But, an important remaining question is which of these two presented algorithms performs better in practice. Theoretical worst-case analysis of algorithms does not necessarily give enough information when comparing the performance of algorithms in real-world applications. This is due to the fact that worst-case analysis might give too pessimistic bounds for the running times of an algorithm.

Therefore, the analytical results in the previous subsections do not give us enough information about how the depth-first search and hierarchical clustering algorithms perform in practice, on practical verification problems. Thus, some further work needs to be done in order to judge the overall winner.

Experimental research may give much valuable information on the actual performance of algorithms. Therefore, we have implemented both algorithms for evaluation and comparison. The results of our experimental researh will be presented later in Section 8.

6 SOLVING GENERAL BLOCKS BY A REDUCTION TO LOGIC PROGRAMS

In this section, we discuss an answer set programming (ASP) based approach for solving general blocks of Boolean equation systems. In ASP a problem is solved by devising a mapping from a problem instance to a logic program such that models of the program provide the answers to the problem instance [66, 73, 78]. The main results presented in this section are originally published in [57].

We first state some facts about general form Boolean equation systems which turn out to be useful in the computation of their solutions. We then develop a mapping from alternating blocks to logic programs providing a basis for effectively solving such hard blocks. Finally, we show the correctness of our translation.

6.1 Solving General Form Blocks in Answer Set Programming

It is seen in Section 4 that, if all variables in a single block have the same sign (i.e. the block is alternation free), the variables in this block can be trivially solved in linear time. Furthermore, in Section 5 it is seen how the variables appearing in conjunctive and disjunctive blocks with alternation can be solved using only sub-quadratic time. The remaining task is to solve alternating blocks containing both mutually dependent variables with different signs and arbitrary connectives in right-hand side formulas.

As mentioned before, the complexity of solving such general blocks is an important open problem; no polynomial time algorithm has been discovered so far. On the other hand, as mentioned before, the problem is in the complexity class NP ∩ co-NP [70] and can be shown to be even in UP ∩ co-UP.

Here, we present a technique to solve an alternating Boolean equation system which is based on answer set programming. We reduce the problem of solving alternating Boolean equation systems to computing stable models of normal logic programs. This is achieved by devising a mapping from Boolean equation systems to normal logic programs so the solution to a given variable in an equation system can be determined by the existence of a stable model of the corresponding logic program.

This kind of approach is motivated by the success that various logic programming systems have had in solving large verification problems. For instance, [26, 85, 86] demonstrate that logic programming can be applied to the construction of model checkers for the alternation free μ-calculus. More recently, [48] show that answer set programming can succesfully be applied to construct a practically efficient symbolic model checker for linear temporal logic (LTL).

Recall from Section 2.6 that the problem of determining the existence of a stable model of a normal logic program is NP-complete [72]. However, there are answer set programming systems that are quite efficient in practice for many large instances of the problem, for example `Smodels` system [90]. Such solvers have been used in various computer aided verification tasks, for example in the symbolic model checking for LTL [48].

Therefore, the hypothesis is that we can employ the answer set programming techniques to improve the methods to solve general form Boolean equation systems, thus leading to practically efficient model checking techniques for the μ-calculus.

Before giving the translation we discuss some useful properties of general form Boolean equation systems on which our translation is based.

6.2 Properties of General Boolean Equation System

The following observation is the basis for our answer set programming based technique to solve general form Boolean equation systems.

We define a subsystem of a Boolean equation system as follows.

Definition 48 *Let $\mathcal{E}$ be a standard form Boolean equation system. The Boolean equation system $\mathcal{E}'$ is a conjunctive subsystem of $\mathcal{E}$, if $\mathcal{E}'$ is obtained from $\mathcal{E}$ by removing exactly one disjunct in every disjunctive equation of $\mathcal{E}$; otherwise the system $\mathcal{E}$ is unchanged.*

For standard form Boolean equation systems with both disjunctive and conjunctive equations we have the following lemma.

Lemma 49 *Let $\mathcal{E}$ be a standard form Boolean equation system. Then the following are equivalent:*

1. $[\![\mathcal{E}]\!] = 1$

2. *There is a conjunctive subsystem $\mathcal{E}'$ of $\mathcal{E}$ with the solution $[\![\mathcal{E}']\!] = 1$.*

Proof:
Here, we may suppose that all non-trivial occurrences of constants 1 and 0 are first removed from the equations of $\mathcal{E}$ by using the simplification rules given in Section 3.3, i.e. there are only equations of the forms $(\sigma_i x_i = 1)$, $(\sigma_i x_i = 0)$, $(\sigma_i x_i = x_j)$, $(\sigma_i x_i = x_j \vee x_k)$ and $(\sigma_i x_i = x_j \wedge x_k)$.

The fact that (1) implies (2) is already shown in Proposition 3.36 of [70] but we give here an alternative proof. Suppose $[\![\mathcal{E}]\!] = 1$ holds. Construct a parity game $G = (V, E, v_0, \Omega)$ corresponding to $\mathcal{E}$ by using the translation from Boolean equation systems to parity games given in Section 2.5. As $[\![\mathcal{E}]\!] = 1$ holds, by Theorem 23, $\exists$ has a winning strategy $\sigma_\exists$ in G. Let $G|_\sigma$ be the parity game that is induced by the winning strategy $\sigma_\exists$ on G.

Now, let $\mathcal{E}'$ be the conjunctive subsystem contructed as follows: for each disjunctive equation $(\sigma_i x_i = x_j \vee x_k)$ of $\mathcal{E}$, keep the variable corresponding to the node $\sigma_\exists(v_i)$ and remove the other variable from the right-hand side expression. Notice that, as every play π in $G|_\sigma$ has winner $\exists$, $\min\{\Omega(v) \mid v \in \inf \pi\}$ must be even for all plays π of $G|_\sigma$. Then, it is easy to verify that the dependency graph of $\mathcal{E}'$ (having the structure similar to graph $G|_\sigma$) cannot satisfy the conditions 2 (a)-(b) of Lemma 41. By Lemma 41, the solution to $\mathcal{E}'$ is $[\![\mathcal{E}']\!] = 1$.

To see that (2) implies (1), suppose there is a conjunctive subsystem $\mathcal{E}'$ of $\mathcal{E}$ such that $[\![\mathcal{E}']\!] = 1$. We show that $[\![\mathcal{E}]\!] = 1$ holds.

Construct a parity game $G = (V, E, v_0, \Omega)$ corresponding to $\mathcal{E}$ by using the translation from Boolean equation systems to parity games given in Section 2.5. We can construct from $\mathcal{E}'$ a winning strategy for player $\exists$ in the parity game $G = (V, E, v_0, \Omega)$ in the following way.

For all nodes $v_i \in V_\exists$ of game G where it is player $\exists$'s turn to move, define a strategy $\sigma_\exists$ for $\exists$ to be $\sigma_\exists(v_i) = v_j$ iff $\sigma_i x_i = x_j$ is an equation of $\mathcal{E}'$. That is, the strategy $\sigma_\exists$ for player $\exists$ is to choose in every node belonging to player $\exists$ the successor node which corresponds to the variable appearing also in the right-hand side expression of the i-th equation in the conjunctive Boolean equation system $\mathcal{E}'$.

It is then easy to see that for the game G player $\exists$ wins every play by playing according to strategy $\sigma_\exists$. To see this recall that, by the assumption, $[\![\mathcal{E}']\!] = 1$ holds. Then, by Lemma 41, the system $\mathcal{E}'$ does not contain any μ labelled variables that depend on x_1 and are self-dependent. The crucial observation here is that the dependency graph of $\mathcal{E}'$ contains all and only those paths which correspond to the plays of the game G where the strategy $\sigma_\exists$ is followed. Consequently, there cannot be any play of the game G that is won by player $\forall$, and where player $\exists$ plays according to $\sigma_\exists$.

As $\exists$ has a winning strategy in G, by Theorem 23, it follows that the solution to $\mathcal{E}$ is $[\![\mathcal{E}]\!] = 1$. □

In the same way for the dual case, from each Boolean equation system $\mathcal{E}$ containing both disjunctive and conjunctive equations we may construct a subsystem $\mathcal{E}'$ which is in a disjunctive form. To obtain a disjunctive form subsystem $\mathcal{E}'$ of $\mathcal{E}$, we remove in every conjunctive equation of $\mathcal{E}$ exactly one conjunct; otherwise the system $\mathcal{E}$ is unchanged.

Then, we have the following lemma.

Lemma 50 *Let $\mathcal{E}$ be a standard form Boolean equation system. Then the following are equivalent:*

1. $[\![\mathcal{E}]\!] = 0$
2. *There is a disjunctive subsystem $\mathcal{E}'$ of $\mathcal{E}$ with the solution* $[\![\mathcal{E}']\!] = 0$.

Proof:
Immediate from Lemma 49, together with Lemma 36. □

Let us illustrate the above lemmas with a simple example.

Example 51 *Recall the Boolean equation system*

$$\mathcal{E}_1 \equiv (\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee x_3)(\nu x_3 = x_3).$$

of Example 10. There is only one conjunctive equation $\nu x_1 = x_2 \wedge x_1$, yielding two possible disjunctive subsystems which can be constructed from $\mathcal{E}_1$:

- *if we throw away the conjunct x_2, then we obtain:*

$$\mathcal{E}_1' \equiv (\nu x_1 = x_1)(\mu x_2 = x_1 \vee x_3)(\nu x_3 = x_3)$$

- *if we throw away the conjunct x_1, then we obtain:*

$$\mathcal{E}_1'' \equiv (\nu x_1 = x_2)(\mu x_2 = x_1 \vee x_3)(\nu x_3 = x_3).$$

Using, for example, Lemma 40, we can see that these disjunctive subsystems have the solutions $[\![\mathcal{E}_1']\!] = [\![\mathcal{E}_1'']\!] = 1$. By Lemma 50, a solution to $\mathcal{E}_1$ is $[\![\mathcal{E}_1]\!] = 1$ as expected.

In the next section we will see the application of the above lemmas to give a compact encoding of the problem of solving alternating, general blocks of Boolean equation systems as the problem of finding stable models of normal logic programs.

6.3 From General Blocks to Logic Programs

Consider a standard form, alternating block of a Boolean equation system. This block itself can be seen as a standard form Boolean equation system, call it $\mathcal{E}$. We construct a logic program which captures the local solution $[\![\mathcal{E}]\!]$ of $\mathcal{E}$. In general, there are two ways of constructing the logic program depending on whether Lemma 49 or Lemma 50 is used.

Let us first consider the case where Lemma 50 is used. In particular, this translation is optimal when the number of conjunctive equations of $\mathcal{E}$ is less than (or equal to) the number of disjunctive equations, or that no conjunction symbols occur in the right-hand sides of $\mathcal{E}$. We construct the following logic program $\Pi_\vee(\mathcal{E})$ to capture the solution of $\mathcal{E}$.

The idea is that $\Pi_\vee(\mathcal{E})$ is a ground program which is polynomial in the size of $\mathcal{E}$. We give a compact description of $\Pi_\vee(\mathcal{E})$ as a program with variables. This program consists of the rules

$$depends(1). \tag{7}$$

$$depends(Y) \leftarrow dep(X, Y), depends(X). \tag{8}$$

$$reached(X, Y) \leftarrow nu(X), dep(X, Y), Y \geq X. \tag{9}$$

$$reached(X, Y) \leftarrow reached(X, Z), dep(Z, Y), Y \geq X. \tag{10}$$

$$\leftarrow depends(Y), reached(Y, Y), nu(Y). \tag{11}$$

extended for each equation $(\sigma_i x_i = \alpha_i)$ of $\mathcal{E}$ by

$$dep(i, j). \qquad \text{if } \alpha_i = x_j \tag{12}$$

$$dep(i, j).\ dep(i, k). \qquad \text{if } \alpha_i = (x_j \vee x_k) \tag{13}$$

$$1\ \{dep(i, j), dep(i, k)\}\ 1. \quad \text{if } \alpha_i = (x_j \wedge x_k) \tag{14}$$

and by $nu(i)$. for each variable x_i such that $\sigma_i = \nu$.

The informal idea of the translation is that for the solution $[\![\mathcal{E}]\!]$ of $\mathcal{E}$, $[\![\mathcal{E}]\!] = 0$ iff $\Pi_\vee(\mathcal{E})$ has a stable model. This is captured in the following way. The system $\mathcal{E}$ is turned effectively into a disjunctive system by making a choice between $dep(i, j)$ and $dep(i, k)$ for each conjunctive equation $(\sigma_i x_i = x_j \wedge x_k)$. Hence, each stable model corresponds to a disjunctive system constructed from $\mathcal{E}$ and vice versa.

The translation can be exemplified as follows.

Example 52 *Recall the Boolean equation system $\mathcal{E}_1$ of Example 51. The program $\Pi_\vee(\mathcal{E}_1)$ consists of the rules 7-11 extended with rules:*

$$
\begin{array}{l}
1\ \{dep(1,2), dep(1,1)\}\ 1. \\
dep(2,1).\ dep(2,3). \\
dep(3,3). \\
nu(1).\ nu(3).
\end{array}
$$

The dual case, i.e. where Lemma 49 is used, goes along exactly the same lines. In particular, the dual translation is optimal in the case where the number of disjunctive equations of $\mathcal{E}$ is less than the number of conjunctive equations, or where no disjunction symbols occur in the right-hand sides of $\mathcal{E}$. We construct the dual logic program $\Pi_\wedge(\mathcal{E})$ to capture the solution of $\mathcal{E}$. This program consists of the rules

$$depends(1). \tag{15}$$

$$depends(Y) \leftarrow dep(X,Y), depends(X). \tag{16}$$

$$reached(X,Y) \leftarrow mu(X), dep(X,Y), Y \geq X. \tag{17}$$

$$reached(X,Y) \leftarrow reached(X,Z), dep(Z,Y), Y \geq X. \tag{18}$$

$$\leftarrow depends(Y), reached(Y,Y), mu(Y). \tag{19}$$

extended for each equation $(\sigma_i x_i = \alpha_i)$ of $\mathcal{E}$ by

$$dep(i,j). \qquad \text{if } \alpha_i = x_j \tag{20}$$

$$dep(i,j).\ dep(i,k). \qquad \text{if } \alpha_i = (x_j \wedge x_k) \tag{21}$$

$$1\ \{dep(i,j), dep(i,k)\}\ 1. \quad \text{if } \alpha_i = (x_j \vee x_k) \tag{22}$$

and by $mu(i)$. for each variable x_i such that $\sigma_i = \mu$.

The informal idea of the dual translation is that for the solution $[\![\mathcal{E}]\!]$ of $\mathcal{E}$, $[\![\mathcal{E}]\!] = 1$ iff $\Pi_\wedge(\mathcal{E})$ has a stable model. This is captured in the following way. The system $\mathcal{E}$ is turned effectively into a conjunctive system by making a choice between $dep(i,j)$ and $dep(i,k)$ for each disjunctive equation $(\sigma_i x_i = x_j \vee x_k)$. Hence, each stable model corresponds to a conjunctive system constructed from $\mathcal{E}$ and vice versa.

6.4 Correctness of the Translation

In this section, we prove formally the correctness of the translation. In particular, the main result below can be established by Lemmas 40 and 50

Theorem 53 *Let $\mathcal{E}$ be a standard form, alternating Boolean equation system. Then $[\![\mathcal{E}]\!] = 0$ iff $\Pi_\vee(\mathcal{E})$ has a stable model.*

Proof:
Consider a system $\mathcal{E}$ and its translation $\Pi_\vee(\mathcal{E})$. The rules (12–14) effectively capture the dependency graphs of the disjunctive systems that can be constructed from $\mathcal{E}$. More precisely, there is a one to one correspondence between the stable models of the rules (12–14) and disjunctive systems that can be constructed from $\mathcal{E}$ such that for each stable model Δ, there is exactly one disjunctive system $\mathcal{E}'$ with the dependency graph $G_{\mathcal{E}'} = (V, E)$ where $V = \{i \mid dep(i,j) \in \Delta \text{ or } dep(j,i) \in \Delta\}$ and $E = \{(i,j) \mid dep(i,j) \in \Delta\}$.

Now, one can establish that each stable model Δ of $\Pi(\mathcal{E})$ is an extension of a stable model Δ' of the rules (12–14), i.e., of the form $\Delta = \Delta' \cup \Delta''$ such that in the corresponding dependency graph there is no variable x_j such that $\sigma_j = \nu$ and x_1 depends on x_j and x_j is self-dependent. By Lemma 50, $[\![\mathcal{E}]\!] = 0$ iff there is a disjunctive system $\mathcal{E}'$ that can be constructed from $\mathcal{E}$ for which $[\![\mathcal{E}']\!] = 0$. By Lemma 40, for a disjunctive system $\mathcal{E}'$, $[\![\mathcal{E}']\!] = 1$ holds if and only if there is a variable x_j such $\sigma_j = \nu$ and x_1 depends on x_j and x_j is self-dependent. Hence, $\Pi(\mathcal{E})$ has a stable model iff there is a disjunctive system $\mathcal{E}'$ that can be constructed from $\mathcal{E}$ whose dependency graph has no variable x_j such that $\sigma_j = \nu$ and x_1 depends on x_j and x_j is self-dependent iff there is a disjunctive system $\mathcal{E}'$ with $[\![\mathcal{E}']\!] \neq 1$, i.e., $[\![\mathcal{E}']\!] = 0$ iff $[\![\mathcal{E}]\!] = 0$. □

Similar theorem holds also for the dual program, which allows us to solve all alternating blocks of standard form Boolean equation systems.

Theorem 54 *Let $\mathcal{E}$ be a standard form, alternating Boolean equation system. Then $[\![\mathcal{E}]\!] = 1$ iff $\Pi_\wedge(\mathcal{E})$ has a stable model.*

Proof:
In the similar way as for the dual program in Theorem 53. □

Perhaps, further explanation of our translations is in order here. Although $\Pi_\vee(\mathcal{E})$ and $\Pi_\wedge(\mathcal{E})$ are given using variables for the theorems above, a finite ground instantiation of the programs are sufficient.

To exemplify the finite ground instantiation for the case $\Pi_\vee(\mathcal{E})$ we introduce a relation *depDom* such that $\mathit{depDom}(i, j)$ holds iff there is an equation $(\sigma_i x_i = \alpha_i)$ of $\mathcal{E}$ with x_j occurring in α_i. Now, the sufficient ground instantiation is obtained by substituting variables X, Y in the rules (8–9) with all pairs i, j such that $\mathit{depDom}(i, j)$ holds, substituting variables X, Y, Z in rule (10) with all triples l, i, j such that $nu(l)$ and $\mathit{depDom}(i, j)$ hold and variable Y in rule (11) with every i such that $nu(i)$ holds. This means also that such conditions can be added as domain predicates to the rules without losing the correctness of the translation. For example, rule (10) could be replaced by

$$\mathit{reached}(X, Y) \leftarrow nu(X), \mathit{depDom}(Z, Y), \mathit{reached}(X, Z),$$

$$\mathit{dep}(Z, Y), Y \geq X.$$

Notice that such conditions make the rules domain-restricted (i.e. each variable in a rule occurs also in a positive domain predicate in the rule body) as required, e.g., by the `Smodels` system.

When incorporating the above solution technique into a general procedure defined in Section 3, a drawback of the above encodings might be that they can only find a *local* solution to a general block of a Boolean equation system as defined in Definition 9. This means that, in order to solve all variables of a general form block of a Boolean equation system one needs to solve each variable separately with (possibly) distinct encodings. This calls for an extension of the answer set programming encoding for finding global solutions as defined in Definition 11. However, the global encoding is left for future work.

6.5 Implementation Issues

In Section 8, we will describe some experimental results on solving alternating Boolean equation systems with the approach presented in Section 6. We will demonstrate the technique on series of examples which are solved using the `Smodels` system (`http://www.tcs.hut.fi/Software/smodels/`) as the ASP solver.

An advantage of using `Smodels` is that it provides an implementation for cardinality constraint rules used in our translation, and includes primitives supporting directly such constraints without translating them first to corresponding normal rules. We have used `Smodels` version 2.26 to find the solutions and `lparse` 1.0.13 for parsing and grounding the input.

The encodings that we have used for the experiments are the translations represented in Section 6.3 with a couple of optimizations. Here, we present these optimizations for the case $\Pi_\vee(\mathcal{E})$; the dual case $\Pi_\wedge(\mathcal{E})$ is omitted but it can be treated in the similar way.

First, when encoding of dependencies as given in rules (12–14) we differentiate those dependencies where there is a choice from those where there is none, i.e., for each equation $(\sigma_i x_i = \alpha_i)$ of $\mathcal{E}$ we add

$$
\begin{array}{ll}
\mathit{ddep}(i,j). & \text{if } \alpha_i = x_j \\
\mathit{ddep}(i,j).\mathit{ddep}(i,k). & \text{if } \alpha_i = (x_j \vee x_k) \\
1\{\mathit{cdep}(i,j), \mathit{cdep}(i,k)\}1.\ \mathit{depDom}(i,j).\ \mathit{depDom}(i,k). & \text{if } \alpha_i = (x_j \wedge x_k)
\end{array}
$$

instead of rules (12–14). Secondly, in order to make use of this distinction and to allow for intelligent grounding, rules (8–10) are rewritten using the above predicates as domain predicates in the following way.

$$
\begin{aligned}
\mathit{depends}(Y) &\leftarrow \mathit{ddep}(X,Y), \mathit{depends}(X). \\
\mathit{depends}(Y) &\leftarrow \mathit{depDom}(X,Y), \mathit{cdep}(X,Y), \mathit{depends}(X). \\
\mathit{reached}(X,Y) &\leftarrow \mathit{nu}(X), \mathit{ddep}(X,Y), Y \geq X. \\
\mathit{reached}(X,Y) &\leftarrow \mathit{nu}(X), \mathit{depDom}(X,Y), \mathit{cdep}(X,Y), Y \geq X. \\
\mathit{reached}(X,Y) &\leftarrow \mathit{nu}(X), \mathit{reached}(X,Z), \mathit{ddep}(Z,Y), Y \geq X. \\
\mathit{reached}(X,Y) &\leftarrow \mathit{nu}(X), \mathit{depDom}(Z,Y), \mathit{reached}(X,Z), \\
&\qquad \mathit{cdep}(Z,Y), Y \geq X.
\end{aligned}
$$

Notice that the size of the resulting translations are $O(e \cdot v) = O(v^2)$ where e is the size of the Boolean equation system and v is the number of left-hand side variables.

It is worthwhile to observe that these kinds of logic programming encodings could also be used to derive encodings into propositional satisfiability because there exist translations from normal logic programs to propositional satisfiability, for example [51]. If one uses the above translations from general form Boolean equation systems to normal logic programs as a basis for such mappings to propositional satisfiability, it will not straightforwardly lead to compact encodings into propositional logic. The translation of [51] is subquadratic; it is roughly of size $O(n \log n)$ where n is the size of the logic program. Therefore, in the next section we will study how to give direct encodings into propositional formulas.

For instance, it turns out that direct encodings of Boolean equation systems into propositional satisfiability are more compact than the propositional encoding obtained by first translating a Boolean equation system into logic program and using the translation in [51].

Next, before turning to experimental issues, we present another approach to solve alternating blocks of Boolean equation systems through reductions to difference logic and propositional logic.

7 SOLVING GENERAL BLOCKS BY A REDUCTION TO SATISFIABILITY

As previous section, this section addresses the question of how to obtain practically effective methods to solve hard, general form Boolean equation systems for which no polynomial time solution algorithms are known to exist. In this section, the main idea of our approach is to define encodings from general blocks of a Boolean equation system into satisfiability problems. The encodings are given in such a way that solutions to Boolean equation systems can be obtained by solving the corresponding satisfiability problems. Thus, the results of this section give solving methods for general form Boolean equation systems using various satisfiability solvers.

The original idea of μ-annotations described in subsection 7.2 is from [54]. There Jurdziński presents a progress measure algorithm for solving parity games which is based on the μ-annotations. The proposional encoding given in subsection 7.5 is initially from [63], whereas the other results in this section are essentially from [47].

We first motivate this kind of approach, and introduce some important notions together with their properties. Then, we develop mappings from general blocks of a Boolean equation system to satisfiability problems. We also show the correctness of our translations and estimate the sizes of the resulting formulas.

7.1 Solving General Form Blocks with Satisfiability Solvers

The propositional satisfiability problem (SAT) is NP-complete [24]. Recall from Theorem 27 in subsection 2.7 that the satisfiability problem for difference logic is NP-complete too. Hence, unlike solving Boolean equation systems, both of these satisfiability problems are widely not believed to admit polynomial time solution algorithms.

Yet, in reality there are modern SAT solvers that are efficient in practice for many large instances of the problem, for example zCHAFF [77]. Such solvers are used successfully in computer aided verification tasks, for example in bounded model checking [16]. Also, quite efficient solvers have recently been developed for testing the satisfiability of difference logic, for instance DPLL(T) [79]. In recent years, many difference logic solvers have effectively been used to solve a wide range of computationally hard problems in the realm of formal verification.

Motivated by the success of these kinds of satisfiability solvers, we present an approach to solve Boolean equation systems by a reduction to satisfiability problems. Here, the research hypothesis is that one can employ the recent results of satisfiability solving techniques to significantly improve the methods to solve general form Boolean equation systems, thus leading to practically efficient model checking techniques for the μ-calculus. The techniques in question include, e.g., learning combined with restarts which are often implemented in modern SAT solvers.

The reduction is done in two stages, first into difference logic [79]. This gives rise to a solving method for general Boolean equation systems through efficient difference logic solvers such as DPLL(T) [79]. In the second stage the integer variables and constraints of the difference logic encoding are replaced with a set of Boolean variables and constraints on them, giving rise to a pure SAT encoding of the problem. This enables a much wider range of solvers to be used than the difference logic framework, for example zChaff [77].

One might argue that, theoretically, the proposed approach is not interesting because it is obvious that such reductions must exist. Furthermore, NP-complete satisfiability problems are believed to be harder than solving general Boolean equation systems which, as mentioned before, are contained in NP ∩ co-NP and even in UP ∩ co-UP. Again, it is expected that clever heuristics and advanced search space pruning techniques implemented in current satisfiability solvers can make up for this, and the reductions may result in solution techniques for general form Boolean equation systems which are efficient in practice.

However, developing a *computationally attractive* reduction to satisfiability problems is often a non-trivial and challenging task. As mentioned before, all known satisfiability checkers use algorithms whose worst case running times are exponential in the number of variables in the formulas. Moreover, if the encoding is of substantial size, this can confuse the search heuristics and introduce significant computational overhead in search space pruning. Hence, a computationally interesting reduction should introduce as few variables as possible, and be of low polynomial size. This means that the problem to be solved by a reduction to SAT often needs to be studied carefully in order to understand the essential properties of the solutions. This is what we aim to do in the following subsections.

Our reduction is based on a comment by Emerson where he shows inclusion of the model checking problem for the μ-calculus in the complexity class NP. Essentially, Emerson writes in [30] that the inclusion of the μ-calculus model checking problem in the class NP can be seen as follows:

> Guess a rank for each μ-subformula at each state in a transition system. Show that the lexicographic order on the tuples through the transition system is well-founded.

Following the idea of Emerson about ranks, with the aim of a characterisation of solutions to Boolean equation systems, we define the notion of μ-annotation for dependency graphs of Boolean equation systems. The notion of μ-annotation is closely related to Jurdziński's progress measures for parity games [54].

Intuitively, progress measures are data structures consisting of local constraints which together ensure the global property that for all cycles of a parity game conforming to a certain strategy the parity of the least occurring priority in the cycle is even. Jurdziński presents a progress measure algorithm [54] for parity games that sets these data structures to an initial value and updates them iteratively.

In contrast to this algorithmic approach, our reductions leave it entirely to the satisfiability solvers to find the final values of the μ-annotations. Since our approach is not in any way an iterative procedure, we call the data structures μ-annotations in order to stress their static nature. But, we attribute the theory of μ-annotations to Jurdziński explicitly. Nevertheless, we provide correctness proofs that differ slightly from Jurdziński's work [54] in the related setting of parity games.

7.2 Characterising Solutions with μ-Annotations

The key notion in our reduction is a μ-annotation which is used to give a characterization of solutions to Boolean equation systems. Intuitively, it is a labeling of vertices of a dependency graph with tuples of natural numbers satisfying certain properties. Let us define formally such labelings for variable dependency graphs.

Given a general, standard form Boolean equation system $\mathcal{E}$ with equations

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2) \ldots (\sigma_n x_n = \alpha_n)$$

and dependency graph $G = (V, E, \ell)$, we define $\mathit{Odd}(G) = \{p \mid p \text{ is odd and } \mathcal{H}(v) = p \text{ for some } v \in V\}$, $\mathit{Odd}^{<p}(G) = \mathit{Odd}(G) \cap \{i \in \mathbb{N} \mid 1 \leq i < p\}$, and $\mathit{mop}(G) = \max\ \mathit{Odd}(G)$.

Definition 55 (**μ-Annotation**) *A μ-annotation for G is a function $\eta : V \to \mathbb{N}^{|\mathit{Odd}(G)|}$ that assigns to each $v \in V$ a tuple $\overline{a} = (a_1, a_3, \ldots, a_{\mathit{mop}(G)}) \in \mathbb{N}^{|\mathit{Odd}(G)|}$.*

Intuitively, a μ-annotation for a dependency graph G is simply a function $\eta : V \to \mathbb{N}^{|\mathit{Odd}(G)|}$ that assigns to each $v \in V$ an integer tuple with as many components as there are odd labels in the alternation hierarchy $\mathcal{H}$.

Given two tuples $\overline{a} = (a_1, \ldots, a_{\mathit{mop}(G)})$ and $\overline{b} = (b_1, \ldots, b_{\mathit{mop}(G)})$ of a μ-annotation for G and a $p \leq \mathit{mop}(G)$ (not necessarily odd), we define an ordering on tuples in the following way

$$\overline{a} \trianglelefteq_p \overline{b} \quad \text{iff} \quad \begin{cases} a_i \leq b_i & \text{for all } i \in \mathit{Odd}^{<p}(G) \quad \text{if } p \text{ is even,} \\ a_p < b_p \text{ and } a_i \leq b_i & \text{for all } i \in \mathit{Odd}^{<p}(G) \quad \text{otherwise.} \end{cases}$$

To demonstrate the ordering on tuples we provide here a simple example.

Example 56 *Suppose we have $\mathit{Odd}(G) = \{1, 3, 5\}$. Then, clearly $\mathit{mop}(G) = 5$. Consider two tuples $\overline{a} = (a_1, a_3, a_5) \in \mathbb{N}^{|\mathit{Odd}(G)|}$ and $\overline{b} = (b_1, b_3, b_5) \in \mathbb{N}^{|\mathit{Odd}(G)|}$. Let $\overline{a} = (2, 4, 3)$ and $\overline{b} = (2, 5, 3)$. Now, we have $\overline{a} \trianglelefteq_2 \overline{b}$ because $a_1 \leq b_1$ holds. On the other hand, $\overline{a} \trianglelefteq_1 \overline{b}$ does not hold because it is not the case that $a_1 < b_1$. Finally, we have that $\overline{a} \trianglelefteq_3 \overline{b}$ because both $a_3 < b_3$ and $a_1 \leq b_1$ hold.*

In the following, we will write $\overline{a}^{(p)}$ for every $p \in \mathit{Odd}(G)$ to denote the p-component of $\overline{a}$. We will also write $\mathit{succ}(v)$ for the set $\{w \mid (v, w) \in E\}$.

Next, we define *successful* μ-annotations for *conjunctive* Boolean equation systems. Such μ-annotations can then be used to give another characterization of solutions to Boolean equation systems.

Definition 57 (**Successful μ-annotation**) *Given a conjunctive Boolean equation system*

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2) \ldots (\sigma_n x_n = \alpha_n)$$

with dependency graph $G = (V, E, \ell)$ and alternation hierarchy $\mathcal{H}$, its μ-annotation is called successful *iff for all $(v, w) \in E$ we have that $\eta(w) \trianglelefteq_{\mathcal{H}(w)} \eta(v)$.*

The following property about μ-annotations holds for conjunctive form Boolean equation systems.

Theorem 58 *Let $G = (V, E, \ell)$ be a dependency graph of a conjunctive Boolean equation system and let $v \in V$. There is a successful μ-annotation for G iff G does not contain a path from v to a cycle of G where the smallest $\mathcal{H}$ label appearing in this cycle is odd.*

Proof:
($\Rightarrow$) Let η be a successful μ-annotation for G. Assume that G contains a cycle $(v_l, v_{l+1}, \ldots, v_k)$ such that there is a path from node v to some node appearing in this cycle. Let v_l be the node of cycle $(v_l, v_{l+1}, \ldots, v_k)$ having the smallest label in the alternation hierarchy and let $p = \mathcal{H}(v_l)$ be odd. Then, we have $\eta(v_l)^{(p)} > \eta(v_{l+1})^{(p)} \geq \ldots \geq \eta(v_k)^{(p)} \geq \eta(v_l)^{(p)}$ which is a contradiction. Hence, there cannot exist a path from v to a cycle of G where the smallest $\mathcal{H}$ label appearing in the cycle is odd.

($\Leftarrow$) We define the μ-annotation from graph G. For every $p \in Odd(G)$ let $E_p = \{(v, w) \in E \mid \mathcal{H}(w) \geq p\}$ be the set of edges in G that lead to nodes with priorities not less than p. Furthermore, for every $v \in V$ let $W_v^p = \{w \mid (v, w) \in E_p^+\} \cap \{v \in V \mid p = \mathcal{H}(v)\}$ be the set of nodes that have priority p and are reachable from v via this relation, where E_p^+ is the transitive closure of E_p.

For every $v \in V$ and every $p \in Odd(G)$, we define a μ-annotation η for G as $\eta(v)^{(p)} = |W_v^p|$ and show that this annotation is successful.

Suppose η is not successful. Then, there is a $v \in V$ and a $w \in succ(v)$ s.t. $\eta(w) \trianglelefteq_{\mathcal{H}(w)} \eta(v)$ does not hold. Since $w \in succ(v)$, i.e. w is reachable from v, we have $W_w^p \subseteq W_v^p$ and, hence, $\eta(w)^{(p)} \leq \eta(v)^{(p)}$ for all $p \leq \mathcal{H}(w)$. Since $\eta(w) \trianglelefteq_{\mathcal{H}(w)} \eta(v)$ does not hold by the assumption, it must be the case that $\mathcal{H}(w)$ is odd and $\eta(w)^{(\mathcal{H}(w))} \not< \eta(v)^{(\mathcal{H}(w))}$. Then, it must be the case that $\eta(w)^{(\mathcal{H}(w))} = \eta(v)^{(\mathcal{H}(w))}$, and we have $W_w^{\mathcal{H}(w)} = W_v^{\mathcal{H}(w)}$. This means that $(w, v) \in E^+_{\mathcal{H}(w)}$, i.e. v is reachable from w whilst not seeing a node with alternation hierarchy label smaller than $\mathcal{H}(w)$. But, then there is a cyclic path on which the least $\mathcal{H}$ label seen infinitely often is $\mathcal{H}(w)$ which is odd. We conclude that η must be a successful μ-annotation. □

Importantly, Theorem 58 gives another characterization for the solution of conjunctive Boolean equation systems.

A direct consequence of the proof of Theorem 58 is the fact that the domain of annotation values can be bounded by a relatively small number. The same observation has also been made regarding the progress measures [54]. Consequently, the following property holds for the domain of μ-annotation values.

Corollary 59 *Let $G = (V, E, \ell)$ be a dependency graph and $\mathcal{H}$ an alternation hierarchy of a conjunctive Boolean equation system $\mathcal{E}$. Let $n_p = |\{v \in V \mid \mathcal{H}(v) = p\}|$ for all $p \in Odd(G)$. There is a successful μ-annotation for G iff there is a successful μ-annotation η for G s.t. for all $v \in V$: $\eta(v) \in \{0, \ldots, n_1\} \times \ldots \times \{0, \ldots, n_{mop(G)}\}$.*

This property plays an essential role in our propositional SAT encoding. We now turn to define the encodings based on μ-annotations.

7.3 Encoding Solutions in Difference Logic

Given a standard form Boolean equation system

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2) \ldots (\sigma_n x_n = \alpha_n)$$

with dependency graph $G = (V, E, \ell)$ and alternation hierarchy $\mathcal{H}$, we build a difference logic formula Φ_G that is satisfiable iff the solution to $\mathcal{E}$ is $[\![\mathcal{E}]\!] = 1$.

The formula Φ_G contains Boolean variables S_v for every $v \in V$ and $T_{v,w}$ for every $(v, w) \in E$. They are used to guess a subgraph of G inducing a conjunctive Boolean equation system $\mathcal{E}'$ constructed from $\mathcal{E}$ where the solution to $\mathcal{E}'$ is $[\![\mathcal{E}']\!] = 1$.

In addition, Φ_G contains integer variables x_p^v for every $v \in V$ and every $p \in \mathit{Odd}(G)$ in order to model a μ-annotation.

First, we partition V into two sets $V_\vee = \{i \in V \mid (\sigma_i x_i = \alpha_i) \text{ is disjunctive}\}$ and $V_\wedge = \{i \in V \mid (\sigma_i x_i = \alpha_i) \text{ is conjunctive}\}$. The formula Φ_G is defined to be

$$(S_1 \wedge \Phi_\vee \wedge \Phi_\wedge \wedge \Phi_V \wedge \Phi_A).$$

Here, the subformulas are defined as follows:

$$\begin{aligned}
\Phi_\vee &= \bigwedge_{v \in V_\vee} (S_v \rightarrow \bigvee_{(v,w)\in E} T_{v,w}), \\
\Phi_\wedge &= \bigwedge_{v \in V_\wedge} (S_v \rightarrow \bigwedge_{(v,w)\in E} T_{v,w}), \\
\Phi_V &= \bigwedge_{v \in V, v \neq 1} ((\bigvee_{(w,v)\in E} T_{w,v}) \rightarrow S_v), \text{ and} \\
\Phi_A &= \bigwedge_{(v,w)\in E} (T_{v,w} \rightarrow \Psi_{v,w}),
\end{aligned}$$

where $\Psi_{v,w}$ is given by

$$\Psi_{v,w} = \begin{cases} \bigwedge_{p \in \mathit{Odd}^{<\mathcal{H}(w)}(G)} (x_p^v \geq x_p^w) & \text{if } \mathcal{H}(w) \text{ even,} \\ (x_{\mathcal{H}(w)}^v > x_{\mathcal{H}(w)}^w) \wedge \bigwedge_{p \in \mathit{Odd}^{<\mathcal{H}(w)}(G)} (x_p^v \geq x_p^w) & \text{otherwise.} \end{cases}$$

Intuitively, the above translation from the Boolean equation system $\mathcal{E}$ to the formula Φ_G is such that the satisfying assignments to Φ_G capture the conjunctive Boolean equation systems which can be constructed from $\mathcal{E}$ and have successful μ-annotations. Here, the Boolean variables of the formula Φ_G which are assigned a value $\top$ model the conjunctive Boolean equation system. In case there is no satisfying assignment to the formula Φ_G, then there is no way of constructing a conjunctive Boolean equation system from $\mathcal{E}$ with successful μ-annotation.

The following example demonstrates the translation of a Boolean equation system into a difference logic formula.

Example 60 *As an example of a difference logic formula capturing the solution to a general Boolean equation system, we give the translation Φ_G of the following equations over $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$:*

$$(\mu x_1 = x_2 \wedge x_3)(\nu x_2 = x_3 \vee x_4)(\mu x_3 = x_2 \wedge x_4)(\mu x_4 = x_2 \vee x_3).$$

First, we see that the alternation hierarchy is given by $\mathcal{H}(1) = 1$, $\mathcal{H}(2) = 2$, $\mathcal{H}(3) = 3$ and $\mathcal{H}(4) = 4$. The dependency graph of this Boolean equation system is $G = (V, E)$ where

$$\begin{aligned} V =& \{1, 2, 3, 4\}, \textit{ and} \\ E =& \{(1, 2), (1, 3), \\ & (2, 3), (2, 4), \\ & (3, 2), (3, 4), \\ & (4, 2), (4, 3)\}. \end{aligned}$$

The set $V = \{1, 2, 3, 4\}$ is partitioned into $V_\vee = \{2, 4\}$ and $V_\wedge = \{1, 3\}$. So, the formula Φ_G is simply

$$(S_1 \wedge \Phi_\vee \wedge \Phi_\wedge \wedge \Phi_V \wedge \Phi_A),$$

where the subformulas are given as follows

$$\begin{aligned} \Phi_\vee =& (S_2 \rightarrow (T_{2,3} \vee T_{2,4}))\wedge \\ & (S_4 \rightarrow (T_{4,2} \vee T_{4,3})), \\ \Phi_\wedge =& (S_1 \rightarrow (T_{1,2} \wedge T_{1,3}))\wedge \\ & (S_3 \rightarrow (T_{3,2} \wedge T_{3,4})), \\ \Phi_V =& ((T_{4,2} \vee T_{3,2}) \rightarrow S_2)\wedge \\ & ((T_{1,3} \vee T_{2,3} \vee T_{4,3}) \rightarrow S_3)\wedge \\ & (T_{2,4} \vee T_{3,4}) \rightarrow S_4), \textit{ and} \\ \Phi_A =& (T_{1,2} \rightarrow (x_1^1 \geq x_1^2))\wedge \\ & (T_{1,3} \rightarrow ((x_1^1 \geq x_1^3) \wedge (x_3^1 > x_3^3)))\wedge \\ & (T_{2,3} \rightarrow ((x_1^2 \geq x_1^3) \wedge (x_3^2 > x_3^3)))\wedge \\ & (T_{2,4} \rightarrow ((x_1^2 \geq x_1^4) \wedge (x_3^2 \geq x_3^4)))\wedge \\ & (T_{3,2} \rightarrow (x_1^3 \geq x_1^2)\wedge \\ & (T_{3,4} \rightarrow ((x_1^3 \geq x_1^4)) \wedge (x_3^3 \geq x_3^4))\wedge \\ & (T_{4,2} \rightarrow (x_1^4 \geq x_1^2))\wedge \\ & (T_{4,3} \rightarrow ((x_1^4 \geq x_1^3) \wedge (x_3^4 > x_3^3))). \end{aligned}$$

Clearly, the solution to the above Boolean equation system is 1, and as expected the formula Φ_G is satisfiable.

7.4 Correctness of the Encoding

Next, we prove the correctness of the difference logic encoding for Boolean equation systems. The correctness of the translation in the previous subsection can be stated as the following theorem.

Theorem 61 *The solution to $\mathcal{E}$ with dependency graph $G_\mathcal{E}$ and alternation hierarchy $\mathcal{H}$ is $[\![\mathcal{E}]\!] = 1$ iff the difference logic formula Φ_G is satisfiable.*

Proof:
($\Rightarrow$) Suppose the solution to $\mathcal{E}$ is $[\![\mathcal{E}]\!] = 1$. By Lemma 49, there is a conjunctive Boolean equation system $\mathcal{E}'$ with the solution $[\![\mathcal{E}']\!] = 1$. Let $G_{\mathcal{E}'} =$

(V', E', ℓ') be the dependency graph of $\mathcal{E}'$. This gives rise to an assignment β of the propositional variables S_v and $T_{v,w}$ for any $(v,w) \in E$: $\beta(S_v) = \top$, resp. $\beta(T_{v,w}) = \top$, if there is a path in $G_{\mathcal{E}'}$ which visits the node $v \in V'$, resp. traverses the edge $(v,w) \in E'$. It is not hard to see that the conjuncts S_1, $\Phi_\vee$, $\Phi_\wedge$, and Φ_V are satisfied by this assignment β.

According to Theorem 58 there is a successful μ-annotation η for $G_{\mathcal{E}'}$. This gives rise to an assignment β to the non-propositional variables x_p^v for all the nodes of the dependency graph $G_{\mathcal{E}'} = (V', E', \ell')$ defined by $\beta(x_p^v) = \eta(v)^{(p)}$. Since η is successful, we have $\eta(w) \trianglelefteq_{\mathcal{H}(w)} \eta(v)$ for all $(v,w) \in E'$, and hence, the conjunct Φ_A is also satisfied. Altogether, there is a satisfying assignment for Φ_G.

$(\Leftarrow)$ Suppose β is a satisfying variable assignment for Φ_G. It is easy to derive from this a dependency graph $G_{\mathcal{E}'} = (V, E', \ell)$ of a conjunctive Boolean equation system $\mathcal{E}'$ as follows: for every node $v \in V_\vee$ such that $\beta(S_v) = \top$ add an arbitrary edge (v,w) to E' such that $\beta(T_{v,w}) = \top$, and for every node $v \in V_\wedge$ such that $\beta(S_v) = \top$ add all edges $(v,w) \in E$ to E'. The conjuncts S_1, $\Phi_\vee$, $\Phi_\wedge$ and Φ_V ensure that suitable edges needed by the construction above exist and that $G_{\mathcal{E}'}$ induced in this way is indeed a dependency graph of a conjunctive Boolean equation system $\mathcal{E}'$ that can be constructed from $\mathcal{E}$.

Furthermore, we can extract a μ-annotation η for $G_{\mathcal{E}'}$ defined by $\eta(v)^{(p)} = \beta(x_v^p)$ for any $v \in V$ and any $p \in \mathit{Odd}(G)$. If some $\beta(x_v^p)$ turns out to be negative, then it is easy to see from our translation that the integer values of a satisfiable model can be made to positive integers by first offsetting every $\beta(x_{v'}^{p'})$ by a large enough positive integer offset to make all integer variables of an assignment positive, and the formula still remains satisfiable with this assignment consisting of positive values only. It is easy to see that the conjunct Φ_A ensures η is successful μ-annotation for $G_{\mathcal{E}'}$.

We note that, given any Boolean equation system, interchanging subsequent equations with the same fixpoint sign does not influence the solution. Thus, Lemma 41 holds also for the alternation hierarchy labels of dependency graphs in the following sense: we have $[\![\mathcal{E}']\!] = 0$ iff $G_{\mathcal{E}'}$ contains a cycle which is reachable from node 1 and the smallest $\mathcal{H}$ label appearing in this cycle is odd.

But, according to Theorem 58 there cannot exist a path from node $1 \in V$ to a cycle of $G_{\mathcal{E}'}$ where the smallest $\mathcal{H}$ label appearing in this cycle is odd. Therefore, the solution to the conjunctive Boolean equation system $\mathcal{E}'$ corresponding to $G_{\mathcal{E}'}$ has the solution $[\![\mathcal{E}']\!] = 1$. Finally, by Lemma 49 the solution to $\mathcal{E}$ is indeed $[\![\mathcal{E}]\!] = 1$. □

By Theorem 61, a general Boolean equation system can now be solved by first converting the equations into a corresponding difference logic formula, and then testing the satisfiability of the formula.

We now estimate the size of the translation. The size of the encoding is characterized as follows.

Proposition 62 *Given a dependency graph $G = (V, E, \ell)$ of a Boolean equation system $\mathcal{E}$, the size of the difference logic formula Φ_G is $O(|E| \cdot p_{max})$ where $p_{max} = \max\{\mathcal{H}(v) \mid v \in V\}$.*

Recall that, according to our wishlist in the previous subsections, a computationally attractive encoding should introduce only a small number of variables and be of low polynomial size. By Proposition 62, one may conclude that this goal has been achieved.

Many satisfiability solvers for difference logic, such as DPLL(T) [79], assume that an input formula is in conjunctive normal form (CNF). Fortunately, the above size bound can be seen to hold even when Φ_G is required to be in CNF. In fact, by careful analysis of the difference logic formula Φ_G our implementation of the translation is able to convert Φ_G directly to CNF without a blowup, and even without introducing any additional Boolean variables.

7.5 An Encoding into Propositional Logic

We present an encoding of the formula Φ_G for a dependency graph G into propositional logic, i.e. the subset of difference logic with Boolean variables only. Clearly, all that remains to be done is to translate the integer variables and constraints on them of the form $(x^v_p \geq x^w_p)$ and $(x^v_{\mathcal{H}(w)} > x^w_{\mathcal{H}(w)})$. In [63], a similar propositional SAT encoding is given for parity games.

Let $G = (V, E, \ell)$ be the underlying dependency graph and $\mathcal{H}$ the alternation hierarchy. By Corollary 59, the domain of the difference logic variables x^v_p for a fixed p and any v can be bounded by $|V_p| + 1$ where $V_p = \{v \in V \mid \mathcal{H}(v) = p\}$. Let $m_p = \lceil \log_2(|V_p| + 1) \rceil$ be the number of bits needed for a binary encoding of a value in the range $\{0, \ldots, |V_p|\}$. Hence, a set of propositional variables $x^v_{p,i}$ for $i \in \{0, \ldots, m_p - 1\}$ will be used to encode the value of each integer variable x^v_p.

For each $v, w \in V$, each $p \in \mathit{Odd}(G)$ and each $m \geq 1$ we present recursively defined propositional formulas *GreaterOrEquals* and *StrictlyGreater* parametrised by v, w, p, m and stating $0 \leq x^w_p \leq x^v_p < 2^m$, and respectively $0 \leq x^w_p < x^v_p < 2^m$.

$$\begin{aligned}
\mathit{GreaterOrEquals}(v,w,p,0) &= x^w_{p,0} \to x^v_{p,0} \\
\mathit{GreaterOrEquals}(v,w,p,m) &= (x^w_{p,m} \to x^v_{p,m}) \ \wedge \ \big((x^w_{p,m} \vee \neg x^v_{p,m}) \to \\
&\qquad \mathit{GreaterOrEquals}(v,w,p,m-1)\big) \\
\mathit{StrictlyGreater}(v,w,p,0) &= x^v_{p,0} \wedge \neg x^w_{p,0} \\
\mathit{StrictlyGreater}(v,w,p,m) &= (x^w_{p,m} \to x^v_{p,m}) \ \wedge \ \big((x^w_{p,m} \vee \neg x^v_{p,m}) \to \\
&\qquad \mathit{StrictlyGreater}(v,w,p,m-1)\big)
\end{aligned}$$

Intuitively, both formulas assert that the mth bit of x^v_p is greater or equals to the mth bit of x^w_p, and if they are equal then the same has to hold recursively for the next lower bit. However, formula *StrictlyGreater* has to ensure in the base case that at least the values of the lowest bits differ unless some higher bits have differed already.

The encodings of the integer constraints occurring in Φ_G can simply be replaced by

$$\begin{aligned}(x_p^v \geq x_p^w) &\text{ with } \mathit{GreaterOrEquals}(v, w, p, m_p), \text{ and}\\ (x_{\mathcal{H}(w)}^v > x_{\mathcal{H}(w)}^w) &\text{ with } \mathit{StrictlyGreater}(v, w, \mathcal{H}(w), m_{\mathcal{H}(w)}).\end{aligned}$$

In this way, we can keep Φ_G as the name of the formula obtained by replacing all integer variables and all integer constraints on them by their Boolean counterparts as described above.

It is easy to see that the above propositional SAT encoding is correct, and we have the following property.

Theorem 63 *The solution to $\mathcal{E}$ with dependency graph $G_\mathcal{E}$ is $[\![\mathcal{E}]\!] = 1$ iff the propositional logic formula Φ_G (i.e. Φ_G with only Boolean variables) is satisfiable.*

Proof:
Immediate from the replacement of integer variables and constraints on them by Boolean counterparts, together with Theorem 61 and Corollary 59. □

When compared to the difference logic encoding, the above SAT encoding is more costly in many cases. This is due to the fact that (usually) the SAT encoding introduces more variables than the difference logic encoding, but the former is still of relatively low polynomial size. More precisely, the size of the SAT encoding can be characterized as follows.

Proposition 64 *Given a Boolean equation system $\mathcal{E}$ with dependency graph $G_\mathcal{E} = (V, E, \ell)$ and alternation hierarchy $\mathcal{H}$ the size of the propositional logic formula Φ_G (i.e. Φ_G with only Boolean variables) is $O(|E| \cdot \lceil \frac{p_{max}}{2} \rceil \cdot \lceil \log(m_{max} + 1) \rceil)$ where $p_{max} = \max\{\mathcal{H}(v) \mid v \in V\}$ and $m_{max} = \max\{|V_1|, |V_3|, \ldots, |V_{mop(G)}|\}$.*

Typically, propositional satisfiability solvers assume that an input propositional formula is in conjunctive normal form. This assumption does not raise a problem because the worst case bound in Proposition 64 can be obtained also for a CNF propositional formula. For instance, as suggested in the well-known Tseitin transformation [98] for propositional logic, the equivalent CNF formula can be obtained by introducing additional Boolean variables in the CNF conversion process.

Instead of using the Tseitin transformation, our implementation of the propositional encoding converts directly a given Boolean equation system instance to a slightly optimized CNF propositional formula in the following way. First, starting with the difference logic formula the subformulas $\Phi_\exists$, $\Phi_\forall$ and Φ_V can be trivially transformed to CNF without introducing any new variables. Then, the equivalence

$$\begin{aligned}&(\phi \rightarrow (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k)) \equiv\\ &((\phi \rightarrow \psi_1) \wedge (\phi \rightarrow \psi_2) \wedge \ldots \wedge (\phi \rightarrow \psi_k))\end{aligned}$$

is used to turn the formula Φ_A into a conjunction of constraints of the forms

$$(T_{v,w} \rightarrow (x > y)) \text{ and } (T_{v,w} \rightarrow (x \geq y))$$

where x and y are integer variables.

To encode the conjuncts of the form

$$(T_{v,w} \rightarrow (x > y))$$

where the integer variables x and y are to be encoded in the Boolean domain in $n+1$ bits, our implementation introduces new Boolean variables x^i and y^i for each $0 \leq i \leq n$ (here x^n and y^n are the most significant bits). In addition, the implementation introduces new variables $gt(x,y)^i$ for each $0 \leq i \leq n$. Our implementation encodes the given conjuncts as the following CNF clauses:

$$\begin{aligned}
&(\neg gt(x,y)^0 \vee \neg y^0) \wedge (\neg gt(x,y)^0 \vee x^0) \wedge \\
&\bigwedge_{0<m\leq n} ((\neg gt(x,y)^m \vee \neg y^m \vee x^m) \wedge \\
&(\neg gt(x,y)^m \vee \neg y^m \vee gt(x,y)^{m-1}) \wedge \\
&(\neg gt(x,y)^m \vee y^m \vee x^m \vee gt(x,y)^{m-1})) \wedge \\
&(\neg T_{v,w} \vee gt(x,y)^n)
\end{aligned}$$

In the same way, to encode conjuncts of the form

$$(T_{v,w} \rightarrow (x \geq y))$$

the implementation introduces variables x^i and y^i for each $0 \leq i \leq n$. In addition, the implementation introduces new variables $ge(x,y)^i$ for each $0 \leq i \leq n$. Then, the implementation encodes the given conjunct as the following CNF clauses:

$$\begin{aligned}
&(\neg ge(x,y)^0 \vee \neg y^0 \vee x^0) \wedge \\
&\bigwedge_{0<m\leq n} ((\neg ge(x,y)^m \vee \neg y^m \vee x^m) \wedge \\
&(\neg ge(x,y)^m \vee \neg y^m \vee ge(x,y)^{m-1}) \wedge \\
&(\neg ge(x,y)^m \vee y^m \vee x^m \vee ge(x,y)^{m-1})) \wedge \\
&(\neg T_{v,w} \vee ge(x,y)^n)
\end{aligned}$$

Notice that the resulting formulas have only one new variable for each bit in the numbers and only three clauses for each bit when $n > 0$, leading to quite compact encodings.

7.6 Discussion

In the previous subsections, we have seen how to reduce the problem of solving general form Boolean equation systems to difference logic satisfiability and propositional SAT. As mentioned before, these reductions are particularly useful because of the availability of several satisfiability checkers to solve the generated difference logic and propositional satisfiability problem instances efficiently in practice. The rate of the improvement in the performance of the state-of-the-art satisfiability checkers has been very high in recent years. It is expected that this makes the reductions even more useful in the future.

We have implemented the translations from general form Boolean equation systems to difference logic and SAT presented in the previous subsections. In Section 8.2, we report experimental results on several classes of benchmark problems.

Notice that the encodings given in the previous subsections can only find a *local* solution to a general Boolean equation system as defined in Definition 9. Therefore, in order to find a global solution to general blocks of a Boolean equation system with these encodings one may need to solve each variable separately. Clearly, this might be a drawback when incorporating the techniques into a general solution procedure defined in Section 3.

For further work, of special interest are thus the extensions of the encodings to find global solutions defined in Definition 11. In fact, it is not difficult to extend the reductions to a *global* Boolean equation system solver – an encoding that computes for each node of the dependency graph the solution to the variable in question.

The straightforward extension to find global solutions roughly doubles the number of variables and clauses in the formulas. For instance, every node of the dependency graph can be equipped with two data structures: a μ-annotation and a dually defined ν-annotation. The formula then asserts that, either the μ- or the ν-annotation needs to be locally successful.

The duality of the problem can also be exploited to optimize the translation, in the same way as in the logic programming encoding. Namely, one can choose between guessing either a conjunctive or a disjunctive Boolean equation system, always giving the encoding for the case where the search space is minimized.

The presented reductions from Boolean equation systems to satisfiability problems are important because they enable integration of these kinds of fixpoint equation checkers to other SAT based verification technologies. For example, the results in the previous subsections are directly applicable to the symbolic model checking of μ-calculus [52], and to the symbolic model checking of related formalisms like alternating automata [46].

8 EXPERIMENTS ON SOLVING BOOLEAN EQUATION SYSTEMS

In this section, we evaluate experimentally the solution techniques proposed in the previous sections. First, we study the performance of solvers for conjunctive and disjunctive Boolean equation systems for μ-calculus model checking problems. The solvers are implementations of the solution algorithms described in Section 5. In the remaining case studies we use SMODELS [90] answer set programming system, difference logic solver DPLL(T) [79] and SAT solver ZCHAFF [77] to solve general form Boolean equation systems. We have compared the various encodings presented in Section 6 and Section 7 on difficult instances of general Boolean equation systems.

The problem instances and generators used in the thesis are available via the internet, see `http://www.tcs.hut.fi/Software/benchmarks/bes/bes-benchmarks.tar.gz`.

8.1 Tests for Conjunctive and Disjunctive Blocks

Next, we give experimental results using implementations of the solution algorithms for alternating, conjunctive and disjunctive form Boolean equation systems presented and discussed in Section 5. We have implemented the algorithms in the C programming language [58]. We have evaluated the performance of the algorithms using verification problem benchmarks.

Model Checking Regular Models

As our first benchmarks we use two sets of μ-calculus model checking problems from [67] and [91], converted to Boolean equation systems. The verification problems consist of model checking μ-calculus formulas of alternation depth 2, on a sequence of regular labelled transition systems M_k of increasing size (see Figure 11).

Suppose we want to check, at initial state s of process M_k, the property that transitions labelled b occur infinitely often along every infinite path of the process. This is expressed with the μ-calculus formula:

$$\phi_1 \equiv \nu X.\mu Y.([b]X \wedge [-b]Y) \tag{23}$$

The property is false at state s and thus the corresponding Boolean equation systems have solution 0.

In second series of examples, we check the property that there is an execution in M_k starting from state s, where action a occurs infinitely often. This is expressed with the μ-calculus formula

$$\phi_2 \equiv \nu X.\mu Y.(\langle a\rangle X \vee \langle -a\rangle Y) \tag{24}$$

which is true at initial state s of the process M_k.

The problems of determining whether the system M_k satisfies the specifications ϕ_1 and ϕ_2 can be directly encoded as problems of solving the corresponding alternating Boolean equation systems, which are in conjunctive and disjunctive forms.

As an illustration we explain here the transformation of the first formula ϕ_1 using the standard translation [1, 4, 70] from μ-calculus to Boolean equation

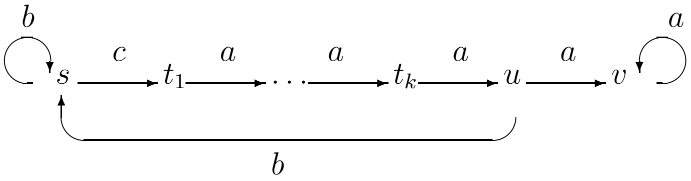


Figure 11: Process M_k from [67, 91] for model checking the properties ϕ_1 and ϕ_2.

systems (see Figure 3). If we consider a labelled transition system $M_k = (S, A, \longrightarrow)$ in Figure 11 then the Boolean equation system looks like:

$$\left.\begin{aligned} \nu\, x_s &= y_s \\ \mu\, y_s &= \bigwedge_{s' \in \nabla(a,s)} x_{s'} \wedge \bigwedge_{s' \in \nabla(\neg a,s)} y_{s'} \end{aligned}\right\} \text{for all } s \in S.$$

Here, $\nabla(a, s) := \{s' | s \xrightarrow{a} s'\}$ and $\nabla(\neg a, s) := \{s' | s \xrightarrow{b} s' \text{ and } b \neq a\}$.

We report the times for the solvers to find the local solutions corresponding to the local model checking problems of the formulas at state s. The times in this section are the times for the solvers to find the local solutions measured as system time, on a 1.0Ghz AMD Athlon running Linux (i.e. the times for the solvers to read the Boolean equation systems from disk and build the internal data structures are excluded).

The experimental results are given in Table 1. The columns are explained below:

- Problem:
  - the process M_k, with $k + 3$ states
  - ϕ_1 the formula (23) to be checked
  - ϕ_2 the formula (24) to be checked
- n: the number of equations in the Boolean equation system corresponding to the model checking problem
- Hierarchical clustering: the time in seconds to find the local solution with the algorithm based on hierarchical clustering
- Depth-first search: the time in seconds to find the local solution with the algorithm based on depth-first search

As shown in the performance measures, there is no clear winner. Indeed, there is no significant difference between the performance of the algorithms on these specific benchmarks. But, one must be careful in drawing general results from these experiments, because of the following reasons.

First of all, the benchmarks from [67, 91] have a quite simple structure. For instance, [84] has gathered a large collection of state spaces derived from realistic system models and has performed an extensive empirical study of

Problem		n	Hierarchical clustering	Depth-first search
M_{100000}	ϕ_1	200 006	0.0	0.0
	ϕ_2	200 006	0.1	0.6
M_{200000}	ϕ_1	400 006	0.0	0.1
	ϕ_2	400 006	0.3	0.7
M_{300000}	ϕ_1	600 006	0.3	0.2
	ϕ_2	600 006	0.4	0.8
M_{400000}	ϕ_1	800 006	0.4	0.3
	ϕ_2	800 006	0.5	0.9
M_{500000}	ϕ_1	1 000 006	0.5	0.3
	ϕ_2	1 000 006	0.6	0.9

Table 1: Comparison of the algorithms for checking properties ϕ_1 and ϕ_2 for benchmarks from [67, 91].

their structural properties. From the results in [84] we know that state spaces of realistic systems have several typical properties which differ significantly from regular graphs like the transition system depicted in Figure 11 from [67, 91].

Second, all the examples examined above are quite easy to solve in fragments of a second with both the hierarchical clustering and the depth-first search based algorithms. The small differences in performance might be meaningless because they can be influenced by various factors which, in fact, may have nothing to do with the algorithms.

For these reasons, a more involved practical evaluation is desirable here. Next, we provide experimental results on Boolean equation system benchmarks from more realistic applications in the domain of protocol verification. A comparison on such harder problems reveals that the hierarchical clustering based algorithm is faster than the depth first-search based algorithm, as suggested by the worst-case analysis in Section 5.

Model Checking Sliding Window Protocols

We have done experiments with models of sliding window protocols described in [94]. To investigate and compare the performance of the algorithms, we have studied three variants of the protocol with different behaviours:

- *Variation 1*: This is an unidirectional version of the protocol where a sender receives data through a channel and passes it to a receiver. There are 2 data elements, window size is 2, and buffer size is 4 at both receiving and sending side.

- *Variation 2*: This is a bidirectional, one bit sliding window protocol where, in addition to the feature of variation 1, also the receiver receives data via a channel and passes it to the sender. There is 1 data element, window size is 1 and buffer size is 2 at both receiving and sending side.

- *Variation 3*: As variation 2, this is a bidirectional version with buffer size 2, window size 1, and 1 data element. However, *piggy backing* is used to guarantee a better bandwidth, i.e. acknowledgements, which are sent between the sender and the receiver, are appended to data elements.

Each of the protocol variations was modelled with the μCRL tool set [10] and its state space, combined with liveness and fairness related formulas, was converted to Boolean equation systems for input by our implementations of the solution algorithms. Again, in the conversion we used the translation from μ-calculus to Boolean equation systems as described in Section 2.4.

The results of our experiments are shown in Table 2. The first column contains the names of the checked formulas which are given explicitly below the table. The column marked "Equation system" gives the number of left hand side variables and the size of the corresponding Boolean equation system. The columns marked "Hierarchical clustering" and "Depth-first search" give the execution times in seconds for the algorithms to solve the Boolean equation systems measured as cpu time. The reported times are the average of three runs on a 1.0Ghz AMD Athlon running Linux with sufficient main memory.

The checked μ-calculus formulas can be explained as follows. Formula A states unconditional fairness for the reception of data by requiring that reception of data happens infinitely often along every infinite execution. Formula B is related to counting silent actions and states the property that the protocol does only finitely many τ-actions, no matter what else it does. Formula C is a liveness property which states that whenever a message is sent then eventually it is received. Formula D expresses a strong fairness property that delivery of data via send action is fairly treated. The last formula is a more involved property which expresses liveness under fairness. More precisely, property E says that, for any execution, if the sender is enabled infinitely often and the receiver is enabled infinitely often, then whenever a message is sent eventually it is received.

In almost all cases the time consumption by the hierarchical clustering based algorithm was considerably less than by the algorithm based on depth-first search. In only three cases, namely variations 1-3 C, the time consumed by the hierarchical clustering algorithm was slightly more than that by the latter algorithm. For instance, in variation 3 the hierarchical clustering algorithm spent less than 3 seconds to solve all formulas A-E while the corresponding total running time for the depth-first search based algorithm was around 8 minutes. Based on these computational results we may draw the conclusion that the algorithm based on hierarchical clustering substantially outperforms the one based on the depth-first search.

We were not able to conduct a comparative study with other approaches because our formulas have non-zero alternation depths. All other publicly available tools are for alternation-free Boolean equation systems (e.g. [75]).

Heuristic Issues

As indicated by the performance measures in the previous section, there exist some examples where the hierarchical clustering based algorithm fares worse than the one based on depth-first search. This suggests to use heuristics to guide the former algorithm to find solutions more quickly.

Table 2: Comparison of the algorithms for checking property ϕ for different versions of the sliding window protocol.

Variation 1: 44540 states, 183344 transitions			
ϕ	Equation system	Hierarchical clustering	Depth-first search
A	54265 193069	0.10	29.69
B	87464 226268	0.32	244.90
C	76348 325660	0.70	0.01
D	69476 269152	1.66	86.29
E	115716 507376	0.51	1.44

Variation 2: 17040 states, 79472 transitions			
ϕ	Equation system	Hierarchical clustering	Depth-first search
A	19185 81617	0.08	69.35
B	33904 96336	0.10	44.88
C	30376 146344	0.21	0.00
D	36832 137892	0.79	46.48
E	48600 232648	1.39	4.37

Variation 3: 23728 states, 112960 transitions			
ϕ	Equation system	Hierarchical clustering	Depth-first search
A	26337 115569	0.06	14.51
B	47152 136384	0.07	62.81
C	42808 208816	0.11	0.00
D	50560 194356	0.28	407.03
E	70072 338364	2.38	4.37

$r(x) \equiv$ receive data x
$s(x) \equiv$ send data x

A $\equiv \nu X.\mu Y.([r(1)]X \wedge [\neg r(1)]Y)$
B $\equiv \mu X.\nu Y.([\tau]X \wedge [\neg\tau]Y)$
C $\equiv \nu Z.([s(1)](\mu Y.\langle - \rangle\top \wedge [\neg r(1)]Y) \wedge [-]Z)$
D $\equiv \nu X.\mu Y.\nu Z.([s(1)]X \wedge (\langle s(1)\rangle\top \Rightarrow [\neg s(1)]Y) \wedge [\neg s(1)]Z)$
E $\equiv \nu X.([s(1)]\psi \wedge [-]X)$ where ψ is given below
$\psi \equiv \mu Y.\nu Z.(([s(1)]\bot \vee [\neg r(1)](\nu V.([r(1)]\bot \vee Y) \wedge [\neg r(1)]V)) \wedge [\neg r(1)]Z)$

Table 3: Effect of heuristics on the *MinNuLoop* algorithm.

Equation system		Heuristic			
Variation	ϕ	None	H1	H2	H3
1	A	0.10	0.10	0.11	0.09
1	B	0.32	0.17	0.16	0.32
1	C	0.70	0.22	0.23	0.78
1	D	1.66	0.18	0.19	1.69
1	E	0.51	0.36	0.36	0.52
2	A	0.08	0.05	0.05	0.08
2	B	0.20	0.07	0.07	0.09
2	C	0.21	0.09	0.10	0.21
2	D	0.79	0.10	0.10	0.79
2	E	1.39	0.17	0.16	1.39
3	A	0.06	0.06	0.06	0.06
3	B	0.07	0.07	0.07	0.07
3	C	0.11	0.11	0.10	0.10
3	D	0.28	0.13	0.13	0.29
3	E	2.38	0.20	0.20	2.35
Total cpu time		8.86	2.08	2.09	8.83

Recall the description of the *MinNuLoop* algorithm given in Section 5.3. In steps 5 and 6 of *MinNuLoop*, two distinct recursive calls are done. It turns out that the order of these recursive calls does not affect the correctness of the algorithm. Steps 5 and 6 might as well be executed in any possible order as long as they are both executed after step 4. But, the differences in the execution order certainly may be reflected in the performance of the algorithm. To investigate the impact of changing the execution order of the recursive calls, various heuristics were used.

The results are shown in Table 3. Only the performance for the *MinNuLoop* algorithm is described in the table. The meaning of the first two columns is the same as in Table 2. The remaining columns contain the measures for the heuristics; the number indicates the cpu time in seconds to find the solution. Here, the column “None” agrees with the column “Hierarchical clustering” in Table 2. Finally, the last row describes the total cpu time in seconds to solve all the problems.

The heuristics that we have investigated include:

H1 Reversed execution order of the recursive calls in steps 5 and 6; i.e. execute step 6 first, and then execute step 5.

H2 Selects those recursive calls that lead to smaller graphs first; i.e. if the graph G' constructed in step 2 has less edges than the graph G'' constructed in step 3, execute step 5 first, and then execute step 6. Otherwise, execute step 6 first, and then execute step 5.

H3 Selects those recursive calls that lead to larger graphs first; i.e. if the graph G' constructed in step 2 has more edges than the graph G'' con-

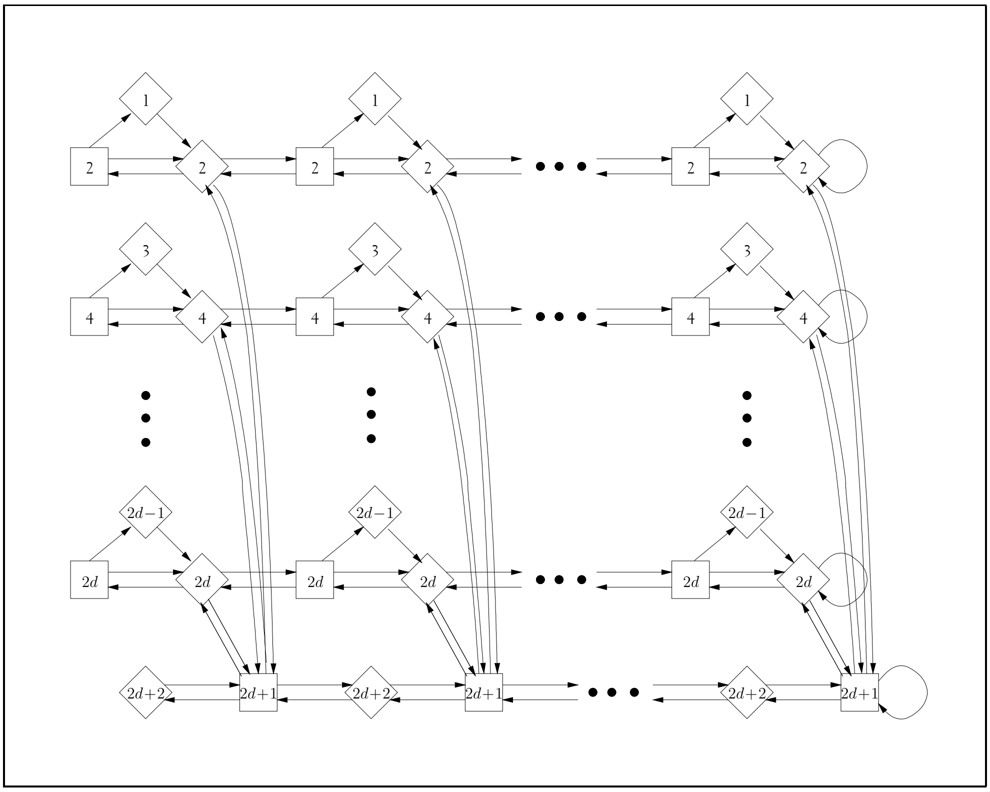


Figure 12: The Jurdziński graphs $J_{d,w}$.

structed in step 3, execute step 5 first, and then execute step 6. Otherwise, execute step 6 first, and then execute step 5.

As the table shows, heuristics H1 and H2 performed up to a factor 10 better than using no heuristic at all. The performance of heuristic H3, which selects those recursive calls that lead to larger graphs first, was the worst.

One must notice that the differences in the performance are very small and, therefore, they may be influenced by other factors too. Of course, some heuristics might work well on some Boolean equation systems, and poorly for others. But, the results indicate that changing the execution order of the recursive calls has a clear impact on the solution times.

8.2 Tests for General Form Blocks

In the following subsections, we give experimental results using the encodings presented in Section 6 and Section 7. We have implemented all these encodings in the C programming language [58]. In order to evaluate and compare the techniques we have conducted experimental research on solving general Boolean equation systems using SMODELS [90], DPLL(T) [79] and ZCHAFF [77] tools.

Jurdziński Graphs

As the first set of benchmarks in general class we use the family of general form Boolean equation systems derived from *Jurdziński graphs* $J_{d,w}$ parameterized by the depth $d \in \mathbb{N}$ and the width $w \in \mathbb{N}$. Jurdziński graphs are an example of parity games on which the small progress measure algorithm

w	parity game $J_{1,w}$			parity game $J_{5,w}$		
	DPLL(T)	zChaff	Smodels	DPLL(T)	zChaff	Smodels
10	0.1	0.0	0.0	0.1	0.0	0.0
20	0.1	0.0	0.1	0.2	0.1	0.2
30	0.1	0.0	0.2	0.3	0.2	0.5
40	0.1	0.0	0.5	0.5	0.3	1.1
50	0.1	0.0	1.0	0.9	0.4	1.9
60	0.1	0.0	3.1	1.3	0.4	3.5
70	0.2	0.1	5.1	1.8	0.6	5.9
80	0.2	0.1	15.7	2.6	0.6	9.9
90	0.2	0.1	16.0	3.6	0.8	15.9
100	0.2	0.1	44.6	6.0	0.8	24.5
110	0.2	0.1	66.2	7.6	0.9	35.9
120	0.2	0.1	68.8	9.2	1.0	51.4
130	0.3	0.1	133.9	11.2	1.3	71.2
140	0.4	0.2	185.6	13.6	1.4	93.4
150	0.4	0.2	199.5	15.8	1.5	125.5
160	0.4	0.2	325.9	18.8	1.5	164.0

Figure 13: The running times on the Jurdziński graphs $J_{1,w}$ and $J_{5,w}$.

exhibits exponential running time [54].

The parity game $J_{d,w}$ can be represented as a rectangle of $2d + 1$ rows and $2w$ columns as depicted in Figure 12. Nodes in $V_\exists$ are represented in a diamond shape, nodes in $V_\forall$ as boxes, and the numbers inside are the corresponding priorities. The maximal priority occurring in $J_{d,w}$ is $2d + 2$. For each Jurdziński graph $J_{d,w}$, the corresponding Boolean equation system is easily obtained through the translation given in Section 2.5.

The Corollary below follows directly from Theorem 12 in [54].

Corollary 65 *Given a Jurdziński graph $J_{d,w}$, the running time of the progress measure algorithm on $J_{d,w}$ is exponential in d.*

Although Jurdziński graphs are difficult to solve with the small progress measure algorithm it is easy to see that the player $\exists$ has a winning strategy from every node in the first $2d$ rows whereas the player $\forall$ has a winning strategy from every node in row $2d + 1$. For instance, these strategies are simply moving to the right end of each row.

In the following we will always choose as the starting node the leftmost node in the second row. Hence, the resulting formulas under the reductions are always satisfiable, and there is always at least one stable model for the corresponding logic program. The reason for this choice is that we observed the instances obtained by setting the starting node to be the leftmost node on the last row (i.e. the unsatisfiable formulas and instances without a stable model) were easier for all the solvers.

We describe now the experimental results on solving Boolean equation systems based on Jurdziński graphs through the reductions given in Section 7. The problem instances are translated into difference logic and SAT by our implementations of the translations which also perform CNF conversion in both cases. In addition, we compare these to the answer set programming approach presented in Section 6.

More precisely, we compare the three methods on four series of graphs. In each test series we measure the times used for the solvers DPLL(T) and

d	parity game $J_{d,5}$			parity game $J_{d,10}$		
	DPLL(T)	zChaff	Smodels	DPLL(T)	zChaff	Smodels
5	0.1	0.0	0.0	0.1	0.0	0.0
10	0.2	0.1	0.0	0.3	0.1	0.1
15	0.3	0.1	0.1	0.8	0.3	0.3
20	0.5	0.2	0.1	2.4	0.6	0.5
25	1.1	0.3	0.2	6.0	0.8	0.7
30	2.4	0.4	0.3	12.3	1.2	1.0
35	5.1	0.6	0.3	22.0	1.6	1.3
40	8.9	0.8	0.4	39.8	2.1	1.7
45	14.3	1.0	0.5	61.1	2.6	2.1
50	21.9	1.2	0.7	93.3	3.2	2.6
55	29.1	1.5	0.8	122.7	3.9	3.2
60	40.3	1.7	0.9	183.3	4.7	3.8
65	59.0	2.0	1.1	51.0	5.4	4.4
70	75.0	2.4	1.2	336.3	6.2	5.1
75	105.8	2.7	1.4	427.7	7.2	5.9
80	134.8	3.1	1.6	505.4	3.8	6.8

Figure 14: The running times on the Jurdziński graphs $J_{d,5}$ and $J_{d,10}$.

zChaff to check the satisfiability of the corresponding formulas, and the time used for Smodels to determine whether or not there is a stable model.

The first series considers the Boolean equation systems based on graphs $J_{d,w}$ with fixed depth $d = 1$ and varying width w. The second series again varies w, but this time with a fixed depth of $d = 5$. The results on benchmarks in classes $J_{1,w}$ and $J_{5,w}$ are shown in Figure 13. Notice that in these two series of examples with fixed d several algorithms which are polynomial in w exist in the literature.

The third test series considers the Boolean equation systems based on graphs $J_{d,w}$ with fixed width $w = 5$ and varying depth d. The fourth series again varies d, but this time with a greater fixed width of $w = 10$. Figure 14 shows the total run times for solving the benchmarks in classes $J_{d,5}$ and $J_{d,10}$. Notice that the small progress measure algorithm [54] shows an exponential time performance on these kinds of examples with increasing depths.

From Figure 13 we can observe that when d is small, both the difference logic and SAT encodings are ahead of the logic programming approach. By investigating the logs of the solvers we are able to analyze the situation further. The explanation for the significantly larger running times in the Smodels column seems to come from a larger encoding of the problem. Namely, the size of the translation used is $O(|E| \cdot |V|)$ irregardless of the number of priorities in $J_{d,w}$, and thus starts to grow quadratically in w, while e.g. the size of the difference logic encoding only grows only linearly in w (as the number of priorities in $J_{d,w}$ is fixed). In addition, the search speed of Smodels is significantly slower than that of highly optimized SAT solvers as zChaff. This is likely due to the fact the Smodels uses a relatively expensive decision heuristic to avoid wrong decisions rather than counting on fast search speed, learning, and restarts as the modern SAT solver zChaff does. In addition, from Figure 13 we can observe that when d is increased from 1 to 5 the running times of the DPLL(T) start to grow, and thus the zChaff outperforms the DPLL(T).

From Figure 14 we can observe a completely different situation. The number of nodes at each row of $J_{d,w}$ is bounded by a small number. There-

n	#sat/#unsat	DPLL(T)	zCHAFF	SMODELS
		min/median/max	min/median/max	min/median/max
100	16 / 15	0.1 / 0.2 / 0.2	0.0 / 0.1 / 0.1	0.0 / 0.0 / 7.2
200	21 / 10	0.2 / 0.8 / 1.1	0.1 / 0.2 / 0.5	0.1 / 0.6 / >1000
300	19 / 12	0.4 / 4.9 / 8.1	0.3 / 0.6 / 4.7	0.1 / 4.5 / >1000
400	15 / 16	0.7 / 11.3 / 23.5	0.6 / 1.1 / 69.1	0.1 / 13.7 / >1000
500	18 / 13	1.1 / 32.2 / 64.6	1.1 / 1.7 / 22.1	0.2 / 37.1 / >1000
600	14 / 17	1.7 / 68.6 / 370.2	1.4 / 3.6 / >1000	0.4 / 147.1 / >1000
700	15 / 16	2.6 / 133.1 / 212.8	2.2 / 3.8 / >1000	0.5 / 230.6 / >1000
800	15 / 16	3.7 / 169.1 / 356.5	2.5 / 7.8 / >1000	0.6 / 378.4 / >1000

Figure 15: The running times on random Boolean equation systems with maximal alternation hierachy level $m = n$.

fore, only a small fixed number of variables is needed to encode the numbers of a μ-annotation. Thus, in the case with fixed w the sizes of all three encodings grow quadratically in d. Therefore, the encoding size drawback that existed in the first two series for SMODELS does not appear anymore when d is increased.

In the two series of examples with fixed w the difference logic encoding is a constant factor smaller than the propositional logic encoding, but zCHAFF significantly outperforms DPLL(T) in running time. An explanation for this might be that the highly optimized algorithms and data structures inside zCHAFF enable it to perform search at admirable speed. Furthermore, it seems that the good performance of SMODELS in the examples with fixed w can be largely explained by the significantly smaller search space covered when compared to the DPLL(T).

Notice that, by Corollary 65, the running time of the small progress measure algorithm [54] on the two series of graphs with fixed w is exponential in d. Therefore, the running times for all three solvers on these examples are quite competitive compared to the algorithmic approach [54].

By investigating the solver logs we find out that the SMODELS solver did not backtrack on any of these four series of examples. It is not yet known whether it is a lucky co-incidence or the heuristics of SMODELS always manage to do so on this family of problem instances. If the latter case could be proved for all values of d and w, then the SMODELS based approach resulted in a guaranteed polynomial time algorithm for examples based on Jurdziński graphs. Also, the DPLL(T) and zCHAFF solvers make surprisingly few wrong branching decisions while searching for a satisfying assignment in these four series of examples.

Finally, to evaluate the performance of a slightly older SAT solver, we have tried SATZ [65] but we have found it to hit the one hour timeout limit even on the smallest Jurdziński graph instances. Therefore, if one wishes to adopt the solution technique based on the propositional encoding, then it seems that a modern SAT solver (as zCHAFF) is actually needed to solve the problem instances in reasonable times.

n	#sat/#unsat	DPLL(T)	zChaff	Smodels
		min/median/max	min/median/max	min/median/max
100	20 / 11	0.1 / 0.1 / 0.1	0.0 / 0.0 / 0.0	0.0 / 0.0 / 0.7
200	23 / 8	0.1 / 0.1 / 0.1	0.0 / 0.1 / 0.1	0.0 / 0.8 / 265.4
300	23 / 8	0.1 / 0.1 / 0.2	0.1 / 0.1 / 0.3	0.1 / 4.8 / >1000
400	21 / 10	0.1 / 0.2 / 0.2	0.1 / 0.2 / 14.4	0.1 / 13.7 / >1000
500	21 / 10	0.1 / 0.2 / 0.3	0.1 / 0.3 / 1.1	0.3 / 40.9 / >1000
600	21 / 10	0.2 / 0.3 / 0.4	0.2 / 0.4 / 1.7	0.3 / 121.3 / >1000
700	17 / 14	0.2 / 0.4 / 0.7	0.3 / 0.6 / >1000	0.4 / 209.9 / >1000
800	20 / 11	0.2 / 0.5 / 0.7	0.3 / 1.3 / 12.8	0.6 / 362.5 / >1000

Figure 16: The running times on random Boolean equation systems with maximal alternation hierarchy level $m = \lceil \sqrt{n} \rceil$.

Tests for Randomly Generated Boolean Equation Systems

Another set of benchmarks we have used is a set of randomly generated Boolean equation systems. These are generated by the following simple algorithm. For a parameter value n, start generating a dependency graph with nodes $V = \{0, 1, \dots, n-1\}$ and generate exactly two outgoing edges for each node. Then, discard all nodes that are not reachable from the smallest node 0. We have found out experimentally that roughly 80% of all nodes are reachable from node 0 on the average. For all the remaining reachable nodes, we pick a connective (of the right-hand side positive Boolean formula) for each node with equal probabilities. Another parameter value is the maximal alternation hierachy level m. We pick the fixpoint sign of each node uniformly at random so that the maximal alternation hierarchy level of the resulting Boolean equation system is m.

Figure 15 shows the running time results for increasing values of n and fixing $m = n$. Figure 16 shows running times with the parameter $m = \lceil \sqrt{n} \rceil$. In both cases, we report minimum, median, and maximum running times on 31 problem instances for each value of n, using a 1000 second timeout limit. In addition, we report the number of satisfiable/unsatisfiable instances for each parameter value.

As can be observed from both Figure 15 and Figure 16, these results are not too conclusive. The main reason for this is that the distances between the minimum and maximum running times are quite large for all solvers. In particular, when $m = n$, the running times for all the solvers seem to vary by several orders of magnitudes on the same sized problem instances.

Unlike in the case of the Jurdziński graph benchmarks, we have not observed here any differences in the difficulty of satisfiable vs. unsatisfiable random problem instances.

The main observation on these random benchamarks is that the Smodels times out on several of the problem instances while zChaff times out only on four instances. The DPLL(T) times out on none of the random instances, and performs well on the benchmarks with $m = \lceil \sqrt{n} \rceil$.

By investigating the solver logs we can observe that, unlike in examples based on Jurdziński graphs, in these sets of random benchmarks Smodels does backtrack. Therefore, in general it is unlikely that Smodels would solve arbitrary Boolean equation systems in polynomial time. Perhaps, the frequent SMODELS timeouts make the logic programming approach less robust than the solution techniques based on satisfiability encodings, at least on these random examples.

9 CONCLUSION

This thesis has presented new methods to solve Boolean equation systems. The work has presented a framework which allows for considerably optimizing the solution techniques by taking advantage of specific properties and features of Boolean equation systems. We have developed techniques to solve various classes of Boolean equation systems, and have demonstrated how these different methods can be combined into a single framework.

We have demonstrated that the verification of many formulas in the μ-calculus with alternating fixed points amounts to solving Boolean equation systems which consist of conjunctive and disjunctive blocks. Subsequently, we have provided new algorithms to solve these kinds of blocks. The hierarchical clustering algorithm in Section 5.3 has better estimation of its worst-case time complexity than the depth-first search based algorithm in Section 5.2. Practical evaluation on protocol verification benchmarks shows that this also leads to practical improvements: the former algorithm is often able to find solutions more quickly than the latter, and additional reduction in time consumption can be gained by using suitable recursion orders to guide the search of the hierachical clustering algorithm. We believe that this makes the verification of a large class of μ-calculus formulas with alternating fixed points tractable, even for large, practical systems.

The sub-quadratic algorithm that we obtain in Section 5.3 for conjunctive and disjunctive form Boolean equation systems is quite efficient in practice, but it still contains an unpleasant logarithmic factor. Despite several efforts, we have been unable to eliminate this factor. It is an interesting further question whether there exists a linear-time algorithm for solving conjunctive and disjunctive Boolean equation systems.

We have presented an answer set programming based technique for computing the solutions to general Boolean equation systems. We have devised a mapping from general Boolean equation systems to normal logic programs. The translation is such that the solution of a given variable of an equation system can be determined by the existence of a stable model of the corresponding logic program.

The experimental results indicate that the stable model computation with the SMODELS system is quite a competitive technique to solve general Boolean equation systems based on Jurdziński graphs with a large alternation depth. Unfortunately, the further experiments on random graphs show that the answer set programming approach is not as competitive as the results on Jurdziński graphs suggest.

In any case, the answer set programming approach provides the basis for verifying μ-calculus formulas with alternating fixed points using answer set programming techniques, and the approach proves itself as a baseline for further investigation.

We have shown how to reduce the problem of solving general Boolean equation systems into difference logic satisfiability and into propositional SAT. The experimental results indicate that both of these encodings are quite competitive in cases where the underlying Boolean equation system has large alternation depths. In particular, the reduction of solving general Boolean equation systems into propositional SAT is mainly interesting because of the availability of practically efficient SAT checkers to solve the generated instances.

The alternation of fixpoint operators gives more expressive power in the μ-calculus, but all known model checking algorithms are exponential in the alternation depth. Consequently, our satisfiability based approaches are expected to be useful in the verification tasks where there is a need of formulas with great expressive power.

As further work it would be interesting to study how the satisfiability based approaches would work as subroutines in a framework of parameterized Boolean equation systems presented in [43, 45].

If one needs to model check models which are too big to store in a computer's memory, then it may not be useful to apply the methods presented in this thesis as they are. Indeed, instead of using explicit representations of models in such a situation it may be wise to resort to symbolic methods mentioned in Section 1. As future work it would be interesting to study an extension of the techniques presented in Section 7 to a symbolic setting where the models are not represented explicitly but in a symbolic way as is done in symbolic model checking [9]. This kind of an extension might allow to handle even larger state spaces than is possible with the techniques in Section 7.

References

[1] H.R. Andersen. Model checking and Boolean graphs. *Theoretical Computer Science*, 126:3–30, 1994.

[2] H.R. Andersen, B. Vergauwen. Efficient Checking of Behavioural Relations and Modal Assertions using Fixed-Point Inversion. In *Proceedings of Conference on Computer Aided Verification*, Lecture Notes on Computer Science 939, pages 142–154, Springer-Verlag, 1995.

[3] A. Arnold and P. Crubille. A linear algorithm to solve fixed-point equations on transition systems *Information Processing Letters*, 29: 57–66, 1988.

[4] A. Arnold and D. Niwinski. Rudiments of μ-calculus. *Studies in logic and the foundations of mathematics*, Volume 146, Elsevier, 2001.

[5] A. Arnold, A. Vincent and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, 303(1): 7–34, 2003.

[6] O. Bernholtz, M. Vardi and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, Lecture Notes on Computer Science 818, pages 142–155, Springer-Verlag, 1994.

[7] D. Berwanger and E. Grädel. Fixed-point logics and solitaire games. *Theory of Computing Systems*, 37: 675–694, 2004.

[8] G. Bhat and R. Cleaveland. Efficient local model-checking for fragments of the modal μ-calculus. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1055, pages 107–126, Springer-Verlag, 1996.

[9] A. Biere, A. Cimatti, E. Clarke and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1579, pages 193–207, Springer-Verlag, 1999.

[10] S. Blom, W. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser and J. van de Pol. μCRL: a toolset for analysing algebraic specifications. In *Proceedings of Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2102, pages 250–254, Springer-Verlag, 2001.

[11] S. Blom and J. van de Pol. State space reduction by proving confluence. In *Proceedings of Conference on Computer Aided Verification*, Lecture Notes on Computer Science 2404, pages 596–609, Springer Verlag, 2002.

[12] J. Bradfield. The modal μ-calculus alternation hierarchy is strict. *Theoretical Computer Science*, 195:133–153, 1998.

[13] J. Bradfield and C. Stirling. Modal Logics and mu-Calculi: An introduction. Chapter 4 of *Handbook of Process Algebra*. J.A. Bergstra, A. Ponse and S.A. Smolka, editors. Elsevier, 2001.

[14] J. Burch, E.M. Clarke, K. McMillan, D. Dill, L. Hwang. Symbolic Model Checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–39, 1990.

[15] H. Björklund, S. Sandberg and S. Vorobyov. A Discrete Subexponential Algorithm for Parity Games. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS'2003)*, Lecture Notes in Computer Science 2607, pages 663–674, Springer-Verlag, 2003.

[16] E. Clarke, A. Biere, R. Raimi and Y. Zhu. Bounded Model Checking Using Satisfiability Solving *Formal Methods in System Design*, 19 (1): 7–34, 2001.

[17] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of Workshop on Logics of Programs*, Lecture Notes in Computer Science 131, pages 52–71, Springer-Verlag, 1981.

[18] E. Clarke, R. Enders, T. Filkorn and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2): 77 – 104, 1996.

[19] E. Clarke, O. Grumberg and D. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5): 1512–1542, ACM Press, 1994.

[20] E. Clarke, O. Grumberg and D. Peled. Model Checking. The MIT Press, 2000.

[21] R. Cleaveland. Tableau-based model checking in the propositional μ-calculus. *Acta Informatica*, 27(8):725–748, 1990.

[22] R. Cleaveland, M. Klein and B. Steffen. Faster model checking for the modal mu-calculus. In *Proceedings of the 4th International Workshop on Computer Aided Verification*, Lecture Notes in Computer Science 663, pages 410–422, Springer-Verlag, 1992.

[23] R. Cleaveland and B. Steffen. Computing Behavioural relations logically. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 510, pages 127–138, Springer-Verlag, 1991.

[24] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Conference Rec. 3rd Annual ACM Symposium on Theory of Computing STOC'71*, pages 151–158, ACM, 1971.

[25] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3): 275–288, 1992.

[26] G. Delzanno and A. Podelski. Model checking in CLP. In Proceedings of the Int. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1579, pages 223–239, Springer-Verlag, 1999.

[27] J. Dix, U. Furbach, and I. Niemelä. Nonmonotonic reasoning: Towards efficient calculi and implementations. In *Handbook of Automated Reasoning*, chapter 19, pages 1241–1354. Elsevier, 2001.

[28] D. Dolew, M. Klawe, and M. Rodeh. An $O(n \, log \, n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3): 245–260, 1982.

[29] W.F. Dowling and J.H. Gallier. Linear-Time Algorithm for Testing the Satisfiability of Propositional Horn Formulae. *J. Logic Programming*, 3:267–284, 1984.

[30] E. A. Emerson. Model checking and the μ-calculus. Chapter 6 of *Descriptive Complexity and Finite Models*, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, volume 31. N. Immerman and P. G. Kolaitis, editors, AMS, 1997.

[31] E. A. Emerson and C. Lei. Efficient model checking in the fragments of the propositional μ-calculus. In *Symposion on Logic in Computer Science*, pages 267–278, IEEE Computer Society Press, 1986.

[32] E. A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9(1/2): 105–131, 1996.

[33] E.A. Emerson, C. Jutla. Tree automata, Mu-Calculus and determinacy. In *Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 368–377, IEEE Computer Society Press, 1991.

[34] E.A. Emerson, C. Jutla and A.P. Sistla. On model checking for fragments of the μ-calculus. In *Proceedings of the Fifth International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 697, pages 385–396, Springer-Verlag, 1993.

[35] E.A. Emerson, C. Jutla, and A.P. Sistla. On model checking for the μ-calculus and its fragments. *Theoretical Computer Science*, 258:491–522, 2001.

[36] K. Etessami, T. Wilke, and R. Schuller. Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. *SIAM J. Comput.*, 34(5):1159–1175, 2005.

[37] C. Fecht and H. Seidl. An Even Faster Solver for General Systems of Equations. In *Proceedings of the Static Analysis Symposium*, Lecture Notes in Computer Science 1145, pages 189–204, Springer Verlag, 1996.

[38] L. Fredlund, J.F. Groote and H. Korver. Formal Verification of a Leader Election Protocol in Process Algebra. *Theoretical Computer Science*, 177: 459–486, 1997.

[39] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.

[40] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '1990)*, Lecture Notes in Computer Science 531, pages 176–185, Springer-Verlag, 1991.

[41] J.F. Groote and M. Keinänen. Solving Disjunctive/Conjunctive Boolean Equation Systems with Alternating Fixed Points. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 2988, pages 436 – 450, Springer Verlag, 2004.

[42] J.F. Groote and M. Keinänen. A Sub-quadratic Algorithm for Conjunctive and Disjunctive Boolean Equation Systems. In *Proceedings of 2nd International Colloquium on Theoretical Aspects of Computing (ICTAC'2005)*, Lecture Notes in Computer Science 3722, pages 545–558, Springer-Verlag, 2005.

[43] J.F. Groote and T. Willemse. A Checker for Modal Formulas for Processes with Data. In *Proceedings of Formal Methods for Components and Objects, Second International Symposium (FMCO'2003)*, Lecture Notes in Computer Science 3188, pages 223–239, Springer-Verlag, 2004.

[44] J.F. Groote and T. Willemse. Parameterised Boolean Equation Systems. *Theoretical Computer Science*, 343:332–369, 2005.

[45] J.F. Groote and T. Willemse. Parameterised Boolean Equation Systems. In *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR'2004)*, Lecture Notes in Computer Science 3170, pages 308–324, Springer-Verlag, 2004.

[46] K. Heljanko, T. Junttila, M. Keinänen, M. Lange and T. Latvala. Bounded Model Checking for Weak Alternating Büchi Automata. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'2006)*, Lecture Notes in Computer Science, Springer-Verlag, to appear.

[47] K. Heljanko, M. Keinänen, M. Lange and I. Niemelä. Solving parity games by a reduction to SAT. Submitted manuscript.

[48] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3: 519–550, Cambridge University Press, 2003.

[49] M. Hennessy and R. Milner. Algebraic laws for non-determinism and concurrency. *Journal of the ACM*, 32(1): 137–161, 1985.

[50] G. Holzmann. Design and Validation of Computer Protocols. Prentice Hall, 1990.

[51] T. Janhunen. Representing Normal Programs with Clauses. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'2004)*, pages 358–362, Valencia, Spain, august 2004.

[52] M. Jehle, J. Johannsen, M. Lange, N. Rachinsky. Bounded Model Checking for All Regular Properties. *In Proceedings of the 3rd International Workshop on Bounded Model Checking (BMC'2005)*, volume 144 (1) of Electronic Notes in Theoretical Computer Science, pages 3–18, Elsevier Science, 2005.

[53] M. Jurdziński. Deciding the winner in parity games is in $UP \cap co-UP$. *Information Processing Letters*, 68:119–124, 1998.

[54] M. Jurdziński. Small progress measures for solving parity games. In *Proceedings of 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS'00)*, Lecture Notes in Computer Science pages 1770, 290–301, Springer-Verlag, 2000.

[55] M. Jurdziński, M. Paterson and U. Zwick. A Deterministic Subexponential Algorithm for Solving Parity Games. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms, SODA'2006*, pages 114–123, ACM/SIAM, 20006, To appear.

[56] M. Keinänen. Obtaining Memory Efficient Solutions to Boolean Equation Systems. *Electronic Notes in Theoretical Computer Science*, 133: 175–191. Elsevier, 2005.

[57] M. Keinänen and I. Niemelä. Solving Alternating Boolean Equation Systems in Answer Set Programming. In *Applications of Declarative Programming and Knowledge Management, Revised Selected Papers from the 15th International Conference on Applications of Declarative Programming and Knowledge Management*, Lecture Notes in Artificial Intelligence 3392, pages 134–148, Springer-Verlag, 2005.

[58] B. Kernighan and D. Rithchie. The C Programming Language. Prentice Hall, 1988.

[59] V. King, O. Kupferman and M. Vardi. On the complexity of parity word automata. In *Proceedings of 4th International Conference on Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science 2030, pages 276–286, Springer-Verlag, 2001.

[60] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[61] K. N. Kumar, C. R. Ramakrishnan, and S. A. Smolka. Alternating fixed points in Boolean equation systems as preferred stable models. In *Proceedings of the 17th International Conference of Logic Programming*, Lecture Notes in Computer Science 2237, pages 227–241, Springer-Verlag, 2001.

[62] O. Kupferman and M. Vardi. μ-Calculus Synthesis. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'2000)*, Lecture Notes in Computer Science 1893, pages 497–507, Springer-Verlag, 2000.

[63] M. Lange. Solving Parity Games by a Reduction to SAT. In *Proceedings of International Workshop on Games in Design and Verification, GDV'2005*, 2005.

[64] K. Larsen. Efficient Local Correctness Checking. In *Proceedings of Conference on Computer Aided Verification*, Lecture Notes on Computer Science 663, pages 30–43, Springer-Verlag, 1992.

[65] C. M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371, Morgan Kaufmann Publishers, 1997.

[66] V. Lifschitz. Answer Set Planning. In *Proceedings of the 16th International Conference on Logic Programming*, pages 25–37, The MIT Press, 1999.

[67] X. Liu, C.R. Ramakrishnan and S.A. Smolka. Fully Local and Efficient Evaluation of Alternating Fixed Points. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1384, pages 5–19, Springer Verlag, 1998.

[68] X. Liu and S.A. Smolka. Simple Linear-Time Algorithms for Minimal Fixed Points. In *Proceedings of the 26th International Conference on Automata, Languages, and Programming*, Lecture Notes in Computer Science 1443, pages 53–66, Springer Verlag, 1998.

[69] D. Long, A. Browne, E. Clarke, S. Jha and W. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *Proceedings of International Conference on Computer Aided Verification (CAV'1994)*, Lecture Notes in Computer Science 818, pages 338–350, Springer-Verlag, 1994.

[70] A. Mader. Verification of Modal Properties using Boolean Equation Systems. PhD thesis, Technical University of Munich, 1997.

[71] M. Mahfoudh, P. Niebert, E. Asarin and O. Maler. A Satisfiability Checker for Difference Logic. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing SAT'02*, pages 222–230, 2002.

[72] W. Marek and M. Truszczyński. Autoepistemic logic. Journal of the ACM, 38:588–619, 1991.

[73] W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398, Springer-Verlag, 1999.

[74] R. Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, Lecture Notes in Computer Science 1785, pages 251–265, Springer-Verlag, 2000.

[75] R. Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2003)*, Lecture Notes in Computer Science 2619, pages 81–96, Springer Verlag, 2003.

[76] R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

[77] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik", Chaff: Engineering an Efficient SAT Solver In *Proceedings of 38th Design Automation Conference DAC'01*, 2001.

[78] I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

[79] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'2005)* Lecture Notes in Computer Science 3576, pages 321–334, Springer-Verlag, 2005.

[80] D. Niwiński. Fixed point characterization of infinite behavior of finite-state systems. *Theoretical Computer Science*, 189(1–2):1–69, 1997.

[81] J. Obdržálek. Fast Mu-calculus Model Checking when Tree-width is Bounded. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'2003)*, Lecture Notes in Computer Science 2725, pages 80–92, Springer-Verlag, 2005.

[82] G. Pace, F. Lang, and R. Mateescu. Calculating τ-Confluence Compositionally. In *Proceedings of Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2725, pages 446–459, Springer-Verlag, 2003.

[83] C. Papadimitriou. Computational Complexity. Addison-Wesley, 1994.

[84] R. Pelánek. Typical Structural Properties of State Spaces. In *Model Checking Software: 11th International SPIN Workshop, Proceedings*, Lecture Notes in Computer Science 2989, pages 5–22, Springer-Verlag, 2004.

[85] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift and D.S. Warren. Efficient Model Checking Using Tabled Resolution. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'1997)*, Lecture Notes in Computer Science 1254, pages 143–154, Springer-Verlag, 1997.

[86] C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X Du, A. Roychoudhury, V.N. Venkatakrishnan. XMC: a logic-programming-based verification toolset. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'2000)*, Lecture Notes in Computer Science 1855, pages 576–580, Springer-Verlag, 2000.

[87] D. Schmitz and J. Vöge. Implementation of a Strategy Improvement Algorithm for Finite-State Parity Games. In *Proceedings of the 5th International Conference on Implementation and Application of Automata*, Lecture Notes in Computer Science 2088, pages 263–271, Springer-Verlag, 2000.

[88] H. Seidl. Fast and Simple Nested Fixpoints. *Information Processing Letters*, 59(6): 303–308, 1996.

[89] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[90] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[91] B. Steffen, A. Classen, M. Klein, J. Knoop and T. Margaria. The fixpoint analysis machine. In I. Lee and S.A. Smolka, editors, In *Proceedings of the Sixth International Conference on Concurrency Theory (CONCUR '95)*, Lecture Notes in Computer Science 962, pages 72–87, Springer-Verlag, 1995.

[92] C. Stirling. Local model checking games. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'1995)*, Lecture Notes in Computer Science 962, pages 1–11, Springer-Verlag, 1995.

[93] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[94] A. Tanenbaum. Computer Networks. Prentice Hall PTR, fourth edition, 2003.

[95] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[96] R.E. Tarjan. A hierarchical clustering algorithm using strong components. *Information Processing Letters*, 14(1):26–29, 1982.

[97] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[98] G.S. Tseitin. On the complexity of derivation in propositional calculus. Stud. Constructive Math. and Math. Logic 2: 115–125, 1968.

[99] A. Valmari. A Stubborn Attack on State Explosion. In *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '1990)*, Lecture Notes in Computer Science 531, pages 156–165, Springer-Verlag, 1991.

[100] M. Vardi and P. Wolper. Yet another process logic. In *Logics of Programs*, Lecture Notes in Computer Science 164, pages 501–512, Springer-Verlag, 1984.

[101] B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating Boolean equation systems. In *Proceedings of the 21st International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 820, pages 302–315, Springer-Verlag, 1994.

[102] J. Vöge and M. Jurdzinski A Discrete Strategy Improvement Algorithm for Solving Parity Games. In *Proceedings of Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1855, pages 202–215, Springer-Verlag, 2000.

[103] U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1–2):343–359, 1996.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE RESEARCH REPORTS

HUT-TCS-A92	Timo Latvala, Armin Biere, Keijo Heljanko, Tommi Junttila Simple Bounded LTL Model Checking. July 2004.
HUT-TCS-A93	Tuomo Pyhälä Specification-Based Test Selection in Formal Conformance Testing. August 2004.
HUT-TCS-A94	Petteri Kaski Algorithms for Classification of Combinatorial Objects. June 2005.
HUT-TCS-A95	Timo Latvala Automata-Theoretic and Bounded Model Checking for Linear Temporal Logic. August 2005.
HUT-TCS-A96	Heikki Tauriainen A Note on the Worst-Case Memory Requirements of Generalized Nested Depth-First Search. September 2005.
HUT-TCS-A97	Toni Jussila On Bounded Model Checking of Asynchronous Systems. October 2005.
HUT-TCS-A98	Antti Autere Extensions and Applications of the A^* Algorithm. November 2005.
HUT-TCS-A99	Misa Keinänen Solving Boolean Equation Systems. November 2005.
HUT-TCS-A100	Antti E. J. Hyvärinen SATU: A System for Distributed Propositional Satisfiability Checking in Computational Grids. February 2006.
HUT-TCS-A101	Jori Dubrovin Jumbala — An Action Language for UML State Machines. March 2006.
HUT-TCS-A102	Satu Elisa Schaeffer Algorithms for Nonuniform Networks. April 2006.
HUT-TCS-A103	Janne Lundberg A Wireless Multicast Delivery Architecture for Mobile Terminals. May 2006.
HUT-TCS-A104	Heikki Tauriainen Automata and Linear Temporal Logic: Translations with Transition-Based Acceptance. September 2006.
HUT-TCS-A105	Misa Keinänen Techniques for Solving Boolean Equation Systems. November 2006.

ISBN–13 978-951-22-8546-4
ISBN–10 951-22-8546-0
ISSN 1457-7615