Solving Alternating Boolean Equation Systems in Answer Set Programming

Misa Keinänen and Ilkka Niemelä

Dept. of Computer Science and Engineering, Lab. for Theoretical Comp. Science
Helsinki University of Technology, P.O. Box 5400, 02015 HUT, Finland
`Misa.Keinanen@hut.fi`, `Ilkka.Niemela@hut.fi`

Abstract. In this paper we apply answer set programming to solve alternating Boolean equation systems. We develop a novel characterization of solutions for variables in disjunctive and conjunctive Boolean equation systems. Based on this we devise a mapping from Boolean equation systems with alternating fixed points to normal logic programs such that the solution of a given variable of an equation system can be determined by the existence of a stable model of the corresponding logic program. The technique can be used to model check alternating formulas of modal μ-calculus.

1 Introduction

Model checking is a verification technique aimed at determining whether a system model satisfies desired properties expressed as temporal logic formulas. In recent years, research on model checking has addressed large scale verification problems, which are often solved by special purpose verification tools.

Yet it has been demonstrated that also logic programming systems can successively be applied to the construction of practical model checkers, like e.g. in [10, 5, 13]. In the present paper, we continue this line of research and restrict the attention to the model checking problem of modal μ-calculus [12], and in particular to its formulation as *Boolean equation systems* [1, 18, 23]. The research topic belongs to the area of formal verification, but more specifically it addresses effective ways of solving systems of fixed point equations.

The modal *μ-calculus* is an expressive logic for systems verification, and has been widely studied in the recent model checking literature (e.g. [3] gives a general exposition). Boolean equation systems provide here a useful framework, because μ-calculus expressions can easily be translated into this more flexible formalism (see [1, 3, 18] for the standard translations). The complexity of μ-calculus model checking is an important open problem; no polynomial time algorithm has been discovered. On the other hand, it is shown in [6, 7] that the problem is in the complexity class NP $\cap$ co-NP (and is known to be even in UP $\cap$ co-UP [11], where UP is the class of problems decided by unambiguous polynomial time nondeterministic Turing machines, see [21]). In theory, the problem appears thus to be solvable with any answer set programming system capable of handling NP-complete problems.

In this paper we propose an answer set programming (ASP) based approach for solving alternating 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 [14, 19, 20]. We develop such a mapping from alternating Boolean equation systems to logic programs providing a basis for a model checking technique for μ-calculus logic.

Previously, answer set programming has been applied to solve Boolean equation systems in [13] where it is argued that alternating Boolean equation systems can be solved by computing certain *preferred stable models* of propositional normal logic programs corresponding to Boolean equation systems. Moreover, it is shown in [13] how alternation-free Boolean equation systems can be mapped to stratified logic programs, which can be directly solved in linear time, preserving the complexity [2] of model checking alternation-free fragment of μ-calculus. However, the approach proposed in [13] does not preserve the polynomial time complexity [9] of solving disjunctive and conjunctive Boolean equation systems.

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. Our translation is such that it ensures polynomial time complexity of solving both disjunctive and conjunctive alternating systems, and hence preserves the complexity of model checking many important fragments of μ-calculus, like $L1$ and $L2$ investigated in [4, 6, 7].

The paper is organized as follows. In the following section we introduce basic notions of Boolean equation systems. In Section 3 we state some properties of Boolean equation systems which are important in solving them. In Section 4 we review stable model semantics of normal logic programs. In Section 5 we show how alternating Boolean equation systems can be solved using answer set programming techniques. In Section 6 we discuss some initial experimental results. Finally, Section 7 contains conclusive remarks.

2 Boolean Equation Systems

We will give in this section 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, μ and ν, specifying the polarity of the fixed points. The equations are of the form $\sigma x = \alpha$, where α is a positive Boolean expression. The sign, σ, is μ if the equation is a least fixed point equation and ν if it is a greatest fixed point equation.

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 given by the grammar:

$$\alpha ::= \ 0 \mid 1 \mid x \in \mathcal{X} \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2$$

where 0 stands for $false$ and 1 for $true$. 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*

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

where the left-hand sides of the equations are all different. We assume that the order on variables and equations are in synchrony, and that all right-hand side variables are from $\mathcal{X}$.

The semantical interpretation of Boolean equation systems is such that each system $\mathcal{E}$ has a uniquely determined solution, which is a valuation assigning a constant value in $\{0, 1\}$ to variables occurring in $\mathcal{E}$. 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, 18]). In particular, we are interested in the value of the left-most variable in the solution of a Boolean equation system. Such a local solution can be characterized in the following way.

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

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

Definition 2 (The local solution of a Boolean equation system). *The solution a Boolean equation system $\mathcal{E}$, denoted by $[\![\mathcal{E}]\!]$, is a Boolean value inductively defined by*

$$[\![\sigma x = \alpha]\!] = \|\alpha[x/b_\sigma]\|$$
$$[\![\mathcal{E}(\sigma x = \alpha)]\!] = [\![\mathcal{E}[x/\alpha[x/b_\sigma]]]\!]$$

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

The following example illustrates the definition of the solution.

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

3 Properties of Boolean Equation Systems

In this section, we discuss important concepts concerning Boolean equation systems. We also state some facts about Boolean equation systems, which turn out to be useful in the computation of their solutions.

The *size* of a Boolean equation system is inductively defined as $|\epsilon| = 0$ and $|(\sigma x = \alpha)\mathcal{E}| = 1 + |\alpha| + |\mathcal{E}|$, where $|\alpha|$ is the number of variables and constants in α.

A Boolean equation system $\mathcal{E}$ is in *standard form* if each right-hand side expression α_i consists of a disjunction $x_i \vee x_j$, a conjunction $x_i \wedge x_j$, or a single variable x_i. As pointed out in [18], for each system $\mathcal{E}$ there is another system $\mathcal{E}'$ in *standard form* such that $\mathcal{E}'$ preserves the solution of $\mathcal{E}$ and has size linear in the size of $\mathcal{E}$. In the sequel we restrict to standard form Boolean equation systems.

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

Definition 3 (A 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 $G_{\mathcal{E}} = (V, E)$ 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 appears in α_i.*

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. 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*.

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

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

We say that a variable x_k is *self-dependent* in the system $\mathcal{E}$, if x_k depends on itself on the graph $G{\restriction}k$.

Example 2. Consider the Boolean equation system $\mathcal{E}_1$ of Example 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 $\sigma_1 \neq \sigma_2$. The variables x_1 and x_3 of $\mathcal{E}_1$ are self-dependent.

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. 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. For example, the system $\mathcal{E}_1$ of Example 1 can be divided in 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. In Mader [18], there are two useful lemmas (Lemma 6.2 and Lemma 6.3) which allow us to solve all blocks of standard form Boolean equation systems one at a time. The basic idea is that we start by solving blocks that do not depend on any other block. For each solved block we can substitute its solution to blocks depending on it and thus iteratively solve them. Alternation-free blocks of standard form Boolean equation systems can be trivially solved in linear time in the size of the blocks [2]. Thus, we focus here on devising a technique to solve an alternating block of standard form Boolean equations, for which no polynomial time solution technique is known.

We call an equation $\sigma_i x_i = \alpha_i$ *disjunctive* if its right-hand side α_i is a disjunction. A standard form Boolean equation system is said to be *disjunctive* if all its equations $\sigma_i x_i = \alpha_i$ are either *disjunctive* or $\alpha_i \in \mathcal{X}$. Similarly, a Boolean equation $\sigma_i x_i = \alpha_i$ is *conjunctive* if its right-hand side α_i is a conjunction. A standard form Boolean equation system is *conjunctive* if all its equations $\sigma_i x_i = \alpha_i$ are *conjunctive* or $\alpha_i \in \mathcal{X}$.

The following lemmas form the basis for our answer set programming based technique to solve standard form Boolean equation systems with alternating fixed points. For a disjunctive (conjunctive respectively) form Boolean equation systems we have:

Lemma 1 (Lemma 4.2 of [9]). *Let $\mathcal{E}$ be a disjunctive (conjunctive) Boolean equation system in standard form. Then the following are equivalent:*

1. $[\![\mathcal{E}]\!] = 1$ *(or $[\![\mathcal{E}]\!] = 0$ respectively)*
2. *There is a variable x_j in $\mathcal{E}$ such that $\sigma_j = \nu$ ($\sigma_j = \mu$) and:*
   (a) x_1 depends on x_j, and (b) x_j is self-dependent.

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:

Lemma 2. *Let $\mathcal{E}$ be a standard form Boolean equation system. Then the following are equivalent:*

1. $[\![\mathcal{E}]\!] = 0$ *(or $[\![\mathcal{E}]\!] = 1$ respectively)*
2. *There is a disjunctive (conjunctive) system $\mathcal{E}'$ with the solution*
   $[\![\mathcal{E}']\!] = 0$ *($[\![\mathcal{E}']\!] = 1$ respectively) which can be constructed from $\mathcal{E}$.*

Proof. We only show that (2) implies (1) for the conjunctive case. The other direction can be proved by a similar argument and also follows directly from Proposition 3.36 in [18].

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 α_i is conjunctive and $P(i) = II$ otherwise.
- $\sigma : V \to \{\mu, \nu\}$ is a parity function assigning a sign to each node; for $1 \leq i \leq n$, σ 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 μ, 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 [18], 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 1, the system $\mathcal{E}'$ does not contain any μ 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 [18] that the solution of $\mathcal{E}$ is $[\![\mathcal{E}]\!] = 1$. $\square$

Example 3. Recall the Boolean equation system $\mathcal{E}_1$ of Example 1. 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 Lemma 1, we can see that these disjunctive systems have the solutions $[\![\mathcal{E}_1']\!] = [\![\mathcal{E}_1'']\!] = 1$. By Lemma 2, a solution to $\mathcal{E}_1$ is $[\![\mathcal{E}_1]\!] = 1$ as expected.

In Section 5 we will see the application of the above lemmas to give a compact encoding of the problem of solving alternating Boolean equation systems as the problem of finding certain stable models of normal logic programs.

4 Stable Models of Normal Logic Programs

For encoding Boolean equation systems we use normal logic programs with the stable model semantics [8]. A normal rule is of the form

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

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

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*. For a program Π and a set of atoms Δ, the reduct Π^Δ is defined by

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

i.e., a reduct Π^Δ does not contain any negative literals and, hence, has a unique subset minimal set of atoms satisfying it.

Definition 4. *A set of atoms Δ is a stable model of a program Π iff Δ is the unique minimal set of atoms satisfying Π^Δ.*

We employ two extensions which can be seen as compact shorthands for normal rules. We use *integrity constraints*, i.e., rules

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

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

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

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

For expressing the choice of selecting exactly one atom from two possibilities we use *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 [22]. The `Smodels` system (`http://www.tcs.hut.fi/Software/smodels/`) provides an implementation for cardinality constraint rules and includes primitives supporting directly such constraints without translating them first to corresponding normal rules.

5 Solving Boolean Equation Systems in ASP

The overall idea of our approach is as follows. Given a standard form Boolean equation system $\mathcal{E}$, we partition its variables into blocks so that variables are in the same block iff they are mutually dependent. The partition can be constructed in linear time on the basis of the dependencies between the variables. Like argued in Section 3, the variables can be solved iteratively one block at a time.

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 (see e.g. [2, 17]). So we only need to concentrate on solving alternating blocks containing mutually dependent variables with different signs. Consequently, we present here a technique to solve an alternating Boolean equation system which applies Lemmas 1-2 from Section 3.

In order to reduce the resolution of alternating Boolean equation systems to the problem of computing stable models of logic programs we define a translation from equation systems to normal logic programs. Consider a standard form, alternating Boolean equation system $\mathcal{E}$. 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.[1] 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

$$solve(1). \tag{3}$$

$$depends(Y) \leftarrow dep(X, Y), solve(X). \tag{4}$$

$$depends(Y) \leftarrow depends(X), dep(X, Y). \tag{5}$$

[1] 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 $\mathcal{E}$.

$$reached(X,Y) \leftarrow nu(X), dep(X,Y), Y \geq X. \tag{6}$$
$$reached(X,Y) \leftarrow reached(X,Z), dep(Z,Y), Y \geq X. \tag{7}$$
$$\leftarrow depends(Y), reached(Y,Y), nu(Y). \tag{8}$$

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

$$dep(i,j).\text{, if } \alpha_i = x_j \tag{9}$$
$$dep(i,j).\text{ and } dep(i,k).\text{, if } \alpha_i = (x_j \vee x_k) \tag{10}$$
$$1\ \{dep(i,j), dep(i,k)\}\ 1.\text{, if } \alpha_i = (x_j \wedge x_k) \tag{11}$$

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

Example 4. Recall the Boolean equation system $\mathcal{E}_1$ of Example 1. The program $\Pi(\mathcal{E}_1)$ consists of the rules 3-8 extended with rules:
$1\ \{dep(1,2), dep(1,1)\}\ 1.\ nu(1).$
$dep(2,1).\ dep(2,3).$
$dep(3,3).\ nu(3).$

The idea 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. Then by Lemmas 1 and 2 the main result can be established.

Theorem 1. *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 (9–11) 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 (9–11) and disjunctive systems that can be constructed from $\mathcal{E}$ such that for each stable model Δ, there is exactly one disjunctive system $\mathcal{E}'$ with the dependency graph $G_{\mathcal{E}'} = (V, E)$ where $V = \{i \mid dep(i,j) \in \Delta \text{ or } dep(j,i) \in \Delta\}$ and $E = \{(i,j) \mid dep(i,j) \in \Delta\}$.

Now it is straightforward to establish by the splitting set theorem [15] that each stable model Δ of $\Pi(\mathcal{E})$ is an extension of a stable model Δ' of the rules (9–11), 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 2 $[\![\mathcal{E}]\!] = 0$ iff there is a disjunctive system $\mathcal{E}'$ that can be constructed from $\mathcal{E}$ for which $[\![\mathcal{E}']\!] = 0$. By Lemma 1 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 $[\![\mathcal{E}]\!] = 0$. $\square$

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

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 α_i. Now the sufficient ground instantiation is obtained by substituting variables X, Y in the rules (4–6) with all pairs i, j such that $depDom(i, j)$ holds, substituting variables X, Y, Z in rule (7) with all triples l, i, j such that $nu(l)$ and $depDom(i, j)$ hold and variable Y in rule (8) 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 (7) 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.

6 Experiments

In this section, we describe some experimental results on solving alternating Boolean equation systems with the approach presented in the previous section. We demonstrate the technique on two series of examples. The times reported 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 encoding used for the benchmarks is that represented in Section 5 with a couple of optimizations. Firstly, when encoding of dependencies as given in rules (9–11) we differentiate those dependencies where there is a choice from those where there is not, i.e., for each equation $\sigma_i x_i = \alpha_i$ of $\mathcal{E}$ we add

$ddep(i, j).$, if $\alpha_i = x_j$

$ddep(i, j).\ ddep(i, k).$, if $\alpha_i = (x_j \vee x_k)$

$1\ \{cdep(i, j), cdep(i, k)\}\ 1.\ \ depDom(i, j).\ \ depDom(i, k).$, if $\alpha_i = (x_j \wedge x_k)$

instead of rules (9–11). Secondly, in order to make use of this distinction and to allow for intelligent grounding, rules (4–7) are rewritten using the above predicates as domain predicates in the following way.

$$depends(Y) \leftarrow ddep(X, Y), solve(X).$$
$$depends(Y) \leftarrow depDom(X, Y), cdep(X, Y), solve(X).$$
$$depends(Y) \leftarrow depends(X), ddep(X, Y).$$
$$depends(Y) \leftarrow depends(X), depDom(X, Y), cdep(X, Y).$$
$$reached(X, Y) \leftarrow nu(X), ddep(X, Y), Y \geq X.$$
$$reached(X, Y) \leftarrow nu(X), depDom(X, Y), cdep(X, Y), Y \geq X.$$

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

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

Fig. 1. The Boolean equation system in [18, p.91] and experimental results.

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

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

The first series deals with solving alternating Boolean equation systems of increasing size and alternation depth. The problem is taken from [18, p.91] and consists of finding the solution of the left-most variable x_1 in Fig. 1. The example is such that a Boolean equation system with n equations has the alternation depth n. The solution of 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 experimental results are summarised in Fig. 1. 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 [18], known algorithms based on approximation techniques are exponential in the size of the equation system in Fig. 1, because a maximal number of backtracking steps is always needed to solve the left-most equation.

In the second series of examples we used a set of μ-calculus model checking problems taken from [16], converted to alternating Boolean equation systems. The problems consist of checking a μ-calculus formula of alternation depth 2, on a sequence of models $M = (S, A, \longrightarrow)$ of increasing size (see Fig. 2 in [16]). Suppose that all transitions of process M in [16] are labelled with a and we want to check, at initial state s, the property that a is enabled infinitely often along all infinite paths. This is expressed with alternating fixed point formula:

$$\phi \equiv \nu X.\mu Y.([-].(\langle a\rangle true \wedge X) \vee Y) \tag{12}$$

which is true at initial state s of the process M. The problem can be directly encoded as the problem of solving the corresponding alternating equation system in Fig. 2. The results are given in Fig. 2. The columns are:

– Problem: Process $M = (S, A, \longrightarrow)$ from [16].

$$\left.\begin{aligned}
\nu\, x_s &= y_s \\
\mu\, y_s &= \bigwedge_{s' \in \nabla(t,s)} z_{s'} \vee y_s \\
\mu\, z_s &= \bigvee_{s' \in \nabla(a,s)} true \wedge x_s
\end{aligned}\right\} \text{for all } s \in S.$$

where $\nabla(t, s) := \{s' | s \xrightarrow{i} s' \wedge i \in A\}$
and $\nabla(a, s) := \{s' | s \xrightarrow{a} s'\}$.

Problem	$\lvert s \rvert$	$\lvert \longrightarrow \rvert$	n	Time (sec)
M_{500}	503	505	1006	4.0
M_{1000}	1003	1005	2006	16.4
M_{1500}	1503	1505	3006	39.0

Fig. 2. The Boolean equation system and experimental results.

- $|S|$: Number of states in M.
- $| \longrightarrow |$: Number of transitions in M.
- n: Number of equations in the corresponding Boolean equation system.
- Time: The time in seconds to solve variable x_s.

The benchmarks in [16] have a quite simple structure and no general results can be drawn from them. In fact, the equation system in Fig. 2 reduces to a sequence of purely conjunctive equations, whose encoding involves no *choose-1-of-2 rules*, i.e. is a Horn program. A more involved practical evaluation of our approach is highly desirable, and benchmarking on real world systems is left for future work.

7 Conclusion

We present an answer set programming based method for computing the solutions of alternating Boolean equation systems. We developed a novel characterization of solutions for variables in Boolean equation systems and based on this devised a mapping from systems with alternating fixed points to normal logic programs. Our translation is such that the solution of a given variable of an equation system can be determined by the existence of a stable model of the corresponding logic program. This result provides the basis for verifying μ-calculus formulas with alternating fixpoints using answer set programming techniques.

The experimental results indicate that stable model computation is quite a competitive approach to solve Boolean equations systems in which the number of alternation is relatively large. The alternation of fixpoint operators gives more expressive power in μ-calculus, but all known model checking algorithms are exponential in the alternation depth. Consequently, our approach is expected to be quite effective in the verification tasks where there is a need of formulas with great expressive power.

Acknowledgements. We would like to thank Keijo Heljanko for valuable discussions. The financial supports of Academy of Finland (project 53695), Emil Aaltonen foundation and Helsinki Graduate School in Computer Science and Engineering are gratefully acknowledged.

References

1. H.R. Andersen. Model checking and Boolean graphs. *Theoretical Computer Science*, 126:3–30, 1994.
2. A. Arnold and P. Crubille. A linear algorithm to solve fixed-point equations on transition systems *Information Processing Letters*, 29: 57–66, 1988.
3. A. Arnold and D. Niwinski. Rudiments of μ-calculus. *Studies in Logic and the foundations of mathematics*, Volume 146, Elsevier, 2001.
4. G. Bhat and R. Cleaveland. Efficient local model-checking for fragments of the modal μ-calculus. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1055, pages 107–126, Springer Verlag, 1996.
5. 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, 1999.
6. E.A. Emerson, C. Jutla and A.P. Sistla. On model checking for fragments of the μ-calculus. In *Proceedings of the Fifth International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 697, pages 385–396, Springer Verlag, 1993.
7. E.A. Emerson, C. Jutla, and A.P. Sistla. On model checking for the μ-calculus and its fragments. *Theoretical Computer Science*, 258:491–522, 2001.
8. 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.
9. 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.
10. 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.
11. M. Jurdzinski. Deciding the winner in parity games is in $UP \cap co{-}UP$. *Information Processing Letters*, 68:119–124, 1998.
12. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
13. 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.
14. V. Lifschitz. Answer Set Planning. In *Proceedings of the 16th International Conference on Logic Programming*, pages 25–37, The MIT Press, 1999.
15. 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.
16. 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.

17. 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.
18. A. Mader. Verification of Modal Properties using Boolean Equation Systems. PhD thesis, Technical University of Munich, 1997.
19. 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.
20. 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.
21. C. Papadimitriou. Computational Complexity. Addison-Wesley, 1994.
22. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
23. 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.