Conjunctive and Disjunctive Boolean Equation Systems

Simon Janssen

December 2007

Abstract

This paper presents two techniques to solve conjunctive and disjunctive boolean equation systems. The first technique is transforming the well known Gauß elimination method to a simpler form, that is still able to solve conjunctive/disjunctive BES. This technique has the time complexity $O(n^2)$, where n is the number of equations. The second technique is using the algorithm discussed in [1], with time complexity $O(elogd)$. Here e are the number of edges from the dependency graph, and d is the alternation depth. Both algorithms will be compared by looking at there implementation and there running time.

1 Introduction

BESs provide a useful framework for verification. It is mostly used for verifying properties of a transition system, for example a μ calculus formula stating that a transition system is deadlock free. Such a μ calculus formula together with the transition system can be translated to a BES. Some BESs have a certain form, in this paper we discus BESs which are in conjunctive/disjunctive form. Before we start with algorithms for conjunctive and disjunctive boolean equation systems we start by explaining what a BES is. The syntax and semantics of a BES is given in Section 2.1 and three general algorithms for solving a BES are given in Sections 2.2 through 2.4. Section 3 will discuss two algorithms for solving conjunctive and disjunctive BES. A gauß based method is given in section 3.1 and a method using the dependency graph is given in Section 3.2. The algorithms are compared and the results are shown in Section 3.3. Finally we draw a conclusion and list topics for further research in Section 4 and Section 5 respectively.

2 Boolean Equation Systems

A boolean equation system is a system which contains a series of boolean equations with a minimal (μ) or maximal (ν) fixedpoint. One use of a BES is deriving it from a modal μ-calculus formula and a transition system. A modal μ-calculus formula states a property of a transition system. Whether this property holds or not can be checked directly, or a BES can be derived to check it. Deriving a BES can be done in a straightforward method. A BES is a simple representation and contains all information it needs to check the property. We will discuss three general methods to solve a BES, but first we describe the formal syntax of a BES and its semantics.

2.1 Syntax and semantics

The syntax of a boolean equation system is as follows:

$S ::= (E)S|\delta$
$E ::= \sigma X_i = Q$
$Q ::= X_i|Q \wedge Q|Q \vee Q|false|true$
$\sigma ::= \mu|\nu$

Here S is a BES containing a set of equations, ending with a δ. We do not end a BES with a δ in this paper. E is an equation which starts with a fixedpoint operator. The operator can be a minimum fixedpoint operator (μ) or a maximum fixedpoint operator (ν). After the operator comes the name of the equation, here X_i for natural i. The righthand side of an equation is a term Q, which can contain equation names, conjunctions, disjunctions and the constants true and false. An equation name, true and false are called *literals*. A simple example of a BES is $(\mu X_1 = X_2)(\nu X_2 = X_1)$

If we do not look at the fixedpoint operators, the outcome of this system is $X_1 = X_2 = false$ or $X_1 = X_2 = true$. An equation of a BES only has one solution, which one is determined by the fixedpoints. A μ-fixedpoint leads to false and a ν-fixedpoint leads to true. In this example both fixedpoints occur. In that case the first fixedpoint has priority. So here $X_1 = X_2 = false$, so the order of the equations are important! The formal semantics of a BES, taken from [1] is:

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

Here ϵ is the solution of a BES relative to a valuation v. MIN defines a minimum fixedpoint and MAX a maximum fixedpoint.

Open and closed systems
A BES is closed if all equation names on the righthand side of an equation are defined in the BES. A BES is open if there exists an equation, where the righthand side contains an undeclared equation name. For example: the BES $(\mu X_1 = X_2 \vee X_3)(\nu X_2 = X_1)$ is open because X_3 is not declared. Removing X_3 from equation X_1 results in an closed system

Alternation depth
The number of μ/ν and ν/μ switches encountered in a BES is called the alternation depth. The *dependent* alternation depth does not count switches between μ or ν blocks, if the lower block does not contain equation names of the higher block.

2.2 Approximation

In [2] and [4] algorithms are described for using a approximation method on a μ calculus formula. This method is translated to work with BESs. The idea of approximation is to set every μ equation to false and every ν equation to true. This usually is not a valid solution, but a first approximation. Now we update the approximation by setting the last equation to the right value according to the rest of the system. After that we set the one before that to the right value. This is repeated until all equations have the right value. After updating equation i, all equations higher than i have to be recalculated again, because there value may change due to this update. Because this algorithm calculates the value for all equations it is called a *global* algorithm. Later we will see the tableau method, which is a *local* algorithm. To illustrate how the approximation method works we give an example:

$\nu X_1 = X_2$
$\nu X_2 = false$
$\mu X_3 = X_1 \vee X_3$
$\mu X_4 = X_2$

The table below shows the approximation steps. The first approximation is setting the values according to the fixedpoint operators. In the second step X_4 is updated. According to the BES X_4 should equal to X_2, which is the value true. In the fourth step X_2 is updated to false, here we see that X_4 also must be updated. So an update can trigger a recalculation of other equations, but if an other equation is changed it triggers an update again. Because of this behavior the algorithm has as worst exponential time complexity.

νX_1	T	T	T	T	F
νX_2	T	T	T	F	F
μX_3	F	F	T	T	F
μX_4	F	T	T	F	F

2.3 Tableau

Solving a BES with the tableau method can be seen as making a proof tree that proofs that an equation is true, see [2]. If such a proof tree does not exist the equation is false. Since we only look at one equation at a time the algorithm is called *local*. Creating the *proof tree*, or *tableau* only has four rules, see 2.3.1. The construction rules consider three cases: An equation is a conjunction, in that case you have to proof the items of the conjunction. The case where an equation is equal to an other equation, in which case only the other equation has to be proved. And a last case where an equation is a disjunction. Here a choice has to be made which on to proof.
Constructing the tableau continues until an equation name (X_i) occurs twice within the path from the root to the last equation. The last equation is called a *leaf*. Now let X_j be the equation, with the lowest value of j, located between (and including) the two occurrences of X_i. If X_j has a minimum fixedpoint the leaf is called false, if X_j has a maximum fixedpoint, the leaf is true.
If all leafs of a tableau are true, the starting equation is true. If a leaf is false, it can be so because of a bad choice made with a disjunction. So the algorithm backtracks to tries out all combinations with disjunctive equations. If all combinations still result to a false leaf, the starting equation is false.

Figure 2.3.1: Four rules for constructing a tree

$$[\wedge_1]\ \frac{X_i}{X_j \quad X_k} \quad \text{if } X_i = X_j \wedge X_k$$

$$[\wedge_2]\ \frac{X_i}{X_j} \quad \text{if } X_i = X_j$$

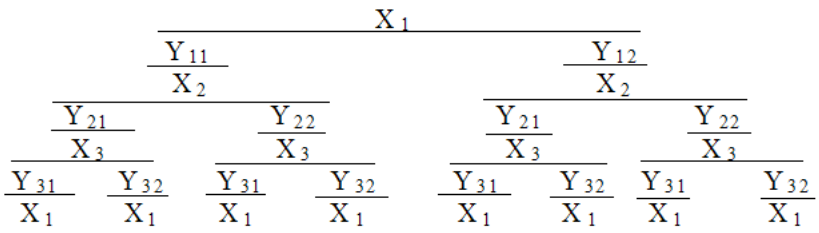
$$[\vee_1]\ \frac{X_i}{X_j} \quad \text{if } X_i = X_j \vee X_k$$

$$[\vee_2]\ \frac{X_i}{X_k} \quad \text{if } X_i = X_j \vee X_k$$

Figure 2.3.2 shows the tableau for proving the first equation of the BES: $(\nu X_1 = Y_{11} \wedge Y_{12})$ $(\mu X_2 = Y_{21} \wedge Y_{22})$ $(\mu X_3 = Y_{31} \wedge Y_{32})$ $(\mu Y_{11} = X_2)$ $(\mu Y_{12} = X_2)$ $(\mu Y_{21} = X_3)$ $(\mu Y_{22} = X_3)$ $(\mu Y_{31} = X_1)$ $(\mu Y_{32} = X_1)$. The names are chosen different here to better show the structure of the BES. All leafs have as smallest equation X_1. Because the fixedpoint of X_1 is ν, all leafs are true. This example also shows what can go wrong, a lot of expressions have to be calculated multiple times. This can be avoided by sharing subtrees. Subtree sharing for a μ calculus formula is explained in [3]. Subtree sharing cannot always be done because a branch stops at a leaf. Whether something is a leaf depends on the whole path from the root to the leaf. Thus sharing can only be done when two

branches also have the same leaf. This cannot be seen beforehand, so everything has to be calculated. This causes a exponential time complexity.

Figure 2.3.2: A tableau for equation X1



2.4 Gauß

In a Gauß method step, the BES is transformed to a shorter equalivant BES. This is done by taking the last equation ($\sigma X_n = Q$), where Q is a term, and substituting all occurrences of X_n on the right hand side of the other equations by Q. So if ($\sigma X_{n-1} = P \wedge X_n$) it is transformed to ($\sigma X_{n-1} = P \wedge Q$). After a substitution step we forget the last equation (not delete it, later we have to use it again). After every substitution step we simplify the BES by using the rules below:

Table 2.4

$X \wedge true$	=	X
$X \vee true$	=	true
$X \wedge false$	=	false
$X \vee false$	=	X
$X \vee (X \wedge Y)$	=	X
$X \wedge (X \vee Y$	=	X
$(X \wedge Y) \vee (X \wedge Z)$	=	$X \wedge (Y \vee Z)$
$(X \vee Y) \wedge (X \vee Z)$	=	$X \vee (Y \wedge Z)$

We also use an other property of BES, namely that if literal X_i occurs on the right hand side of equation X_i, the literal may be replaced by:
true if the fixedpoint of X_i is ν
false if the fixedpoint of X_i is μ

For a BES with n equations we do all these steps n-1 times. Then we remember all forgotten equations. For all X_i where the right hand side of X_i is either true or false, we substitute that value in all occurrences of X_i in the right hand side of all other equations. The order in which this happens does not matter. This is done until the only literals are true and false.

Because all equations have a value true or false, this is a global algorithm. It takes n-1 steps but a step can be exponential in time. This is because the right hand side can grow exponential. Therefor the algorithm takes exponential time.

3 Conjunctive and Disjunctive BES

A conjunctive BES is an equation system that does not contains disjunctions. Therefor every right hand side of an equation is a literal or a conjunction of literals. A Disjunctive BES is a BES with no conjunctions. This extra restriction leads to algorithms with polynomial time complexity. Later we will discuss a algorithm with time complexity $O(n^2)$ and an algorithm with time complexity $O(elogd)$. Here n are the number of equations in the BES, e the number of edges in the dependency graph and d the alternation depth. Conjunctive and disjunctive BES are limited, they cannot express all things a normal BES can. There are still some useful cases though. This is because a BES can sometimes be split into conjunctive/disjunctive parts. This can be done by finding the strongly connected components of a BES. If a BES can be split in a set of conjunctive components and a set of disjunctive components, the algorithms presented here can solve each component. If component A is dependent on an equation of component B, then component B has to be calculated first and the results must be inserted in component A. Note that if component A depends on B, component B does *not* depend on component A. This is true because else the two components would be one strongly connected component. One set of formulas satisfying this property is CTL. A CTL formula on a transition system can be written as strongly connected components, where each component is either conjunctive or disjunctive.
We will now present two algorithms for solving Conjunctive BES. Algorithms for disjunctive BES are similar.

3.1 Gauß method

With the extra restriction that we only work with a conjunctive BES we can derive a faster algorithm for solving this BES. We start by using the gauß elimination method. First we note that the righthand side of an equation cannot have more literals then the number of equations, that is if we remove all duplicates and simplify the literals true and false. The original gauß algorithm had an exponential running time because the right hand side could get exponentially large. Now the righthand side is linear, a faster algorithm exists using the gauß method. Table 2.4 presented eight simplification rules, most of these rules contain disjunctions. Removing these rules will lead to Table 3.1:

Table 3.1

$X \wedge true$	=	X
$X \wedge false$	=	false

Since all right hand sides of an equation is a conjunction, the second rule is very strong; When the literal *false* is encountered the whole equation is false. As explained before this algorithm works on a strongly connected BES. This means that if one equation is false, it is eventually substituted in the right hand side of all other equations. This means all equations are false. Therefor the strategy of this algorithm is *finding a false*. If it is not initially present, the only way to obtain false is if there is a X_i, for which $\mu X_i = \ldots X_i \ldots$ holds. In this case the literal X_i is replaced by false, which results in the whole conjunctive BES to be false. This is the first step of the algorithm, where only the μ equations are looked at, because only they can "make" a false:

```
i := 0
do i < n
  if Sign(Xi) == mu
    checkFalse(i,i)
  fi
  i := i+1
od
```

A μ equation X_i is only false if the right hand side contains the literal X_i. The literal X_i can be there initially, but it can also be inserted by a substitution step of a higher equation. For example $\ldots (\mu X_5 = X_2 \wedge X_{12}) \ldots (\nu X_{12} = X_5) \ldots$. Here X_5 is a μ equation and contains literal X_{12}. When doing substitution steps, by the time X_5 is reached X_{12} is already substituted by the literal X_5. Therefor the literal X_5 is part of the right hand side of the equation X_5 and the BES is false. The function *checkFalse* checks if it is possible to obtain literal X_i for equation X_i by looking at the literals that are substituted. This means the algorithm checks all literals higher than X_i. The function checkFalse has two arguments. The first argument of checkFalse is the literal to be found, the second argument is where we are looking to find it. Initially we try to find X_i in equation X_i before any substitution steps are done. Here is the pseudo code of the algorithm.

```
fun checkFalse(i, j)
  if "false is found"
    break recursion
  else
    S := "all literals of Xj not yet investigated and higher than i"
    if "i is in S"
      return "false found"
    else
      checkFalse(i,s) forall s in S
    fi
  fi
nuf
```

The algorithm looks at all literals that can be reached from X_i. If X_i itself is found, the whole BES is false. If none of the μ equations is false the BES is true. The first part of the algorithm checks all μ equations and calls checkFalse, so it has time complexity O(n*$checkFalse$). CheckFalse looks at all unique literals, which is at most the number of equations, which is O(n). The time complexity of the whole algorithm is $O(n^2)$.

3.2 Dependency graph method

This method only works on a conjunctive BES. The disjunctive algorithm is similar and is described in detail in [1]. Here we give a short description how the algorithm works. Furthermore we look more to the implementation side. This method works with the *dependency graph* of a BES. A dependency graph is obtained by making a node for every equation, and creating a directed edge from A to B if B is on the righthand side of A. The fixedpoint of an equation is saved in the node name, also called the label name. More general, for a BES with equations $\sigma X_1 \dots \sigma X_n$ we make n nodes named $\sigma X_1 \dots \sigma X_n$. Afterwards for every equation $\sigma X_i = X_{j1} \dots X_{jm}$ we add the edges $(X_i, X_{j1}) \dots (X_i, X_{jm})$. The result is a dependency graph G = <V, E, l> where V are the nodes, E are the edges and l are the labels. The rest of the algorithm does not need the original BES, it only works with the graph.
Just like the last algorithm we try to find a loop starting at a μ equation X_i and only following literals X_j where $j > i$. Here we do not do this for one equation at a time but for a whole μ block. Here we present pseudo-code to illustrate how to check if a μ block contains a self loop:

```
G = <V,E,l>

fun muBlock(Xi)
  E' = {(Xk, Xl)| k>=i && l>=i && (Xk,Xl) in E}
  G' = <V,E',l>
  SCC = "Non trivial strongly connected components of G' "

  forall Xj in SCC do
    inSCC[j] := true
  od
  forall Xj in mu-block do
    if inSCC[j]
      return FALSE FOUND
    fi
  od
nuf
```

The first step of the function muBlock is removing all edges from and to equations X_j where $j <$ i. After removing these edges the algorithm cannot

make a loop using equations which are lower then X_i. In line three we find all strongly connected components (SCC) of the graph with the removed edges. There exist several algorithms to do this in $O(E)$ time. One of such algorithms is Tarjans SCC algorithm. This is a depth first algorithm that makes a list of all SCCs. Making G' with the removed edges can be ommited if we use a slightly altered version of Tarjans SCC algorithm, where we skip nodes with a lower index then i. The next lines of code check if there is a node of the μ block in a SCC. If this is the case it can make a loop using only higher equations and thus makes the equation false (and the whole conjunctive BES). This is true because all equations with a index lower then i were removed.

The function muBlock is used in the main algorithm. The main algorithm checks if there is a false equation between equation k1 and k2, where initially k1 is the first μ equation and k2 is the last equation. The middle mu block, starting with equation k3, is checked with muBlock(k3), if false is not found the graph is split into two smaller graphs. Note that $k1 \leq k3 \leq k2$. Finally the main function is called twice, once for the lower half and once for the higher half, both with adjusted ranges k1..k2:

```
fun graphMethod(G, k1, k2)
  k3 := middle mu block starting point
  if muBlock(k3)
    return FALSE FOUND
  else
    G'  := lowerG
    G'' := higherG
    graphMethod(G', k1, k3-2)
    k4 := first mu equation after middle mu block
    graphMethod(G', k4, k2)
  fi
nuf
```

The ranges for G' and G" are simple, since k3 is a *first* μ equation k3-1 is a ν equation. for G' k2 can therefor safely be set to k3-2 because it does not skip a μ equation. For G" we set k1 to the next unchecked μ equation. G' and G" hold only the information needed to check the lower and higher half respectively. G' and G" are defined by:
$G' = (V', E', l')andG'' = (V'', E'', l)$

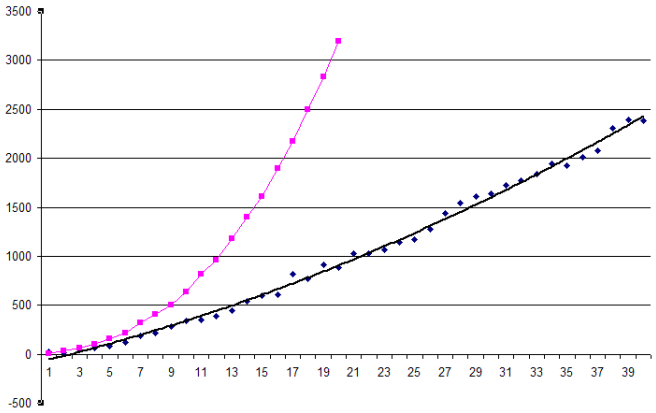
$$V' = \{C(i) \mid i \in V \text{ and } \exists j.\langle i,j\rangle \in E \text{ and } C(i) \neq C(j)\},$$
$$E' = \{\langle C(i), C(j)\rangle \in V' \times V' \mid \langle i,j\rangle \in E,\ C(i) \neq C(j)\} \text{ and}$$
$$\ell'(i) = \begin{cases} \ell(i) & \text{if } C(i) \text{ trivial,} \\ \nu &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

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

Here $E' \cap E'' = \emptyset$, and the range is split in two halves. One iteration of graphMethod has time complexity $O(E)$. Every iteration one mu block is checked, and the recursion halves the number of μ blocks. Because the number of μ blocks are $O(alternationdepth)$ the total time complexity for this algorithm is $O(elogd)$, where e is then number of edges and d the alternation depth.

3.3 Results

Looking purely at the worst case running time we suspect that the second algorithm is a lot faster. This is tested by implementing both algorithms in C#. There is still room for small optimizations, but both implementations have the right time complexities. One thing that becomes clear immediately is that the gauß based algorithm is much simpler then the dependency graph algorithm. This is clear when we look at the lines of code (LOC). The implementation of the gauß based algorithm only takes 25 LOC, while the implementation of the dependency graph takes over 400 LOC.
To test if this extra work is worth it we measure the running time of both algorithms on a lot of generated conjunctive BES. The tests vary the number of literals on the right hand side of the equations, but all tests show the same results. The graph below shows the running time in ms for equations with an average right hand side of ten literals. The x-axis shows the number of thousands of equations. The dots indicate the value and the line is a *trend line*. The pink line illustrates the running time for the gauß based algorithm and the blue line the dependency graph algorithm. The gauß based method is faster for 1000 equations, but both algorithms finish within a few milliseconds. For big systems we can clearly see that the dependency graph algorithm is a lot faster.



A test on a real world application can be done for example by making models with the mcrl2 toolset. These models can be translated to a linear process, which can be translated, together with a $/mu$ calculus formula, to a parameterized BES, which in turn can be translated to a normal BES. Unfortunately it can take hours to make a BES if the statespace is large. Four tests have been done on models that come with the toolset. They are checked to see if they are deadlock free. The results are shown below. Here creation time is the time spend on transforming a mcrl2 specification to a BES.

Model	Creation Time	Time Gauß	Time Graph
Alternating bit protocol	50 ms	0 ms	0 ms
Trains (mcrl example)	20 ms	0 ms	0 ms
Solitaire	>8 hours	n/a	n/a
Hex (game)	>8 hours	n/a	n/a

Here we immediately see a problem. The time to transform a PBES to a BES takes a lot of time so two models have been aborted. This makes using plain BESs with the mcrl2 toolset very cumbersome for models with a big state space. For now we cannot give a decent comparison of the two algorithms in real world applications. This is now open for later investigation.

4 Conclusion

For the general case there does not yet exists an algorithm with polynomial time complexity. Putting extra constraints on a BES can lead to polynomial time algorithms. For conjunctive and disjunctive BES, we loose a lot of power, but we still can express a lot of things. This is because we can split some systems into parts which are in conjunctive or disjunctive form. This way we can calculate for example CTL formulas.
Comparing the gauß based algorithm and the dependency graph algorithm we

see a great difference in running time. The running time of the gauß based algorithm increases very fast for large inputs, while the dependency graph algorithm is sub-quadratic. This makes the extra work to implement this method worthwhile.

5 Further research

There are linear time algorithms for solving alternation free BESs. Alternation free means that if equation X depends on Y and Y depends on X, both X and Y should have the same fixedpoint operator. According to [1] many equivalence/preorder checking problems result in an alternation free BES in conjunctive/disjunctive form. Both the alternation free and dependency graph algorithms can be used here. It would be interesting to test these two algorithms for running time and memory usage.

The two algorithms for solving conjunctive/disjunctive BESs that are presented in this paper are not yet tested against large real world applications. This is still open, but looking at the results, the dependency graph method is likely to have a much better running time.

6 bibliography

[1] Jan Friso Groote and Misa Keinanen. A sub-quadratic algorithm for conjunctive and disjunctive BESs
[2] Angelika Mader. Verification of modal properties using boolean equation systems
[3] Angelika Mader. Tableau recycling
[4] Rance Cleaveland, Marion Klein and Bernhard Steffen. Faster Model Checking for the Modal Mu-Calculus