Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 99
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 99
Espoo 2005 HUT-TCS-A99

# 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 99
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 99
Espoo 2005 HUT-TCS-A99

# SOLVING BOOLEAN EQUATION SYSTEMS

Misa Keinänen

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
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi



ISBN 951-22-7994-0
ISSN 1457-7615

Multiprint Oy
Helsinki 2005

**ABSTRACT:** Boolean equation systems are ordered sequences of Boolean equations decorated with fixpoint operators. Boolean equation systems provide a useful framework for computer aided verification because various specification and verification problems can be encoded as the problem of solving such fixpoint equation systems. In this work, techniques for finding solutions to Boolean equation systems are studied, and new methods for solving such systems are devised. An approach to solve a general form Boolean equation system, which simplifies the process of finding the solution by dividing the system into more simple subsystems and solving these by optimized procedures, is introduced and analyzed. New solution algorithms for disjunctive and conjunctive classes of Boolean equation systems are presented, together with an implementation and experimental evaluation of a solver for these classes. A novel translation of the problem of solving a general form Boolean equation system into the problem of finding a stable model of a logic program is given. The translation allows to use an implementation of an answer set programming framework, the `Smodels` system, to solve Boolean equation systems. Experimental tests have been performed using the presented approach and these experiments indicate the effectiveness of using answer set programming in this problem domain.

**KEYWORDS:** Boolean equation systems, computer aided verification, model checking, logic programs, stable model semantics

# CONTENTS

# 1 INTRODUCTION

In this report, we study Boolean equation systems [1, 42, 45, 57]. These are ordered sequences of Boolean equations decorated with fixpoint signs. More precisely, a Boolean equation system consists of equations with Boolean variables in left-hand sides and positive propositional formulas in 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.

Boolean equation systems provide a useful framework for studying verification problems of finite-state concurrent systems, mainly because model checking problem of $\mu$-calculus [35] can be translated into this formalism (see [4, 42, 45] for such translations). Model checking is a verification technique aimed at determining whether a system model satisfies desired properties expressed as temporal logic formulas (see [11] for a survey). Modal $\mu$-calculus [35] is an expressive logic for verification, and most model checking logics can be encoded in the $\mu$-calculus.

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

## 1.1 Related Work

The notion of a Boolean equation system goes back at least to the work of Larsen [37], 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 simple 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 [57], and by Andersen and Vergauven [2].

In [42], Mader studies basic properties of Boolean equation systems. Mader shows how the model checking problem of full $\mu$-calculus can be solved in terms of Boolean equation systems. In addition, she provides a proof system for solving general Boolean equation systems by means of algebraic manipulations. Such an approach is also applicable to solve infinite systems of equations, an extension of Boolean equation systems to infinite sequences of Boolean equations involving infinite Boolean formulas.

In [45], Mateescu describes solution algorithms for alternation-free Boolean equation system. The approach from [45] can be used for both bisimulation checking and for model checking of alternation-free $\mu$-calculus on finite-state systems. Furthermore, in [44], 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 [36], Kumar and others apply answer set programming to solve Boolean equation systems. They argue that general form Boolean equation systems can be solved 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 centred 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 [25], Groote and Willemse show how a $\mu$-calculus formula and a process algebraic specification, both involving data parameters, can be transformed into a parameterized Boolean equation system. In [26], 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 fixpoint equation systems have been studied in some depth in the recent systems verification literature.

## 1.2 Contribution of the Report

Firstly, the work reports 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. Firstly, 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 enables considerable optimization of the solution methods.

Secondly, the work presents novel 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 observation [23] that many practically relevant $\mu$-calculus formulas can be encoded as Boolean equation systems that consist of conjunctive and disjunctive blocks. Hence, the problem of solving these subclasses is so important that developing special purpose solution techniques for these classes is worthwhile.

Mateescu [45] presents a solution algorithm for disjunctive/conjunctive Boolean equation systems. However, this approach is restricted to alternation-free systems. We are only aware of one sketch of an algorithm that is directed to alternating disjunctive/conjunctive Boolean equation systems, namely Proposition 6.5 and 6.6 of [42]. Here a $O(n^2)$ time and $O(n^2)$ space algorithms are provided where $n$ is the number of variables[1]. Our algorithms [23, 24] are substantial improvement over this.

The work presents efficient solution methods for 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 [42], it should be possible to employ fast answer set programming solvers (such as `DLV` and `Smodels`) as effective proof

[1]The work [42] presents an $O(n^2)$ time algorithm assuming the existence of an algorithm which allows union of (large) sets, and finding and deletion of elements in these sets in constant time. However, we are not aware of any data structure which allows all of these operations in constant time.

engines to solve such Boolean equation systems[2]. As mentioned before, this kind of approach is already suggested in [36].

But, while the translation from Boolean equation systems to logic programs presented in [36] preserves the linear-time complexity of solving alternation-free Boolean equation systems, it does not preserve the polynomial time complexity of solving conjunctive and disjunctive form Boolean equation systems. Moreover, the approach proposed in [36] does not seem to preserve the best known NP∩co-NP time complexity of solving general Boolean equation systems. An additional drawback of the approach from [36] is that typical answer set programming systems (such as `DLV` and `Smodels`) do not support the computation of answer sets satisfying the kind of preference criteria defined in [36].

We overcome the above drawbacks by introducing a novel mapping [32] from Boolean equation systems to normal logic programs. Namely, we reduce the problem of solving alternating Boolean equation systems to computing stable models of normal logic programs. Our translation is such that it ensures polynomial time complexity of solving both disjunctive and conjunctive alternating systems, and also ensures NP∩co-NP time complexity of solving general form Boolean equation systems. In addition, our translation only uses the kinds of rules that are already implemented in `DLV` and `Smodels` answer set programming systems.

Finally, it is worthwhile to observe that, by computational complexity results, it is possible to construct a mapping from Boolean equation systems to propositional satisfiability (for a description of propositional satisfiability problem, see p. 77 in [49]).[3] In principle, this kind of mapping would give a way to solve Boolean equation systems with various propositional satisfiability checkers. However, no mappings from Boolean equation systems to propositional satisfiability has been presented in the literature. One can use our translation from Boolean equation systems to normal logic programs as a basis for such a mapping because there exists a succinct encoding [29] of normal logic programs as propositional satisfiability.

In summary, the main contributions of this work are:

- The design of an approach to solve a general form Boolean equation system which works by dividing the system into individual blocks and solving these blocks in isolation. This simplifies the process of finding solutions to Boolean equation systems in many settings. The approach is presented and discussed in [32, 23].

- The introduction of novel solution algorithms for conjunctive and disjunctive classes of Boolean equation systems. This improves the best known upper bound for solving conjunctive and disjunctive portions of a Boolean equation system and makes the verification of a large class

[2]For example, see `http://www.dbai.tuwien.ac.at/proj/dlv/` and `http://www.tcs.hut.fi/Software/smodels/`) for descriptions of these answer set programming systems.

[3]Note that the problem of solving a Boolean equation system is itself very different from the satisfiability problem of propositional logic. Indeed, the question of satisfiability does not make a clear sense in the setting of Boolean equation systems because the propositional formulas appearing in such systems are positive, and all positive Boolean formulas are always satisfiable.

of fixpoint expressions more tractable. These results are presented and discussed in [23, 24].

- The development of a novel characterization of solutions to Boolean equation systems with alternating fixed points, and the design of a mapping from such systems to normal logic programs. This enables the application of answer set programming techniques to solve hard instances of Boolean equation systems, and gives a new efficient way of solving alternating portions of such systems. The approach is presented and discussed in [32]. In addition, when combined with the results from [29], our translation enables the the application of propositional satisfiability checkers to solve Boolean equation systems.

- The implementation and initial experimental evaluation of a solver for conjunctive and disjunctive Boolean equation systems with alternation. These are described in [23], and are also presented partly in [31].

- The implementation and initial experimentation of the answer set programming approach to solve alternating Boolean equation systems, as a proof of concept on the simplicity and effectiveness of using logic programming in this problem domain. These results are described in [32].

## 1.3 Organization of the Report

The organization of this work is as follows. Section 2 provides the needed background to read the work. 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 a Boolean equation system. Section 6 details an answer set programming approach to solve general form, alternating parts of Boolean equation systems. Section 7 details some case studies. Finally, Section 8 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 Boolean equation systems, modal $\mu$-calculus, and answer set programming.

## 2.1 Boolean Equation Systems

We will give in this subsection a short presentation of Boolean equation systems. Essentially, a Boolean equation system is an ordered sequence of fixed point equations over Boolean variables, with associated signs, $\mu$ and $\nu$, specifying the polarity of the fixed points. The equations are of the form $\sigma x = \alpha$, where $\alpha$ is a positive Boolean expression. The sign, $\sigma$, is $\mu$ if the equation is a least fixed point equation and $\nu$ 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:

$$\alpha ::= \; 0 \mid 1 \mid x_i \mid \alpha \wedge \alpha \mid \alpha \vee \alpha$$

where 0 stands for *false*, 1 stands for *true*, and $x_i \in \mathcal{X}$. The meaning of positive Boolean expressions is trivially defined as the usual semantics for Boolean formulas.

We define the syntax of Boolean equation systems as follows.

**Definition 1 (The syntax 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 is an ordered sequence of Boolean equations*

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

*where the left-hand sides of the equations are all different. We assume that all right-hand side variables are from $\mathcal{X}$.*

The priority ordering on variables and equations of a Boolean equation system is important for it ensures the existence of a unique solution. But, 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 2 (The size of a Boolean equation system)** *The size of a Boolean equation system*

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

*is*

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

*where* $|\alpha_i|$ *is the number of variables in* $\alpha_i$*.*

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

**Definition 3 (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 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 [42]. The idea is that to determine the alternation depth only chains of equations in a Boolean equation system must be followed that depend on each other. Using for instance Lemma 3.22 of [42] a Boolean equation system can be reordered such that our notion of alternation depth and the notion of [42] coincide.

Notice that for each equation system $\mathcal{E}$ with variables from $\mathcal{X}$ we have that $ad(\mathcal{E}) < |\mathcal{X}|$. That is, the alternation depth of a Boolean equation system is always less than the number of variables involved.

As pointed out in [42] (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 4 (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* $\alpha_i$ *has the form* $y \circ z$ *or* $y$ *where* $\circ \in \{\wedge, \vee\}$ *and* $y, z \in \{x_1, x_2, \dots, x_n\} \cup \{0, 1\}$*.*

The transformation from unrestricted Boolean equation systems to standard form systems is based on the fact that, for each Boolean equation ($\sigma x = y \circ \alpha$) 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 such that the right-most equations have higher priority over left-most equations (see, e.g., [1, 42]).

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 $\alpha$ be a closed positive Boolean expression (i.e. without occurrences of variables in $\mathcal{X}$). Then $\alpha$ has a uniquely determined value in the set $\{0, 1\}$ which we denote by $\|\alpha\|$. We define a substitution for positive Boolean expressions. Given Boolean expressions $\alpha, \beta \in B(\mathcal{X})$, let $\alpha[x/\beta]$ denote the expression $\alpha$ where all occurrences of variable $x$ are substituted by $\beta$ 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 following definition of the solution is from [32].

**Definition 5 (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*

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

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

The following example illustrates the definition of the solution.

**Example 6** *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*
$[\![(\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee x_3)(\nu x_3 = x_3)]\!] =$
$[\![(\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee x_3)[x_3/1]]\!] =$
$[\![(\nu x_1 = x_2 \wedge x_1)(\mu x_2 = x_1 \vee 1)]\!] =$
$[\![(\nu x_1 = x_2 \wedge x_1)[x_2/x_1 \vee 1]]\!] =$
$[\![(\nu x_1 = (x_1 \vee 1) \wedge x_1)]\!] = \|((1 \vee 1) \wedge 1)\| = 1$

We have taken a definition of the semantics which is non-standard in the sense that it characterizes a *local* value only for the least variable $x_1$. This deviates from the standard definition of a solution to a Boolean equation system which gives a *global* solution, i.e. solutions to all variables.

**Global Semantics of Boolean Equation Systems**

As opposed to Definition 5, the standard semantics of Boolean equation systems provides a uniquely determined solution to each variable of a Boolean equation system. According to the standard definition, a solution is a valuation assigning a constant value in $\{0, 1\}$ to all variables occurring in a system. To illustrate the difference, we restate here the standard semantics but a reader familiar with the standard semantics may well skip this subsection.

Let $\theta$ stand for a valuation which is a function $\theta : \mathcal{X} \to \{0,1\}$. Let $\theta[x{:=}a]$ denote the valuation that coincides with $\theta$ for all variables except $x$ which has the value $a$. Similarly, given distinct variables $x_i, ..., x_j \in \mathcal{X}$, $\theta[x_i{:=}a_i, ..., x_j{:=}a_j]$ denotes the valuation that coincides with $\theta$ for all variables except for $x_i, ..., x_j$ which have the values $a_i, ..., a_j$.

We extend the definition of valuations to terms in the standard way. So, $\theta(\alpha)$ is the value of the term $\alpha$ after replacing each free variable $x$ in $\alpha$ by $\theta(x)$. We suppose that $[x{:=}a]$ has priority over all operations and $\theta[x{:=}a]$ stands for $(\theta[x{:=}a])$. Similarly, we apply $[x{:=}a]$ to terms; $\alpha[x{:=}a]$ indicates the term $\alpha$ where all occurrences of $x$ have been replaced by $a$.

The standard, global definition of a solution to a Boolean equation system is given as follows (see also, e.g., Definition 3.3 in [42]).

**Definition 7 (The global solution to a Boolean equation system)** *The global solution to a Boolean equation system $\mathcal{E}$ relative to a valuation $\theta$, denoted by $[\![\mathcal{E}]\!]\theta$, is an assignment inductively defined by*

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

*Here*

$$MIN(x_i, \alpha_i, \mathcal{E}, \theta) = \bigwedge\{a | \alpha_i([\![\mathcal{E}]\!](\theta[x_i{:=}a])) \Rightarrow a\}$$
$$MAX(x_i, \alpha_i, \mathcal{E}, \theta) = \bigvee\{a | a \Rightarrow \alpha_i([\![\mathcal{E}]\!](\theta[x_i{:=}a]))\}$$

*where $x_i$ is a variable in $\mathcal{X}$, $\alpha_i$ a positive Boolean formula over variables in $\mathcal{X}$, $\mathcal{E}$ a Boolean equation system, and $\theta$ a valuation. Notice that $\epsilon$ denotes an empty Boolean equation system, the operator $\bigvee$ denotes the least upper bound and $\bigwedge$ the greatest lower bound of the Boolean lattice* $(\{0,1\}, \Rightarrow)$.

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

**Example 8** *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 7, the solution to this system is calculated as follows. Consider an arbitrary valuation $\theta$. First, we calculate*

$$\theta[x_1{:=}MIN(x_1, x_2, (\nu x_2 = x_1), \theta)]. \tag{1}$$

*Thus, we calculate*

$$MIN(x_1, x_2, (\nu x_2 = x_1), \theta) = \bigwedge\{a | x_2([\![(\nu x_2 = x_1)]\!](\theta[x_1{:=}a])) \Rightarrow a\} \tag{2}$$

*and within this*

$$[\![(\nu x_2 = x_1)]\!](\theta[x_1{:=}\ a]) = [\![(\nu x_2 = x_1)\epsilon]\!](\theta[x_1{:=}\ a]) =$$

$$[\![\epsilon]\!](\theta[x_1{:=}a, x_2{:=}MAX(x_2, x_1, \epsilon, (\theta[x_1{:=}a]))]) = \theta[x_1{:=}a, x_2{:=}a].$$

*Now, (2) $= \bigwedge\{a | a \Rightarrow a\} = \bigwedge\{0,1\} = 0$. Thus, (1) is $v[x_1{:=}0]$. Hence, the solution to $\mathcal{E}_2$ is:*
$[\![(\mu x_1 = x_2)(\nu x_2 = x_1)]\!]\theta =$

$[\![(\nu x_2 = x_1)]\!]\theta[x_1{:=}MIN(x_1, x_2, (\nu x_2 = x_1), \theta)] =$
$[\![(\nu x_2 = x_1)]\!]\theta[x_1{:=}0] =$
$[\![\epsilon]\!]\theta[x_1{:=}0, x_2{:=}MAX(x_2, x_1, \epsilon, \theta[x_1{:=}0])] =$
$\theta[x_1{:=}0, x_2{:=}0].$

When applied to non-trivial Boolean equation systems, i.e. to systems involving more than two simple equations, the standard definition of the semantics is quite tedious. Therefore, in this report we have adopted the local semantics given in Definition 5.

But, notice that our local definition can easily be used to find the global solutions as well. Of course, the local semantics in Definition 5 coincides with the above standard semantics in Definition 7.

There are also alternative characterizations of the solution to a Boolean equation system which help to provide more insight, for instance Proposition 3.6 in [42] and Definition 1.4.10 in [4].

## 2.2 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 9 (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*

- $V = \{i \mid 1 \leq i \leq n\}$ *is the set of nodes;*
- $E \subseteq V \times V$ *is the set of edges such that, for all equations $\sigma_i\, x_i = \alpha_i$, $(i, j) \in E$ iff a variable $x_j$ occurs in $\alpha_i$*
- $\ell$ *is a labelling function defined by $\ell(i) = \sigma_i$.*

We often omit the labelling function $\ell$ 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 report.

**Definition 10 (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 the nodes $v_0, v_1, v_2, ..., v_k$.*

**Definition 11 (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 12 (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 $k \geq 2$.*

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

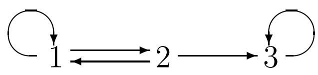


Figure 1: The dependency graph of Boolean equation system $\mathcal{E}_1$ in Example 6.

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 directed path from node $i$ to node $j$. It is said that two variables $x_i$ and $x_j$ 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 $x_i$ and $x_j$ are 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 such 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 13** *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 $G{\restriction}k$.*

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

**Example 14** *Consider the Boolean equation system $\mathcal{E}_1$ of Example 6. The dependency graph of $\mathcal{E}_1$ is depicted in Figure 1. 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 15 (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 each pair of nodes $k, l \in W$, $l$ is reachable from $k$ in $E$. A strongly connected component is called maximal if there does not exist a larger set of nodes which is also a strongly connected component. A maxiaml 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.*

These graph-theoretic notions and definitions are very standard in the literature.

## 2.3 Modal $\mu$-Calculus

In this subsection, we will briefly give the basic definitions concerning modal $\mu$-calculus [35]. The modal $\mu$-calculus is based on fixpoint computations [56], and a more detailed survey on this logic can be found from [9]. In this subsection, we will also discuss the connections between $\mu$-calculus model checking problem and Boolean equation systems.

**Syntax of $\mu$-Calculus**

Modal $\mu$-calculus [35] is an expressive logic for system verification, and most model checking logics can be encoded in the $\mu$-calculus. Many important features of system models, like equivalence/preorder relations and fairness constraints, can also be expressed with the logic. For these reasons, $\mu$-calculus is a logic widely studied in the recent systems verification literature.

We define the syntax of modal $\mu$-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 modal $\mu$-calculus formulas with respect to $\mathcal{Z}$ and $\mathcal{L}$ is defined as follows:

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

As usual, for the above syntax we assume that modal operators ($[a]$ and $\langle a \rangle$) have higher precedence than Boolean connectives ($\wedge$ and $\vee$), and that fixpoint operators ($\mu$ and $\nu$) have 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 $[-]\Phi$ means $\forall a \in \mathcal{L} : [a]\Phi$, and notation $[-b]\Phi$ means $\forall a \in \mathcal{L} \setminus \{b\} : [a]\Phi$.

**Semantics of $\mu$-Calculus**

Given a set $\mathcal{L}$ of action labels, formulas of modal $\mu$-calculus are interpreted relative to a *labelled transition system* $\mathcal{T} = (S, \{\xrightarrow{a} \mid a \in \mathcal{L}\})$ where $S$ is a (finite) set of states and, for every $a \in \mathcal{L}$, relation $\xrightarrow{a} \subseteq S \times S$ is a transition relation. We will write $(s, t) \in \xrightarrow{a}$ as $s \xrightarrow{a} t$.

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

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

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

The *model checking problem* for $\mu$-calculus can be stated as follows.

**Definition 16 ($\mu$-calculus model checking)** *Given a $\mu$-calculus formula $\Phi$ and a state $s$ of a labelled transition system $\mathcal{T}$, the $\mu$-calculus model checking problem is to determine whether $\mathcal{T}, s \models \Phi$ holds.*

It is well-known that the $\mu$-calculus model checking problem is in the complexity class NP $\cap$ co-NP. Emerson, Jutla, and Sistla [18, 19] show that the problem can be reduced to determining the winner in a parity game, and thus is in NP (and also by symmetry in co-NP). Jurdzinsky [30] show that the problem is even in the complexity class UP $\cap$ co-UP. Yet, the complexity of the $\mu$-calculus model checking problem for the unrestricted logic is an open problem; no polynomial algorithm has been discovered so far.

Nevertheless, various effective model checking algorithms exist for expressive subsets. Arnold and Crubille [3] present an algorithm for checking alternation depth 1 formulas of $\mu$-calculus which is linear in the size of the model and quadratic in the size of the formula. Cleaveland and Steffen [13] improve this result by making the algorithm linear also in the size of the formula. Andersen [1], and similarly Vergauwen and Lewi [57], 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 $\mu$-calculus have been investigated by Bhat and Cleaveland [5] as well as Emerson et al. [18, 19]. They present polynomial time model checking algorithms for fragments L1 and L2 which may contain alternating fixed point formulas.

| Property | Formula |
|---|---|
| No deadlock can occur (i.e. in all states some action is enabled). | $\nu Z.([-]Z \wedge \langle - \rangle \top)$ |
| An *error* action does not occur along any execution path. | $\nu Z.([error]\bot \wedge [-]Z)$ |
| A *send* action is always eventually 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 2: Examples of properties expressed in modal $\mu$-calculus.

Modal $\mu$-calculus allows to express very concisely a wide range of useful properties of finite-state concurrent systems. Figure 2 shows some typical examples of such properties encoded as modal $\mu$-calculus formulas (for a comprehensive survey of the use of fixpoint operators, see [9]). More examples of formulas expressing system properties will be given in Section 7.

**From $\mu$-Calculus to Boolean Equation Systems**

In this report, instead of treating $\mu$-calculus expressions together with their semantics we 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 $\mu$-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 $\mu$-calculus model checking problem.

In this subsection, we demonstrate the *standard translation* from a $\mu$-calculus formula and a labelled transition system to a Boolean equation system as defined in [45]. Similar translations serving the same purpose are presented, for example, in [1, 4, 42].

The transformation maps a modal $\mu$-calculus formula $\Phi$ 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 $\mu$-calculus formula $\Phi$ imposed by text, and then map the $\mu$-calculus expression $\Phi$ 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 $\Phi$ to ensure that in every subformula $\sigma X.\Phi'$ of $\Phi$ with $\sigma \in \{\mu, \nu\}$, $\Phi'$ 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 $\Phi$. 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.

| $\Phi$ | $(\Phi)_s$ | $\Phi$ | $(\Phi)_s$ |
|---|---|---|---|
| $\bot$ | $0$ | $\top$ | $1$ |
| $\Phi_1 \vee \Phi_2$ | $(\Phi_1)_s \vee (\Phi_2)_s$ | $\Phi_1 \wedge \Phi_2$ | $(\Phi_1)_s \wedge (\Phi_2)_s$ |
| $\langle a \rangle \Phi$ | $\bigvee_{s \xrightarrow{a} t} (\Phi)_t$ | $[a]\Phi$ | $\bigwedge_{s \xrightarrow{a} t} (\Phi)_t$ |

Figure 3: The translation from a $\mu$-calculus formula to a Boolean equation system.

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 $\mu$-calculus formula and a labelled transition system (see Theorem 5.2 in [42]).

The following example illustrates the standard translation from $\mu$-calculus to Boolean equation systems.

**Example 17** *Consider the following $\mu$-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.*

*The closed fixed point formula*

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

*is first translated into a sequence of equations*

$$\left.(\nu\, z_s = ([-]z \wedge \langle - \rangle \top)_s)\ \right\}\forall s \in S$$

*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:*

$$\left.(\nu\, z_s = ([-]z)_s \wedge (\langle - \rangle \top)_s)\ \right\}\forall s \in S$$

*This sequence is translated to the Boolean equations below*

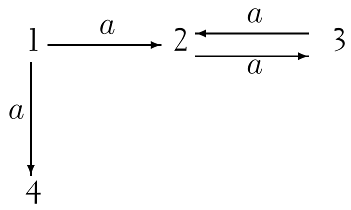
$$\left.(\nu\, z_s = (\bigwedge_{s \xrightarrow{a} s'} z_{s'}) \ \wedge\ (\bigvee_{s \xrightarrow{a} s'} (\top)_{s'}))\ \right\}\forall s \in S$$

*where each variable $Z_s$ expresses that state $s$ satisfies variable $Z$ of the $\mu$-calculus formula.*

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

$$\begin{array}{l}
(\nu Z_0 = (Z_1 \wedge Z_2) \wedge (1 \vee 1)) \\
(\nu Z_1 = 1 \wedge 0) \\
(\nu Z_2 = Z_3 \wedge 1) \\
(\nu Z_3 = Z_2 \wedge 1)
\end{array}$$

(a)




(c)

$$
\begin{array}{l}
(\nu x_1 = x_2 \wedge x_3) \\
(\nu x_2 = x_4 \wedge x_5) \\
(\nu x_3 = 1 \vee 1) \\
(\nu x_4 = 1 \wedge 0) \\
(\nu x_5 = x_6 \wedge 1) \\
(\nu x_6 = x_5 \wedge 1)
\end{array}
$$

(b)

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

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

*Notice in this step that an empty disjunction is written as* 0 *and an empty conjunction is written as* 1.

*Then, by introducing fresh variables we may transform the above system to standard form, and we obtain the following equation system:*

$$
\begin{array}{l}
(\nu Z_0 = Z_0' \wedge Z_0'') \\
(\nu Z_0' = Z_1 \wedge Z_2) \\
(\nu Z_0'' = 1 \vee 1) \\
(\nu Z_1 = 1 \wedge 0) \\
(\nu Z_2 = Z_3 \wedge 1) \\
(\nu Z_3 = Z_2 \wedge 1)
\end{array}
$$

*Finally, by renaming the variables we get the Boolean equation system over* $\mathcal{X} = \{x_1, x_2, ..., x_6\}$

$$
\begin{array}{l}
(\nu x_1 = x_2 \wedge x_3) \\
(\nu x_2 = x_4 \wedge x_5) \\
(\nu x_3 = 1 \vee 1) \\
(\nu x_4 = 1 \wedge 0) \\
(\nu x_5 = x_6 \wedge 1) \\
(\nu x_6 = x_5 \wedge 1)
\end{array}
$$

*which corresponds to the given model checking problem. Notice that the last two steps are not necessary for the translation.*

Additional examples of the mapping will be given in Section 7. Next, we turn to issues concerning logic programs and answer set programming.

## 2.4 Normal Logic Programs

In this report, we often use normal logic programs with the stable model semantics [22] for encoding and solving Boolean equation systems. In this subsection, we provide a brief introduction to normal logic programs and stable model semantics.

The definitions in this section appeared also in [32], and they are very standard. A complete description of these topics and notions can be found, for instance, in [15].

Normal logic programs consist of *rules*. A normal rule is of the form

$$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n. \tag{3}$$

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 $\Delta$ 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 (3) is satisfied by $\Delta$ 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 $\Delta$ and a program $\Pi$ is satisfied by $\Delta$ if each rule in $\Pi$ is satisfied by $\Delta$.

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 $\Pi$ and a set of atoms $\Delta$, the reduct $\Pi^{\Delta}$ 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 $\Pi^{\Delta}$ 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 18 (Stable models of a logic program)** *A set of atoms $\Delta$ is called a stable model of a program $\Pi$ iff $\Delta$ is the unique minimal set of atoms satisfying $\Pi^{\Delta}$.*

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

**Example 19** *Let $\{a, b\}$ be the set of ground atoms. Consider the program:*

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

*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 20** *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 report, we will use two extensions which serve as shorthands for normal rules. We will use so-called *integrity constraints*. Integrity constraints are simply rules

$$\leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n. \tag{4}$$

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 $\Delta$ satisfies an integrity constraint (4) only if at least one of its body literals is not satisfied by $\Delta$.

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 [52].

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 [38, 43, 47].

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.

# 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 $B_j$. The resulting dependency relation among blocks is an ordering.

Below we have a simple example of such a partitioning on a Boolean equation system from a previous example.

**Example 21** *Consider again the Boolean equation system $\mathcal{E}_1$ of Example 6. 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.*

Notice that finding the blocks of a Boolean equation system can be done in linear time using any algorithm suitable to detect maximal strongly connected components in directed graphs, for instance, those from [51, 54].

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.

Furthermore, 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.

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

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 $\mu$ or $\nu$. 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 $\mu$-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) [28], Computation Tree Logic (CTL) [10], and many equivalence/preorder checking problems result in alternation-free Boolean equation systems with alternation-free blocks only (see for instance [45]).

In Section 4, we will review solution methods for alternation-free blocks. It will be seen that such blocks can easily 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 $\mu$-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 L1 and L2 fragments of the $\mu$-calculus [5, 18, 19] (and similar subsets) or many fairness constraints as Boolean equation systems result in alternating systems which are in a 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 $\mu$ and $\nu$. 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 of marginal interest because 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 – not to say all of them – are theoretical constructions (see, e.g., [8] 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 [42] (and it is known to be even in UP ∩ co-UP), in the same way as the equivalent problem of $\mu$-calculus model checking. Furthermore, it is widely believed that a polynomial time algorithm for the problem may well be found but the best known algorithms to date are exponential in the size of the system.

In Section 6, we will propose an approach to solve alternating blocks of a Boolean equation system which is based on answer set programming.

## 3.3 General Solution Algorithm for Boolean Equation Systems

In Mader [42], 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 22 (Lemma 6.2 of** [42]**)** *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 $\alpha_i'$ be exactly the same Boolean expression as $\alpha_i$, except that all occurrences of $x_i$ in $\alpha_i$ 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 23 (Lemma 6.3 of** [42]**)** *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 $\sigma_i x_i = \alpha_i$, $\sigma_i x_i = \alpha_i'$ and $\sigma_j x_j = \alpha_j$ be equations where $\alpha_i'$ is exactly the same Boolean expression as $\alpha_i$ except that all occurrences of $x_j$ are substituted with expression $\alpha_j$. 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 23 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 was also discussed in [32, 23]. Previously, a quite similar algorithm for equational systems has been given in [12].

**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 $B_m, \ldots, B_2, B_1$; here blocks are numbered so that higher-numbered variables belong to higher-numbered blocks.*
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 Section 6*

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

**Theorem 24** *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 23 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. So the complexity of the general procedure depends naturally on the costs of the subroutines.

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

**Example 25** *Consider again the Boolean equation system $\mathcal{E}_1$ from Example 6. In step 1, the algorithm builds the dependency graph of the system which is depicted in Figure 1. In step 2, the algorithm divides the system in blocks, as explained in Example 21, 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 $B_2, B_1$. Accordingly, in step 4 the algorithm first solves block $B_2$, and then solves 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, alternation-free system*

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

*As this subsystem is alternation-free, block $B_2$ can be solved by using appropriate techniques in step 4. (b) i.*

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

# 4 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 $\mu$, and, dually, all equations of a maximal block have the sign $\nu$. 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 exist 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] presented 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 addition, very simple linear-time algorithms to solve a Boolean equation system, whose all equations have the same fixpoint sign, can be found from [41] (see e.g. Fig. 2 on p. 5). In [41], there are also straightforward, linear-time reductions between alternation-free Boolean equation systems and HornSAT, the problem of Horn formula satisfiability. This allows to solve alternation-free Boolean equation systems by using a linear-time algorithm for HornSAT given in [17].

More recently, additional linear-time algorithms for alternation-free Boolean equation systems were presented in [45]. Very similar algorithms are presented in [46] too.

These kinds of standard algorithms can effectively be applied to solve minimal and maximal blocks in our setting. As they are very simple algorithms, we do not consider them in detail here.

Instead, in the following section, it will be seen how minimal and maximal blocks of a Boolean equation system can be solved with an approach based on logic programming.

## 4.2 Minimal and Maximal Blocks as Logic Programs

An alternative 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 [36]. In brief, it is suggested in [36] 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 [36] 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, [36] does not provide a complete translation but only sketches an informal idea via a few examples. However, it is important to notice that the same kind of idea based on logic programming approach can efficiently be applied to solve minimal and maximal blocks in our setting too.

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 [17].

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 $\mathcal{E}$.

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 $\mu x_i = \alpha_i$ of $\mathcal{E}$, the program $\Pi(\mathcal{E})$ contains the rules:

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

$$p_i \leftarrow p_j, p_k. \qquad \text{if } \alpha_i = x_j \wedge x_k \tag{6}$$

$$p_i \leftarrow p_j.\ p_i \leftarrow p_k. \qquad \text{if } \alpha_i = x_j \vee x_k \tag{7}$$

$$p_i. \qquad \text{if } \alpha_i = 1 \tag{8}$$

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 verify.

**Theorem 26** *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 7, Definition 18, and by the construction of $\Pi(\mathcal{E})$. □

By Theorem 26, 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 27** *Consider the standard form 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:*

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

*The stable model of program $\Pi(\mathcal{E})$ is $\{p_2, p_6\}$. As expected, by Theorem 26, the only variables of $\mathcal{E}$ with solution $1$ is $x_2$ 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 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 below.

**Definition 28 (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 29 (Lemma 3.35 of [42])** *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 29, 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 29.

**Example 30** *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 26, 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 29, 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 DISJUNCTIVE AND CONJUNCTIVE BLOCKS WITH ALTERNATION

In this section, we examine *conjunctive* and *disjunctive* fragments of Boolean equation systems. As pointed out in [23], many practically relevant properties of systems can be expressed by means of fixed point formulas that lead to Boolean equations in either conjunctive or disjunctive forms. It is therefore interesting to develop specific resolution techniques for disjunctive and conjunctive blocks with alternation.

We first introduce basic properties concerning conjunctive and disjunctive blocks of Boolean equation systems. Then, we present two distinct algorithms for solving disjunctive and conjunctive blocks. We also deal with the correctness and complexity of these algorithms. Finally, we provide comprehensive examples of how the algorithms work.

## 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 31** *Let $(\sigma x = \alpha)$ be an equation. We call this equation disjunctive if no conjunction symbol $\wedge$ appears in $\alpha$. Let $\mathcal{E}$ be a Boolean equation system. We call $\mathcal{E}$ disjunctive iff each equation in $\mathcal{E}$ is disjunctive.*

**Definition 32** *Let $(\sigma x = \alpha)$ be an equation. We call this equation conjunctive if no disjunction symbol $\vee$ appears in $\alpha$. Let $\mathcal{E}$ be a Boolean equation system. We call $\mathcal{E}$ conjunctive iff each equation in $\mathcal{E}$ is conjunctive.*

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

The following essential lemma comes from [23], and it is closely related to parity automata to be discussed in Section 5.3.

**Lemma 33** *Let $\mathcal{E}$ be a 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 23 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 $\beta_j$ is exactly the same Boolean expression as $\alpha_j$ except that, for $1 \leq i \leq n$, all occurrences of $x_{k_i}$ are substituted with expression $\alpha_{k_i}$. Now note that the right hand side $\beta_j$ of equation $\nu x_j = \beta_j$ contains only disjunctions and the variable $x_j$ at least once. Hence, by Lemma 22 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 $\beta_l$ 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 $\beta_k$ 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 $\alpha_k$ 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 $\beta_l$ only 0 and variables from $x_1, \ldots, x_k$ occur. Using Lemma 23 such variables $x_l$ can be replaced by $\beta_l$ and hence, $\alpha_k$ is replaced by $\gamma_k$ in which 0 and variables from $x_1, \ldots, x_k$ can occur.

What remains to be done is to remove $x_k$ from $\gamma_k$ assuming $x_k$ occurs in $\gamma_k$. This can be done as follows. Suppose $\sigma_k = \nu$. Then, as $x_k$ occurs in $\gamma_k$, 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 22 the variable $x_k$ in $\alpha_k$ can be replaced by 0. $\square$

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

**Lemma 34** *Let $\mathcal{E}$ be a 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 33. $\square$

One can see that, as a simple consequence of the above properties, all variables in a conjunctive or disjunctive blocks have the same solutions. We may thus solve all variables of a conjunctive or disjunctive block by simply computing the solution to the the smallest variable.

Furthermore, since a block 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.

Thus, the condition that needs to be checked is whether there is a cycle in the dependency graph of which the lowest numbered vertex has label $\nu$ (or $\mu$ respectively). In the following sections we define algorithms that perform this task efficiently.

## 5.2 Depth-First Search Based Algorithm

There is a very simple algorithm based on depth-first search [55] on directed graphs which can be used to solve conjunctive and disjunctive blocks of a Boolean equation system. The algorithm was first discussed in [23] 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.

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

$$minNuLoop(G, i)$$

to be true iff the subgraph $G \restriction i$ of $G$ contains a cycle $(i, v_0, v_1 ..., i)$ such that $\min\{i, v_0, v_1 ..., i\} = i$.

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 $minNuLoop(G, i)$ holds reduces to the task of computing the reachability of node $i$ from itself in the subgraph $G \restriction i$ 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 $minNuLoop(G, i)$, 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 $minNuLoop(G, i)$ holds, then report "solution to smallest variable is $1$" and STOP.*
2. *Report "solution to smallest variable is $0$".*

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

**Theorem 35 (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

$$minNuLoop(G, i)$$

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 33, 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 $minNuLoop(G, i)$ holds. By Lemma 33, the solution to smallest variable is 0. □

For instance, using a standard adjacency-list representation of dependency graphs, the worst-case time complexity of this algorithm is easily seen to be quadratic in the size of the block.

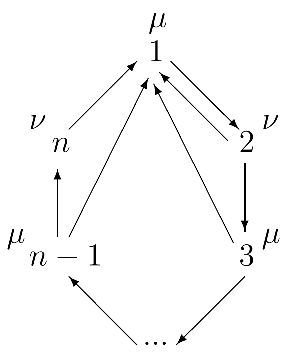



Figure 5: A worst-case example.

**Theorem 36** *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 *cycle* at most $n$ times and each call takes time $O(n + m)$. $\square$

Note that the space complexity of *SolveDisjunctive*$(G)$ is $O(|G|)$.

We now give an example block of a Boolean equation system which shows that the above algorithm may take quadratic time in the size of the block.

**Example 37** *For some even $n \in \mathbb{N}$ s.t. $n \geq 4$, consider the Boolean equation system:*

$$(\mu x_1 = x_2)$$

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

$$(\mu x_3 = x_2 \vee x_4)$$

$$\vdots$$

$$(\mu x_{n-1} = x_{n-2} \vee x_{n-1})$$

$$(\nu x_n = x_1)$$

*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. Note that, in order to solve the block with the depth-first search based algorithm from [23], we need at least*

$$|G{\restriction}2| + |G{\restriction}4| + \cdots + |G{\restriction}n| = O(n^2)$$

*steps.*

All previous algorithms for solving conjunctive and disjunctive blocks, including those from [23, 42], 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 and preorder/equivalence checking of realistic systems, these algorithms often lead to unpleasant running times. It has been an open question whether or not the quadratic upper bound for conjunctive and disjunctive Boolean equation systems could be improved (e.g. see [23]). In [24] we resolve the question by presenting an especially fast algorithm for finding a solution to a Boolean equation system in either conjunctive or disjunctive form. Given such a Boolean equation system with size $e$ and alternation depth $d$, our algorithm finds the solution using time $O(e \log d)$ in the worst case. This improves the best known upper bound and makes the verification of a large class of fixpoint expressions more tractable.

The algorithm in [24] combines graph theoretic techniques for finding strong components [51, 54] and hierarchical clustering [55]. King, Kupferman and Vardi [34] recently found an improved algorithm for deciding non-emptiness of parity word automata. Our algorithm is very similar to [34], and is also based on the ideas in [55].

## 5.3 Conjunctive/Disjunctive Blocks and Parity Word Automata

In this subsection, we relate the problem of solving conjunctive and disjunctive Boolean equation systems to the non-emptiness problem of parity word automata [34]. More precisely, we present a linear-time reduction from a disjunctive block of a Boolean equation system to the non-emptiness problem of parity word automata. Also, similar linear-time reduction can be given to the other direction, from a parity word automaton to a disjunctive block of a Boolean equation system.

This helps to clarify the connection between the two problems, and shows that the quadratic upper bound for conjunctive and disjunctive Boolean equation systems can be substantially improved.

Let us first define the non-emptiness problem of parity word automata.

**Definition 38** *A parity word automaton can be represented as a directed graph $G = (\{1, 2, ..., 2k\}, V, E, \ell)$ where $V$ is a set of nodes, $E \subseteq V \times V$ is a set of edges and $\ell : V \to \{1, 2, ..., 2k\}$ is a labelling function. Given such a graph $G$, the non-emptiness problem is to determine whether there is a cycle $C$ in $G$ such that $\min_{v \in C}\{\ell(v)\}$ is even.*

The following result stems from [34].

**Lemma 39 (Theorem 4 of [34])** *Let $G = (\{1, 2, ..., 2k\}, V, E, \ell)$ be a directed graph with $|V| = n$ and $|E| = m$ representing a parity word automaton. The non-emptiness problem for $G$ can be solved in time $O((n + m) \log k)$.*

By the application of the above lemma, we obtain an improved upper bound for solving a disjunctive block of a Boolean equation system.

**Theorem 40** *Let $B$ be a disjunctive block of a Boolean equation system and let $G = (V, E, \ell)$ be the dependency graph of $B$ with $|V| = n$ and $|E| = m$. The least variable of $B$ can be solved in time $O((n + m) \log n)$.*

**Proof:**
We present a linear-time reduction from $B$ to the non-emptiness problem described in Def. 38. Given the dependency graph $G = (V, E, \ell)$ of $B$, let $G' = (D', V', E', \ell')$ be another graph defined as follows:

- $D' = \{1, 2, \ldots, 2n\}$
- $V' = V$
- $E' = E$
- the labelling $\ell'$ is defined as

$$\ell'(1) = \begin{cases} 1 & \text{if } \sigma_1 = \mu \\ 2 & \text{if } \sigma_1 = \nu \end{cases}$$

  and for $1 < i \leq n$

$$\ell'(i) = \begin{cases} (2i) - 1 & \text{if } \sigma_i = \mu \\ (2i) & \text{if } \sigma_i = \nu \end{cases}$$

With suitable representations of the graphs $G$ and $G'$, the above mapping is a linear-time reduction.

We show that the least variable of $B$ holds iff the non-emptiness problem for $G'$ is positive.

Suppose the least variable of $B$ holds. By Lemma 33, $\exists j \in V$ with $\ell(j) = \nu$ such that $j$ is reachable from node 1 in $G$, and $G$ contains a cycle $C$ of which the lowest index of a node on this cycle is $j$. But, cycle $C$ in $G$ induces a corresponding cycle $C'$ in $G'$ such that $\min_{v \in C'}\{\ell'(v)\}$ is even. Hence, the non-emptiness problem is positive.

Suppose that $G'$ contains a cycle $C'$ such that $\min_{v \in C'}\{\ell'(v)\}$ is even. Then, there must be a corresponding cycle $C$ of $G$ where the lowest indexed node on this cycle has label $\nu$. As all nodes appearing in block $B$ are reachable from the smallest numbered node, both conditions (2a) and (2b) of Lemma 33 hold. Thus, by Lemma 33, the least variable of $B$ holds.

Hence, solving a disjunctive block $B$ with $|V| = n$ and $|E| = m$ reduces in time $O(n + m)$ to the task of deciding the even-cycle problem for $G' = (\{1, 2, ..., 2n\}, V, E, \ell')$ which, by Lemma 39, can be solved in time $O((n + m) \log k)$. $\square$

In the very same way, a similar result can be proved for the dual case, and we thus have the following upper bound for solving a conjunctive block of a Boolean equation system.

**Theorem 41** *Let $B$ be a conjunctive block of a Boolean equation system and let $G = (V, E, \ell)$ be the dependency graph of $B$ with $|V| = n$ and $|E| = m$. The least variable of $B$ can be solved in time $O((n + m) \log n)$.*

**Proof:**
In the same way as for Theorem 40. □

We observe that a very similar linear-time reduction can be given to the other direction as well, namely from a parity word automaton to a disjunctive block of a Boolean equation system. It follows that, if one can find improved algorithms for disjunctive and conjunctive Boolean equation systems, then these algorithms could be effectively applied to check the non-emptiness of parity word automata too.

It seems that the reductions in Theorem 40 and Theorem 41 are not optimal, and could be even improved. In fact, a slightly more efficient algorithm to solve disjunctive and conjunctive Boolean equation systems is defined in [24]. This algorithm works in time $O(e \log ad)$ where $e$ is the size of the Boolean equation system and $ad$ its alternation depth. However, it remains open whether or not the reductions in this section could be improved to match the time complexity of the algorithm in [24].

Finally, it would be interesting to find out whether or not the presented theoretical improvement in Theorems 40 and 41 also leads to practical improvements over [23] and other existing algorithms for conjunctive and disjunctive Boolean equation systems. Unfortunately, to the best of our knowledge there do not exist any implementations of algorithms in [34] and [24]. Therefore, evaluation on practical verification problems, and empirical comparison between the related algorithms are left for future work.

# 6 GENERAL FORM BLOCKS

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 [38, 43, 47]. 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 discuss the correctness of our translation.

## 6.1 Solving General Blocks in Answer Set Programming

It was 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 was seen how the variables appearing in conjunctive and disjunctive blocks with alternation can be solved using only sub-quadratic time. So the remaining task is to solve alternating blocks containing both mutually dependent variables with different signs and arbitrary connectives as right-hand sides. The complexity of solving such general blocks is an important open problem; no polynomial time algorithm has been discovered. On the other hand, it is shown in [42] that the problem is in the complexity class NP $\cap$ co-NP (and is known to be even in UP $\cap$ co-UP).

Here, we present a technique to solve an alternating Boolean equation system which applies Lemma 33 from Section 5 and another useful property, Lemma 42 to be introduced in the following section. In particular, we reduce the resolution of alternating Boolean equation systems to the problem of computing stable models of logic programs by defining a translation from equation systems to normal logic programs.

We reduce the problem of solving alternating Boolean equation systems to computing stable models of normal logic programs. This is achieved by devising an alternative mapping from Boolean equation systems to normal logic programs so the solution for a given variable in an equation system can be determined by the existence of a stable model of the corresponding logic program. The results presented in this section were mainly reported in [32].

Before giving the translation we discuss some useful properties of general form Boolean equation systems.

## 6.2 Properties of General Boolean Equation System

The following observation forms the basis for our answer set programming based technique to solve general form Boolean equation systems with alternating fixed points.

From each Boolean equation system $\mathcal{E}$ containing both disjunctive and conjunctive equations we may construct a new Boolean equation system $\mathcal{E}'$, which is either in a disjunctive or in a conjunctive form. To obtain from $\mathcal{E}$ a disjunctive form system $\mathcal{E}'$, we remove in every conjunctive equation of $\mathcal{E}$

exactly one conjunct; otherwise the system $\mathcal{E}$ is unchanged. The dual case is similar.

For any standard form Boolean equation system having both disjunctive and conjunctive equations we have the following two properties.

**Lemma 42 (Lemma 2 of** [32]**)** *Let $\mathcal{E}$ be a standard form Boolean equation system. Then the following are equivalent:*

1. $[\![\mathcal{E}]\!] = 0$

2. *There is a disjunctive system $\mathcal{E}'$ with the solution $[\![\mathcal{E}']\!] = 0$ which can be constructed from $\mathcal{E}$.*

**Proof:**
We only show that (2) implies (1). The other direction can be proved by a similar argument and also follows directly from Proposition 3.36 in [42].

Define a *parity game* in the following way. Given a standard form Boolean equation system $\mathcal{E} = (\sigma_1 x_1 = \alpha_1), (\sigma_2 x_2 = \alpha_2), ..., (\sigma_n x_n = \alpha_n)$, we define a game $\Gamma_{\mathcal{E}} = (V, E, P, \sigma)$ where $V$ and $E$ are exactly like in the dependency graph of $\mathcal{E}$ and

- $P : V \to \{I, II\}$ is a player function assigning a player to each node; for $1 \leq i \leq n$, $P$ is defined by $P(i) = I$ if $\alpha_i$ is conjunctive and $P(i) = II$ otherwise.

- $\sigma : V \to \{\mu, \nu\}$ is a parity function assigning a sign to each node; for $1 \leq i \leq n$, $\sigma$ is defined by $\sigma(i) = \mu$ if $\sigma_i = \mu$ and $\sigma(i) = \nu$ otherwise.

A *play* on the game graph is an infinite sequence of nodes chosen by players $I$ and $II$. The play starts at node 1. Whenever a node $n$ is labelled with $P(n) = I$, it is player $I$'s turn to choose a successor of $n$. Similarly, if a node $n$ is labelled with $P(n) = II$, it is player $II$'s turn to choose a successor of $n$. A *strategy* for a player $i$ is a function which tells $i$ how to move at all decision nodes, i.e. a strategy is a function that assigns a successor node to each decision node belonging to player $i$. Player $I$ *wins* a play of the game if the smallest node that is visited infinitely often in the play is labelled with $\mu$, otherwise player $II$ wins. We say that a player has a *winning strategy* in a game whenever she wins all the plays of the game by using this strategy, no matter how the opponent moves. According to Theorem 8.7 in [42], player $II$ has a winning strategy for game on $\Gamma_{\mathcal{E}}$ with initial vertex 1 iff the solution of $\mathcal{E}$ is $[\![\mathcal{E}]\!] = 1$.

So suppose there is a conjunctive equation system $\mathcal{E}'$ obtained from $\mathcal{E}$ by removing exactly one disjunct from all equations of the form $\sigma_i x_i = x_j \vee x_k$ such that $[\![\mathcal{E}']\!] = 1$. We can construct from $\mathcal{E}'$ a winning strategy for player $II$ in the parity game $\Gamma_{\mathcal{E}}$. For all nodes $i$ of $\Gamma_{\mathcal{E}}$ where it is player $II$'s turn to move, define a strategy for $II$ to be $str_{II}(i) = j$ iff $\sigma_i x_i = x_j$ is an equation of $\mathcal{E}'$. That is, the strategy $str_{II}$ for $II$ is to choose in every $II$ labelled node of $\Gamma_{\mathcal{E}}$ the successor which appears also in the right-hand side expression of the $i$-th equation in $\mathcal{E}'$.

It is then straightforward to verify that for the game on $\Gamma_{\mathcal{E}}$ with initial node 1 player $II$ wins every play by playing according to $str_{II}$. By Lemma 33, the system $\mathcal{E}'$ does not contain any $\mu$ labelled variables that depend on $x_1$ and are self-dependent. The crucial observation is that the dependency graph of $\mathcal{E}'$ contains all and only those paths which correspond to the plays of the game $\Gamma_{\mathcal{E}}$ where the strategy $str_{II}$ is followed. Consequently, there cannot be a play of the game $\Gamma_{\mathcal{E}}$ starting from node 1 that is won by player $I$ and where player $II$ plays according to $str_{II}$. It follows from Theorem 8.7 in [42] that the solution of $\mathcal{E}$ is $[\![\mathcal{E}]\!] = 1$. □

**Lemma 43** *Let $\mathcal{E}$ be a standard form Boolean equation system. Then the following are equivalent:*

1. $[\![\mathcal{E}]\!] = 1$
2. *There is a conjunctive system $\mathcal{E}'$ with the solution $[\![\mathcal{E}']\!] = 1$ which can be constructed from $\mathcal{E}$.*

Let us illustrate the above lemmas with a simple example.

**Example 44** *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 6. There is only one conjunctive equation $\nu x_1 = x_2 \wedge x_1$, yielding two possible disjunctive Boolean equation systems 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 33, we can see that these disjunctive systems have the solutions $[\![\mathcal{E}_1']\!] = [\![\mathcal{E}_1'']\!] = 1$. By Lemma 42, 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 $\Pi(\mathcal{E})$ which captures the solution $[\![\mathcal{E}]\!]$ of $\mathcal{E}$. Suppose that 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 $\mathcal{E}$. The dual case goes along exactly the same lines and is omitted.[4] The idea is that $\Pi(\mathcal{E})$ is a ground program which is polynomial in the size of $\mathcal{E}$. We give a compact description of $\Pi(\mathcal{E})$ as a program with variables. This program consists of the rules

$$depends(1). \tag{9}$$
$$depends(Y) \leftarrow dep(X,Y), depends(X). \tag{10}$$
$$reached(X,Y) \leftarrow nu(X), dep(X,Y), Y \geq X. \tag{11}$$
$$reached(X,Y) \leftarrow reached(X,Z), dep(Z,Y), Y \geq X. \tag{12}$$
$$\leftarrow depends(Y), reached(Y,Y), nu(Y). \tag{13}$$

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

$$dep(i,j). \qquad \text{if } \alpha_i = x_j \tag{14}$$
$$dep(i,j).\ dep(i,k). \qquad \text{if } \alpha_i = (x_j \vee x_k) \tag{15}$$
$$1\ \{dep(i,j), dep(i,k)\}\ 1. \quad \text{if } \alpha_i = (x_j \wedge x_k) \tag{16}$$

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 $\Pi(\mathcal{E})$ 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 $x_i = (x_j \wedge x_k)$. Hence, each stable model corresponds to a disjunctive system constructed from $\mathcal{E}$ and vice versa.

The translation can be exemplified as follows.

**Example 45** *Recall the Boolean equation system $\mathcal{E}_1$ of Example 44. The program $\Pi(\mathcal{E}_1)$ consists of the rules 9-13 extended with rules:*

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

## 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 33 and 42

**Theorem 46** *Let $\mathcal{E}$ be a standard form, alternating Boolean equation system. Then $[\![\mathcal{E}]\!] = 0$ iff $\Pi(\mathcal{E})$ has a stable model.*

**Proof:**
Consider a system $\mathcal{E}$ and its translation $\Pi(\mathcal{E})$. The rules (14–16) 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 (14–16) and disjunctive systems that can be constructed from $\mathcal{E}$ such that for each stable model $\Delta$, there is exactly

[4] This is 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 $\mathcal{E}$.

one disjunctive system $\mathcal{E}'$ with the dependency graph $G_{\mathcal{E}'} = (V, E)$ where $V = \{i \mid dep(i,j) \in \Delta$ or $dep(j,i) \in \Delta\}$ and $E = \{(i,j) \mid dep(i,j) \in \Delta\}$.

Now it is straightforward to establish by the splitting set theorem [39] that each stable model $\Delta$ of $\Pi(\mathcal{E})$ is an extension of a stable model $\Delta'$ of the rules (14–16), 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 42 $[\![\mathcal{E}]\!] = 0$ iff there is a disjunctive system $\mathcal{E}'$ that can be constructed from $\mathcal{E}$ for which $[\![\mathcal{E}']\!] = 0$. By Lemma 33 for a disjunctive system $\mathcal{E}'$, $[\![\mathcal{E}']\!] = 1$ iff 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 property holds also for the dual program, which allows us to solve all alternating blocks of standard form Boolean equation systems.

Perhaps, further explanation of our translation is in order here. Although $\Pi(\mathcal{E})$ is given using variables, for the theorem above a finite ground instantiation of it is sufficient. For explaining the ground instantiation we introduce a relation *depDom* such that $depDom(i,j)$ holds iff there is an equation $\sigma_i x_i = \alpha_i$ of $\mathcal{E}$ with $x_j$ occurring in $\alpha_i$. Now the sufficient ground instantiation is obtained by substituting variables $X, Y$ in the rules (10–11) with all pairs $i, j$ such that $depDom(i,j)$ holds, substituting variables $X, Y, Z$ in rule (12) with all triples $l, i, j$ such that $nu(l)$ and $depDom(i,j)$ hold and variable $Y$ in rule (13) with every $i$ such that $nu(i)$ holds. This means also that such conditions can be added as domain predicates to the rules without compromising the correctness of the translation. For example, rule (12) could be replaced by

$$reached(X,Y) \leftarrow nu(X), depDom(Z,Y), reached(X,Z), dep(Z,Y), Y \geq X.$$

Notice that such conditions make the rules domain restricted as required, e.g., by the `Smodels` system.

Finally, it is worthwhile to observe that it is possible to construct a mapping from Boolean equation systems to propositional satisfiability as well (for a description of propositional satisfiability problem, see p. 77 in [49]). In principle, this would give a way to solve Boolean equation systems with various propositional satisfiability checkers.

However, if one uses the above translation from general form equation systems to normal logic programs as a basis for such a mapping, it will not straightforwardly lead to compact encodings. One of the reasons for this is the fact that there are difficulties to encode predicates like $depends(Y)$ and $reached(X,Y)$ (see rules 9-13) with short propositional formulas.

In the next section, we will describe some experimental results on solving alternating Boolean equation systems with the approach presented in this section. We will demonstrate the technique on a series of examples which are solved using the `Smodels` system as the ASP solver.

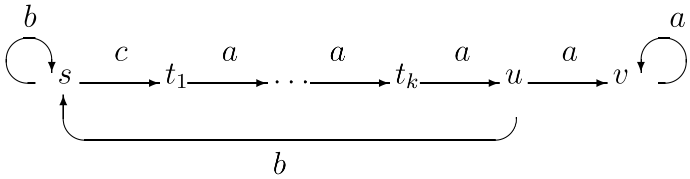



Figure 6: Process $M_k$ from [40, 53] for model checking the properties $\phi_1$ and $\phi_2$.

# 7 CASE STUDIES

In this section, we discuss some case studies using the Boolean equation system solution methods proposed in this report. Two of these case studies used a solver for conjunctive and disjunctive Boolean equation systems to tackle $\mu$-calculus model checking problems. This solver is an implementation of the solution algorithm reported in [23] which is also described in Section 5.2. In the last case studies we used Smodels answer set programming system to solve instances of general form Boolean equation systems with alternation. We used the logic programming encoding reported in [32] which is also described in Section 6.

## 7.1 Model Checking Examples

In this section, we describe an implementation of the solution algorithm presented in [23] and discussed in Section 5.2. This prototype solver for alternating conjunctive and disjunctive form Boolean equation systems is implemented in the C programming language [33]. To give an impression of the performance, we report experimental results on solving two verification problems using the tool.

As our first benchmarks we used two sets of $\mu$-calculus model checking problems borrowed from [40] and [53], converted to Boolean equation systems. The verification problems consist of checking $\mu$-calculus formulas of alternation depth 2, on a sequence of regular labelled transition systems $M_k$ of increasing size (see figure 6).

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 alternating fixed-point formula:

$$\phi_1 \equiv \nu X.\mu Y.([b]X \wedge [-b]Y) \tag{17}$$

The property is false at state $s$ and we use the solver to find a counter-example for the formula.

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 alternating fixed point formula

$$\phi_2 \equiv \nu X.\mu Y.(\langle a\rangle X \vee \langle -a\rangle Y) \tag{18}$$

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

The problems of determining whether the system $M_k$ satisfies the specifications $\phi_1$ and $\phi_2$ 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 $\phi_1$ using the standard translation [1, 4, 42] from $\mu$-calculus to Boolean equation systems (see Figure 3). If we consider a labelled transition system $M_k = (S, A, \longrightarrow)$ in Figure 6 then the Boolean equation system looks like:

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

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 solver 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 time for the solver to find the local solutions measured as system time, on a 2.4Ghz Intel Xeon running Linux (i.e. the times for the solver to read the equation systems from disk and build the internal data structures are excluded).

The experimental results are given in Figure 7. The columns are explained below:

- Problem:
  - the process $M_k$, with $k + 3$ states
  - $\phi_1$ the formula 17 to be checked
  - $\phi_2$ the formula 18 to be checked
- $n$: the number of equations in the Boolean equation system corresponding to the model checking problem
- Time: the time in seconds to find the local solution

In the problem with the property $\phi_1$, the solver found solutions even without executing the quadratic part of the algorithm. In the problem with property $\phi_2$, the quadratic computation needed to be performed only on very small portions of the equation systems. These facts are reflected in the performance of the solver, which exhibits linear growth in the execution times with increase in the size of the systems to be verified, in all of the experiments.

The benchmarks in [40] and [53] have a quite simple structure, and therefore we must be careful in drawing general results from them. A more involved practical evaluation is desirable here.

In the next section, we provide initial experimental results on Boolean equation system benchmarks from more realistic applications in the domain of protocol verification.

| Problem | | n | Time (sec) |
|---|---|---|---|
| $M_{5000000}$ | $\phi_1$ | 10 000 006 | 2.6 |
| | $\phi_2$ | 10 000 006 | 3.0 |
| $M_{10000000}$ | $\phi_1$ | 20 000 006 | 5.5 |
| | $\phi_2$ | 20 000 006 | 6.4 |
| $M_{15000000}$ | $\phi_1$ | 30 000 006 | 7.5 |
| | $\phi_2$ | 30 000 006 | 9.0 |

Figure 7: Summary of execution times.

### 7.2 Model Checking DKR Leader Election Protocol

As second set of benchmarks, we used protocol models taken from [7], instantiated with the $\mu CRL$-tool set [6], and converted to Boolean equation systems. The verification problem consists of model checking a $\mu$-calculus formula on a sequence of distributed leader election protocol models of increasing size, as described below.

In brief, the Dolev-Klawe-Rodeh (DKR) leader election protocol [16] consists of $n$ parties connected in a ring. These parties exchange messages and after a finite number of messages, the party with the highest identification informs the others being a leader. The protocol is modelled in [21] and the desired safety property we need to check is that "Two *leader*-actions can never occur on a same execution path of the system". This specification is expressed with a $\mu$-calculus formula (19) below

$$\nu X.([true]X \wedge [leader]\phi) \tag{19}$$

where the subformula $\phi$ is (20):

$$\nu Y.([true]Y \wedge [leader]false) \tag{20}$$

The formula (19) is globally true on all protocol models.

If we consider the labelled transition systems $M = (S, A, \longrightarrow)$ from [7], then the above verification problem can be directly formalized as a Boolean equation system prompting us to encode it as follows:

$$\left.\begin{array}{l} \nu\, x_s = \bigwedge\limits_{s' \in \nabla(t,s)} x_{s'} \wedge \bigwedge\limits_{s' \in \nabla(l,s)} y_{s'} \\ \nu\, y_s = \bigwedge\limits_{s' \in \nabla(l,s)} false \wedge \bigwedge\limits_{s' \in \nabla(t,s)} y_{s'} \end{array}\right\} \forall s \in S$$

Here, $\nabla(l, s) := \{s' | s \overset{leader}{\longrightarrow} s'\}$ and $\nabla(t, s) := \{s' | s \overset{i}{\longrightarrow} s' \text{ and } i \in A\}$.

We then evaluated the performance of our algorithm from [23], which was also discussed in Section 5.2, by measuring solution times. The testing was done on a 1.0Ghz AMD Athlon running Linux with sufficient main memory, and the times reported in this section are the average of 3 runs of the times for the solvers to find solutions as reported by the */usr/bin/time* command

| Benchmark | $\lvert\mathcal{E}\rvert$ | Time (sec) |
|---|---|---|
| DKR(5) | 7 192 | 0.0 |
| DKR(6) | 36 714 | 0.1 |
| DKR(7) | 190 618 | 0.2 |
| DKR(8) | 632 284 | 0.8 |
| DKR(9) | 3 162 712 | 4.2 |

Figure 8: Total solution times on DKR benchmark data.

(the times for the solver to read the equation systems from disk and build the internal data structures are included).

The results are given in Fig. 8 where the columns are:

- Benchmark: DKR(n) DKR leader election protocol model with $n$ parties;
- $\lvert\mathcal{E}\rvert$: the size of the Boolean equation system corresponding to the verification problem;
- Time(sec): The time in seconds needed to solve the equation system.

Previously, the above results were reported in [31]. It would have been interesting to compare the performance of our algorithm to other Boolean equation system solvers, like for instance those from [45]. Unfortunately, we were not able to conduct such comparisons as we did not find any publicly available implementations of other Boolean equation system solvers.

## 7.3 Solving Alternating Boolean Equation Systems

In this section, we describe some experimental results on solving alternating Boolean equation systems with the approach presented in Section 6. We demonstrate the technique on a 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.

Once again, we were not able to compare this method to other approaches as we did not find any implementation capable of handling Boolean equation systems with high alternation depths. We also experimented with another ASP system, `DLV` (release 2004–05–23 available on `http://www.dbai.tuwien.ac.at/proj/dlv/`), as the underlying ASP solver but ran into performance problems as explained below. We used `Smodels` version 2.26 to find the solutions and the time needed for parsing and grounding the input with `lparse` 1.0.13 is included.

The encoding used for the benchmarks is that represented in Section 6.3 with a couple of optimizations. Firstly, when encoding of dependencies as given in rules (14–16) we differentiate those dependencies where there is a

$$\left.\begin{array}{l}
\nu\, x_1 = x_2 \wedge x_n \\
\mu\, x_2 = x_1 \vee x_n \\
\nu\, x_3 = x_2 \wedge x_n \\
\mu\, x_4 = x_3 \vee x_n \\
\ldots \\
\nu\, x_{n-3} = x_{n-4} \wedge x_n \\
\mu\, x_{n-2} = x_{n-3} \vee x_n \\
\nu\, x_{n-1} = x_{n-2} \wedge x_n \\
\mu\, x_n = x_{n-1} \vee x_{n/2}
\end{array}\right\} \text{for } n \in 2\mathbb{N}$$

Figure 9: The Boolean equation system in [42, p.91].

choice from those where there is not, i.e., for each equation $\sigma_i x_i = \alpha_i$ of $\mathcal{E}$ we add

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

instead of rules (14–16). Secondly, in order to make use of this distinction and to allow for intelligent grounding, rules (10–12) are rewritten using the above predicates as domain predicates in the following way.

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

All benchmark encodings are available at:
`http://www.tcs.hut.fi/Software/smodels/tests/inap2004.tar.gz`.

The experiments deal with solving alternating Boolean equation systems of increasing size and alternation depth. The problem is taken from [42, p.91] and consists of finding the solution to the left-most variable $x_1$ of the Boolean equation system depicted in Fig. 9.

The example is such that a Boolean equation system in Fig. 9 with $n$ equations has the alternation depth $n$. The solution to the system is such that $[\![\mathcal{E}]\!] = 1$ which can be obtained by determining the existence of a stable model of the corresponding logic program.

The times reported in this section are the average of 3 runs of the time for `Smodels` 2.26 to find the solutions as reported by the */usr/bin/time* command on a 2.0Ghz AMD Athlon running Linux. The time needed for parsing and grounding the input with `lparse` 1.0.13 is included. The experimental results are summarised in Fig. 10.

| Problem (n) | Time (sec) |
|---|---|
| 1800 | 33.6 |
| 2000 | 41.8 |
| 2200 | 51.4 |
| 2400 | 60.0 |
| 2600 | 71.7 |

Figure 10: The experimental results on the Boolean equation system in Fig. 9.

Our benchmarks are essentially the only results in the literature for alternating Boolean equation systems with the alternation depth $n \geq 4$ of which we are aware. Notice that our benchmarks have the alternation depths $1800 \leq n \leq 2600$.

Like pointed out in [42], typical solution algorithms take exponential time in the size of the equation system in Fig. 9, because a maximal number of backtracking steps is always needed to solve the left-most equation.

We tried to use also `DLV` as the underlying ASP solver on this benchmark, but found that it does not scale as well as `Smodels` when the size $n$ of the problem grows. For example, for size $n = 1800$ the running time for `DLV` was over 30 minutes.

# 8 CONCLUSION

Boolean equation systems give a useful formalism to encode various problems encountered in a wide range of problem domains. For instance, these domains include propositional logic programming (e.g. Horn clause satisfiability [41]), automatic program analysis (e.g. abstract interpretation of functional and logic programming languages [20]), and verification of concurrent systems (e.g. equivalence checking [2, 13, 37, 45], model checking [1, 23, 32, 46], partial order reduction [48]). Boolean equation system solvers can be used as general purpose tools targeted to handle these kinds of problems.

In this report, we develop an environment to solve Boolean equation systems. Namely, we present a framework which allows for considerably optimizing the solver by taking advantage of many basic properties and features of Boolean equation systems. We show how to solve various classes of Boolean equation systems efficiently, and show how these different methods can be combined into a single framework.

We discuss implementations as well as experiments with the proposed solution techniques. Also, we demonstrate how the approach of the report can be applied to solve model checking problems by discussing several case studies.

There is still room for further research on Boolean equation systems. For instance, one might consider the following directions.

In Section 5.3, we give the proof of the complexity of solving conjunctive and disjunctive Boolean equation systems without providing any implementation. The algorithm from [24] should be implemented, and its performance should be evaluated on practical, real-life problems. Also, the approach from [24] should be compared to other related algorithms. It will be interesting to see whether the complexity of solving conjunctive and disjunctive classes can be further improved.

In Section 7.3, we provide a proof of concept implementation of the answer set programming approach to solve general form, alternating Boolean equation system. A full-blown Boolean equation system solver for general systems, which is based on this approach, needs to be implemented. In addition, a more involved practical evaluation, and a comparison with the related methods and tools (e.g. the one from [58, 50]) would be highly desirable.

Also, an interesting open question is whether or not there exist compact encodings of Boolean equation systems as propositional satisfiability. Our translation from Boolean equation systems to normal logic programs, together with the results in [29], gives a polynomial reduction from Boolean equation systems to propositional satisfiability. It remains to be shown whether this translation can be improved.

Recently, there has been an increasing interest to the study of various extensions of Boolean equation systems. These extensions include parameterized Boolean equation systems [25, 26]. Some of the results in this report can well be extended to parameterized systems as well, and automated tool support for solving such extended Boolean equation systems should be developed.

## Acknowledgements

This report is a reprint of my Licentiate's thesis. First of all, I would like to thank Professor Ilkka Niemelä for providing the possibility to join his research group at Laboratory for Theoretical Computer Science, Helsinki University of Technology. Prof. Niemelä first introduced me to answer set programming and has taught me a lot. Also, I thank D.Sc. (Tech.) Keijo Heljanko for suggesting to study $\mu$-calculus model checking from the answer set programming perspective. I am grateful to Prof. Niemelä and Dr. Heljanko for many discussions. They both read some drafts of this report and patiently made several helpful remarks.

I thank Professor Jan Friso Groote for various constructive ideas as well as always expressing encouraging attitude to my work. I am greatly indebted to Professor Wan Fokkink for the possibility to visit his research group at CWI, in the Netherlands. Thanks to Prof. Fokkink, Dr. Jaco van de Pol and others for discussions and the supporting research environment at the CWI. I want to thank Professor Angelika Mader for discussion and useful information about Boolean equation systems.

The financial supports of Helsinki Graduate School in Computer Science and Engineering, the Academy of Finland (project 53695), and the Emil Aaltonen Foundation are gratefully acknowledged.

Especially, I want to express my gratitude to her with whom I have just started a project much more important and challenging than this report.

## 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 $\mu$-calculus. *Studies in logic and the foundations of mathematics*, Volume 146, Elsevier, 2001.

[5] G. Bhat and R. Cleaveland. Efficient local model-checking for fragments of the modal $\mu$-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.

[6] S. Blom, W. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser and J. van de Pol. $\mu$CRL: 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.

[7] 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.

[8] J. Bradfield. The modal $\mu$-calculus alternation hierarchy is strict. *Theoretical Computer Science*, 195:133–153, 1998.

[9] 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.

[10] 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.

[11] E. Clarke, O. Grumberg and D. Peled. Model Checking. The MIT Press, 2000.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] E.A. Emerson, C. Jutla and A.P. Sistla. On model checking for fragments of the $\mu$-calculus. In *Proceedings of the Fifth International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 697, pages 385–396, Springer Verlag, 1993.

[19] E.A. Emerson, C. Jutla, and A.P. Sistla. On model checking for the $\mu$-calculus and its fragments. *Theoretical Computer Science*, 258:491–522, 2001.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] J.F. Groote and M. Keinänen. A Sub-quadratic Algorithm for Conjunctive and Disjunctive Boolean Equation Systems. *Computer Science Report 04/13, Department of Mathematics and Computer Science*, Eindhoven University of Technology, 2004.

[25] J.F. Groote and T. Willemse. A Checker for Modal Formulas for Processes with Data. *Computer Science Report 02-16, Department of Mathematics and Computer Science*, Eindhoven University of Technology, 2002. To appear in Science of Computer Programming, Elsevier.

[26] 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.

[27] 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.

[28] M. Hennessy and R. Milner. Algebraic laws for non-determinism and concurrency. *Journal of the ACM*, 32(1): 137–161, 1985.

[29] 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.

[30] M. Jurdzinski. Deciding the winner in parity games is in $UP \cap co-UP$. *Information Processing Letters*, 68:119–124, 1998.

[31] M. Keinänen. Obtaining Memory Efficient Solutions to Boolean Equation Systems. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'2004)*. To appear in Electronic Notes in Theoretical Computer Science, Elsevier.

[32] 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.

[33] B. Kernighan and D. Rithchie. The C Programming Language. Prentice Hall PTR, 1988.

[34] 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.

[35] D. Kozen. Results on the propositional $\mu$-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[36] 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.

[37] 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.

[38] V. Lifschitz. Answer Set Planning. In *Proceedings of the 16th International Conference on Logic Programming*, pages 25–37, The MIT Press, 1999.

[39] V. Lifschitz and H. Turner. Splitting a Logic Program. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 23–37, The MIT Press, 1994.

[40] 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.

[41] 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.

[42] A. Mader. Verification of Modal Properties using Boolean Equation Systems. PhD thesis, Technical University of Munich, 1997.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] G. Pace, F. Lang, and R. Mateescu. Calculating $\tau$-Confluence Compositionally. In *Proceedings of Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2725, pages 446–459, Springer Verlag, 2003.

[49] C. Papadimitriou. Computational Complexity. Addison-Wesley, 1994.

[50] 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.

[51] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[52] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[53] 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.

[54] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[55] R.E. Tarjan. A hierarchical clustering algorithm using strong components. *Information Processing Letters*, 14(1):26–29, 1982.

[56] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[57] 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.

[58] 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.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

| | |
|---|---|
| HUT-TCS-A86 | Tommi Syrjänen<br>Logic Programming with Cardinality Constraints. December 2003. |
| HUT-TCS-A87 | Harri Haanpää, Patric R. J. Östergård<br>Sets in Abelian Groups with Distinct Sums of Pairs. February 2004. |
| HUT-TCS-A88 | Harri Haanpää<br>Minimum Sum and Difference Covers of Abelian Groups. February 2004. |
| HUT-TCS-A89 | Harri Haanpää<br>Constructing Certain Combinatorial Structures by Computational Methods. February 2004. |
| HUT-TCS-A90 | Matti Järvisalo<br>Proof Complexity of Cut-Based Tableaux for Boolean Circuit Satisfiability Checking. March 2004. |
| HUT-TCS-A91 | Mikko Särelä<br>Measuring the Effects of Mobility on Reactive Ad Hoc Routing Protocols. May 2004. |
| HUT-TCS-A92 | Timo Latvala, Armin Biere, Keijo Heljanko, Tommi Junttila<br>Simple Bounded LTL Model Checking. July 2004. |
| HUT-TCS-A93 | Tuomo Pyhälä<br>Specification-Based Test Selection in Formal Conformance Testing. August 2004. |
| HUT-TCS-A94 | Petteri Kaski<br>Algorithms for Classification of Combinatorial Objects. June 2005. |
| HUT-TCS-A95 | Timo Latvala<br>Automata-Theoretic and Bounded Model Checking for Linear Temporal Logic. August 2005. |
| HUT-TCS-A96 | Heikki Tauriainen<br>A Note on the Worst-Case Memory Requirements of Generalized Nested Depth-First Search. September 2005. |
| HUT-TCS-A97 | Toni Jussila<br>On Bounded Model Checking of Asynchronous Systems. October 2005. |
| HUT-TCS-A98 | Antti Autere<br>Extensions and Applications of the $A^*$ Algorithm. November 2005. |
| HUT-TCS-A99 | Misa Keinänen<br>Solving Boolean Equation Systems. November 2005. |

ISBN 951-22-7994-0
ISSN 1457-7615