# Inference in Biological Regulatory Networks
## Extended Abstract

Alexandre Lemos
{alexandre.lemos}@tecnico.ulisboa.pt

INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal

**Abstract.** Models of biological regulatory networks are increasingly used to formally describe and understand complex biological processes. Such models often need repairing whenever new observations become available, because the model cannot generate behaviours consistent with the new observations, or because the behaviours are contradictory. This process of model repair is often manual and therefore prone to errors.

This work, describes biological regulatory networks using the Boolean formalism. We propose to repair the model by changing inconsistent functions, with four types of atomic repairs which can be further combined. The goal is to find the cardinality minimal set of repairs allowing the model to satisfy all available observations. The proposed method is implemented using Answer Set Programming (ASP) and Maximum Satisfiability (MaxSAT). The systems were tested using real life organisms and find adequate solutions to ensure consistency for all observations. The solution implemented using Maximum Satisfiability was the most efficient.

**Keywords:** Boolean regulatory networks, Model repair, Boolean functions, Answer set programming, Maximum satisfiability

## 1 Introduction

Nowadays, most biological models are still handmade and require a great amount of effort by the modeller. Different models can be derived from the same set of data and different modellers will therefore most likely build different models. Every time new data is obtained, it is necessary to reassess the model consistency. If the model is not consistent with the new data, then it needs to be corrected. So, it is important to reduce the difficulty of this task by creating computational tools that allow the representation of models and to reason over them.

Biological regulatory and signalling networks are composed by regulatory components, representing the expression level of genes or the activity of their corresponding proteins. However, often the amount of available data detailing many biological processes is scarce and a qualitative (less detailed) model is more suited to describe them. Many qualitative mathematical formalisms exist, which have been applied for the modelling of biological regulatory and signalling networks, such as Petri nets [1], Sign Consistency Model (SCM) [2], piecewise-linear differential equations [3] or the logical formalism [4]. Even though these formalisms generate complex dynamics, in this work we focus only on the long term system behaviours, in particular the stable states of the system, which denote biologically relevant behaviours (e.g. cell fates in a differentiation process).

In the Boolean formalism, nodes are represented by Boolean variables denoting biological components and edges denote regulatory interactions between components. Additionally, the evolution of the level of activity of a given component is described by a logical function, combining the values of the regulators of the component. Here, we propose the use of the Boolean formalism to describe biological models at steady state and the use of two different approaches too check its consistency, thus allowing the model to be revised. First, an Answer Set Programming (ASP)-based approach, that it is easier to encode and modify, and then a MaxSAT-based approach that is harder to encode but faster to solve. The corrections considered here are focused on repairing the Boolean functions that generate a conflict.

This paper is organized as follows. The next section introduces the preliminaries, namely Boolean regulatory graphs, ASP and MaxSAT. In Section 3 our approach will be described and explained. Section 4 describes the encodings for both ASP and MaxSAT.[1]. In Section 5 we test the proposed approach with three real data sets and discuss the results.

[1] The complete ASP and MaxSAT encodings are available at http://web.ist.utl.pt/~alexandre.lemos/rbn/

## 2 Preliminaries

### 2.1 Boolean regulatory graphs

A Boolean regulatory graph is defined by:

- a set of $n$ regulatory components $G = \{g_0, \cdots, g_n\}$ , where each component is associated with a Boolean variable representing the level of expression or activity of the component;
- a set of edges $E$, where $(g_i, g_j) \in E$, with $i, j \in [1, \cdots, n]$ denotes a regulatory interaction between components $g_i$ and $g_j$, *i.e.*, $g_i$ is a regulator of (influences) $g_j$;
- to each component $g_i$ there is an associated regulatory logical function, $K_i$: $B^k \rightarrow B$ where $B = \{0, 1\}$ and $k >= 0$, which defines its value based on the value of its regulators. Components without regulators are denoted as inputs and have constant values $\in B$.

Since a multivalued network can be represented by an equivalent Boolean network [5], this work will focus only on the Boolean case. In this case, a regulatory component is considered to be active/inactive if the value of the variable is `true`/`false`. A regulatory logical function is defined by the combinations of three basic Boolean functions (And, Or, Not).

### 2.2 Answer Set Programming

Tools for reasoning over biological regulatory and signalling networks have been implemented in the past using ASP [6]. An ASP program is defined by a set of rules, where each rule has a head and a body, and is written in the following form:

$$l_0 \leftarrow l_1, \ldots lm, \sim l_{m+1}, \cdots \sim l_n$$

where $l_i$ is a literal (i.e. a predicate in first-order logic) and $\sim l_i$ is its (default) negation. The left side of $\leftarrow$ is denoted as the head of the rule and the right side as the body. The head is `true` if the body holds, i.e. if all the positive literals, $l_1$ to $l_m$, are `true` and the negative literals, $\sim l_{m+1}$ to $\sim l_n$, *can* be `false` [7]. $\leftarrow l$ is a rule without a head and thus represents a constraint meaning that $l$ must not be satisfied. A rule that only has a head $l$, means that $l$ must be satisfied and it is called a *fact*. In ASP, as in Prolog and in first-order logic, it is possible to express predicates with n arguments, which can be represented by $p(l_0, ..., l_n)$. This predicate can be used as a literal on the body or as a head of a rule. The ground instantiation of an ASP program is obtained by replacing all the first-order logic variables by elements of the Herbrand universe [8] of the program (a universe that contains all the constants from the program and every function whose arguments belong to this universe). A set of literals is a model of the program $P$ if the set satisfies all the rules of $P$. The idea behind ASP is to encode the problem into a program such that the answer is the solution to the problem.

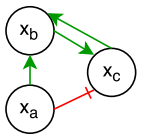
### 2.3 Maximum Satisfiability

Maximum Satisfiability (MaxSAT) solvers are already used to reason over biological networks [9]. Many algorithms that solve MaxSAT apply a Boolean Satisfiability Problem (SAT) solver iteratively. The SAT solvers are used as black-boxes, making it easy to change the SAT solver used (it can always be updated with new developments of SAT solvers).

In order to present the MaxSAT problem it is important to define first the concepts of SAT. A literal, l, is either a Boolean variable $x$ (positive literal) or its negation $\neg x$ (negative literal). A clause `c` is a disjunction of $n$ literals $c = l_1 \vee ... \vee l_n$. A formula is written in the Conjunctive Normal Form (CNF)[2], *i.e.* a conjunction of $n$ clauses $f = c_1 \wedge ... \wedge c_n$. A partial (complete) assignment is a mapping between some (all) the variables of a formula and a Boolean value. A positive (negative) literal is satisfied iff is assigned the value true (false). A clause is satisfied iff there is at least one literal satisfied. Otherwise the clause is unsatisfied. A formula is satisfiable iff there is at least one assignment where all the clauses are satisfied. A formula is unsatisfiable iff there is no assignment that makes the formula satisfiable. SAT is the problem of checking the satisfiability of a Boolean formula, i.e., determining if a given formula has an assignment that satisfies all the clauses in the formula.

Finally, we can now introduce the MaxSAT problem. The MaxSAT problem is a generalization of SAT, where the objective is to find an assignment that maximizes the number of satisfied clauses. It is also

[2] Most SAT algorithms require the formulas to be written in CNF. The formulas that are not represented in CNF can be transformed into CNF [10].




$K_a(x) = x_a$ $\quad$ $K_a(x) = x_a$

$K_b(x) = x_a \vee x_c$ $\quad$ $K_b(x) = \neg(x_a \vee x_c)$

$K_c(x) = x_b \wedge \neg x_a$ $\quad$ $K_c(x) = \neg(x_b \wedge \neg x_a)$

Fig. 1: Representation of a small network (left), the respective logical functions (center) and the repaired logical functions for an experimental profile $x_a$=`true`, $x_b$=`false` and $x_c$=`true` (right).

| | $A \wedge B$ | $A \wedge \neg B$ | $\neg A \wedge \neg B$ | $\neg A \vee B$ | $A \vee B$ | $A \vee \neg B$ | $\neg A \vee \neg B$ | $B$ | $A$ | $\neg B$ | $\neg A$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| repair | g | g,i | i | g | g | g, i | g, i | e | e,g | e,i | e |

Table 1: Possible replacements for function ¬ A ∧ B and which repairs are used to achieve them.

possible to see the MaxSAT problem with a different perspective: finding an assignment that minimizes the number of unsatisfied clauses. There are other variations of the MaxSAT problem and the one in focus here is partial MaxSAT. In the partial MaxSAT problem the clauses have two different categories. Some clauses can be categorized as hard and the others as soft. The objective of partial MaxSAT is to find an assignment that satisfies all the hard clauses and maximizes the number of soft clauses that are satisfied.

## 3 Repairing Boolean regulatory networks

After creating a model, such as the one described in Figure 1, new observations of the process under study may arise that should also be explained by the model. Combining both may generate inconsistencies and so the model will need to be revised in order to find a new model that also satisfies the new data. For example, considering the experimental profile: $x_a$=`true`, $x_b$=`false` and $x_c$=`true`, one can see that the model in Figure 1 is inconsistent, since the functions that explain the value of the node $b$ and $c$ are incoherent with the experimental profile. The potential model revisions should try to find repairs that make the model coherent to all the available data.

Here, we propose four basic types of repairs to the logical functions, which can be further combined:

- $e$ - removes a regulator (ensuring that a component has at least one regulator, *i.e.*, never becomes an input);
- $i$ - negates any number of regulators of a function;
- $n$ - changes an AND/OR function into NAND/NOR function, respectively;
- $g$ - changes an AND into an OR and a NOT into the `identity` function.

Considering the model described in Figure 1 and the given experimental profile, the Boolean functions (center) can be repaired by applying two repairs of type $n$, *i.e.* by negating the functions $K_b$ and $K_c$ (Figure 1 right). These repairs are cardinality minimal, corresponding to the minimal number of repairs required to correct the model. However, it is possible to perform other types of repairs to correct this model, such as removing the NOT and negating the remaining regulators of the inconsistent functions. The logical functions will be: $K_b(x) = \neg x_a \vee \neg x_c$ and $K_c(x) = \neg x_b \wedge x_a$.

Generically, the number of possible Boolean functions that can be used to repair a function will increase with the number of regulators of a component. For example, the number of possible functions with two arguments is sixteen and this number will increase exponentially with the number of arguments (number of regulatory components that influence one specific component). The growth will follow the expression $2^{2^n}$ where $n$ is the number of arguments of the function [11].

Considering the case of a function with two arguments, by combining repairs $e$, $i$ and $g$, one can achieve the total of twelve functions (all basic Boolean functions, plus one of the derived Boolean functions, the implication). The functions XOR, XNOR (equivalence operation), `true` and `false` are not achievable by these repairs. Table 1 shows the different combinations of repairs needed to convert the binary function $\neg A \wedge B$ into a different one (whenever possible).

```
funcOr(1,b).      funcAnd(2,c).          funcNot(3,temp(a)).
regulator(1,c).   regulator(2,b).        regulator(3,a).
regulator(1,a).   regulator(2,temp(a)).  node(temp(a)).
node(a).          node(b).               node(c).
edge(temp(a),c).  edge(a,b).             edge(a,temp(a)).
edge(c,b).        edge(b,c).
```

Fig. 2: Partial encoding for the network shown in Figure 1 is shown.

# 4 Implementation

## 4.1 Encoding in Answer Set Programming

To encode the network shown in Figure 1, it is necessary to write the predicate `node(V)` for each node `v` and the predicate `obs_vlabel(P,V,S)` to give to each node an observed value `S` in an experimental profile `P`. The predicate `edge(`$v_1$`,`$v_2$`)` represents an unidirectional relation between two nodes $v_1$ and $v_2$.

Four types of basic Boolean functions can be encoded - AND, OR, `identity` and NOT - through the predicate `func<F>(N,O)`, where `<F>` is the name of the function, `N` a unique identifier and `O` the output node of the function `<F>`. The unique identifier `N` is used to set the arbitrary number of arguments of the function. The predicate `regulator(N,V)` associates the function with the name `N` with one of its argument `V`. To define functions with more than one argument, it is necessary to define one predicate for each argument. It is possible to construct more complex functions, using temporary nodes to combine functions. The partial encoding for the network shown in Figure 1 is shown in Figure 2. The predicate `temp` is used to define temporary nodes used to create more complex functions.

A predicate `consistentFunc(P,V)`, where `P` is a profile and `V` a node, is generated if and only if the value of the node `V` is coherent with the result of the function that explains the presence of this node. The rules (1) and (2) are used to verify the consistency of an AND function if the function AND is not repaired to an OR. The predicates `vlabel(P,O,S)` and `obs_vlabel(P,O,S)` are similar, since both represent the value of the node `V` in the experimental profile `P`. `vlabel(P,O,S)` is inferred when the observed value (`obs_vlabel(P,O,S)`) is present or, if no observed value was specified, with a previously computed value. The predicate `noneNegative` is inferred (3) when all the regulators of the node have the value `true`.

The consistency check of the network, with or without repairs, is made with two auxiliary predicates (`onePositive` and `oneNegative`). The presence of these predicates indicates that a node `V` influenced by the function `N` and based on the profile `P` has at least one regulator with the value `true`/`false`.

The non-existence of an instance of the predicate `consistentFunc(P,V)` for a non-input node in a given profile means that the network is inconsistent (8). Note that all nodes without incoming edges are considered input nodes.

Rule (4) shows the possibility of creating a NAND function; to use this repair it is necessary to have the repair active ($repair_n$) and the existence of an AND function (not previously repaired to an OR/NAND/NOR).

Removing an edge is a possible repair when the flag $repair_e$ is active and there are at least two edges that were not removed for the given node (rule 5).

Rules (6) and (7) are used to infer the possibility of a repair $i$ or $g$. Rule (6) ensures that a previously negated regulator is not negated more than once.

```
consistentFunc(P,O) ← funcAnd(N,O), noneNegative(O,N,P),
          vlabel(P,O,1), ∼ repair(funcOr(N,O)).                    (1)
consistentFunc(P,O) ← funcAnd(N,O),∼ noneNegative(O,N,P),
           vlabel(P,O,0), ∼ repair(funcOr(N,O)).                   (2)
noneNegative(V,N,P) ← ∼ oneNegative(V,N,P), onePositive(V,N,P).    (3)
pos(funcNand(N,O)) ← repair_n, funcAnd(N,O), ∼ isNandNor(O),
                  ∼ repair(funcOr(N,O)).                           (4)
pos(rEdge(U,V)) ← repair_e, edge(U,V), edge(W,V), W!=V,
        W!=U,U!=V, ∼ rEdge(U,V), ∼ rEdge(W,V).                     (5)
pos(regulator(N,V)) ← repair_i,∼ isRegulatorNot(N,V), regulator(N,V). (6)
pos(funcAnd(N,O)) ← repair_g, funcOr(N,O).                         (7)
← vlabel(P,V,S), ∼ input(P,V), ∼ consistentFunc(P,V).              (8)
```

Fig. 3: Part of the ASP encoding to check the consistency and to repair the network.

### 4.2 Encoding into Maximum Satisfiability

In order to use a MaxSAT solver, the biological network has to be encoded into CNF and for that it is necessary to define a few Boolean variables.

For each node `v`, there is a Boolean variable $label_v$ such that $label_v$ is assigned the value `true/false` if node `v` is active/inactive. Every edge between two nodes can have a function NOT associated. In order to describe it two auxiliary variables are used:

- $f\text{NOT}_{vu}$ is a Boolean variable that when assigned `true` represents the existence of the function NOT on the edge between the nodes $v$ and $u$;
- $rf\text{NOT}_{vu}$ is a Boolean variable that represents the result of the application of the NOT function on the edge between the nodes $v$ and $u$.

With these variables it is possible to write constraints to allow a MaxSAT solver to check the network's consistency. First, for each node a unit clause [3] corresponding to the value of the node in the experimental profile is created. There will also be created unit clauses for the variables $f\text{NOT}_{vu}$. These clauses will consist of a single positive literal. When one only wants to check the satisfiability of a network it is not necessary to have a list of unit clauses for every edge without a NOT function. To encode the function one can use the Tseitin transformation [10] that creates a CNF formula based on an arbitrary circuit. A function AND is encoded as follows:

$(\neg r_0 \vee .. \vee \neg r_n \vee v) \wedge (r_0 \vee \neg v) \wedge .. \wedge (r_n \vee \neg v)$,

where $r_0$ and $r_n$ are regulators and $v$ is the output of the function. Very similarly one can encode an OR function:

$(r_0 \vee .. \vee r_n \vee \neg v) \wedge (\neg r_0 \vee v) \wedge .. \wedge (\neg r_n \vee v)$.

The encoding for the NOT function is the following:

$(\neg r \vee \neg v) \wedge (r \vee v)$.

This encoding requires that the number of clauses for each function is the number of regulators plus one. Note that it is not necessary to have a variable to identify an input node since an input is not regulated and so the solver will not check its consistency.

With this type of encodings it is possible to verify the consistency of the network shown in Figure 1 with the following CNF formula:

$$(\neg x_a \vee \neg t_{ac}) \wedge (x_a \vee t_{ac}) \wedge (\neg x_b \vee \neg t_{ac} \vee x_c) \wedge (r_b \vee \neg x_c) \wedge (t_{ac} \vee \neg x_c) \wedge (X_a \vee x_c \vee \neg x_b) \wedge (\neg x_a \vee x_b) \wedge (\neg r_c \vee x_b).$$

Repairing a network is an optimization problem that can be encoded into Partial MaxSAT. The files containing the encoding are generated specifically to each repair. This allows to have shorter encodings when possible. To apply the repairs previously shown it is necessary to add more constraints depending on which repair is available.

**Repair *e*** In order to remove an edge, it is necessary to add a new variable, $N_{vu}$ for each edge $(v, u)$ that can be removed. This variable is used to deactivate clauses, *i.e.* satisfy clauses where an edge is removed. When $N_{vu}$ is assigned the value `true` it means that the edge from $u$ to $v$ has been removed. With this repair one could remove all edges going to a node. To avoid removing all the edges into a node, a clause $(\neg N_{vu} \vee .. \vee \neg N_{wu})$ is defined, ensuring that each node has at least one incoming edge. The variable $N_{vu}$ is added to all clauses containing the variables for the node $v$ and $u$ at the same time. This repair only influences the functions with more than one argument. As it is possible to see, the encoding of these functions has two types of clauses: one type where only one regulator is present, and another type where we have all the regulators together. When the variable $N_{vu}$ is assigned to `true`, the clauses that contain this variable are trivially solved. However, the second type of clauses also becomes `true` under this condition. To solve this problem it is necessary to replicate this clause with all combinations of regulators. These new clauses are trivially solved when the variable $N_{vu}$ is assigned to `false`. A unit clause for the variable $N_{vu}$ will be created. This clause will consist of a single negative literal and will be considered a soft clause.

For example, the AND function with two arguments is now encoded as follows:

$(N_{r0}v \vee \neg r_0 \vee N_{r1}v \vee \neg r_1 \vee v) \wedge (N_{r0}v \vee r_0 \vee \neg v) \wedge (N_{r1}v \vee r_1 \vee \neg v)$.

[3] A clause that has only one unassigned literal and so must be assigned `true` for the clause to be satisfied.

**Repair *g*** To allow a function to be changed, for example from AND to OR, it is necessary to use variables to describe which function is associated to each node. For each regulatory logical function $K_v$ there is a variable $f{<}function{>}_v$ that is assigned the value `true`/`false` if the function for the node $v$ is of the type $<function>$. In this case, $<function>$ can only be AND or OR. It is only specified the value for the active function (otherwise changing a function would cost twice). This value is represented as a soft unit clause. This way this constraint can be broken when needed. All nodes have encoded the OR an AND functions. The clauses required to encode the functions OR and AND are the same as shown before. However, these clauses need to have one more variable. Each clause of the encoding of the function needs to include the literal $\neg f{<}function{>}_v$ in order to specify which function is currently active. For example, the AND function with two arguments is now encoded as follows:

$(\neg f\text{AND}_v \vee \neg r_0 \vee \neg r_1 \vee v) \wedge (\neg f\text{AND}_v \vee r_0 \vee \neg v) \wedge (\neg f\text{AND}_v \vee r_1 \vee \neg v)$.

In order to have only one type of function for each node (*e.g.* having the AND and OR function for the node `v`) it is necessary to define the following constraint:

$$\bigwedge_{v \in V} \bigvee_{f, f_1 \in \{f\text{AND}, f\text{OR}\}} \neg f_v \neg f_{1v}. \tag{9}$$

This constraint guarantees that at most one function is active for a given node. However, this is not enough since we want to have exactly one function per node. To achieve this we need to add a new constraint:

$$\bigwedge_{v \in V} f\text{OR}_v \vee f\text{AND}_v. \tag{10}$$

This repair also allows changing a NOT function into an IDENTITY function. For this it is necessary to encode the IDENTITY function. The IDENTITY function is encoded like the AND or OR function but with only one argument, $(fnot_{vu} \vee v \vee \neg u) \wedge (fnot_{vu} \vee \neg v \vee u)$. The unit clause representing the assignment of $fnot_{vu}$ as `true`, *i.e.* $(fnot_{vu})$, is considered a soft clause.

**Repair *i*** In a similar way as the repair $g$ deals with the change between AND and OR, this repair deals with having a function's argument negated or not. In order to do that it is necessary to have a variable that is assigned to `true` when the argument is negated and `false` otherwise. As it was said before, this repair can only negate an argument that has not been previously negated. So all the arguments that are negated on the model are described like when considering repair $g$ with the exception that now the unit clause $(f\text{NOT}_v)$ is considered hard instead of soft. The constraints that are described for repair $g$ (about the function NOT and IDENTITY) are instantiated for all arguments. Now, for the arguments that are not already negated, the variable $f\text{NOT}_v$ is assigned as `false` in a soft clause making it possible to negate a function's argument.

**Repair *n*** For this repair, it is necessary to add two more variables to describe the functions NAND and NOR for each node. When generating the encoding for this repair alone, it is only necessary to have one of these variables for each node since it is impossible to change an AND for an OR. To verify the consistency of these new functions it is necessary to define some more constraints. These new constraints are trivially solved if the corresponding function is inactive, *i.e.* if the variable $f\langle\text{function}\rangle_v$ is `false`. To describe the behaviour of the NAND function it is required to:

$(\neg r_0 \vee .. \vee \neg r_n \vee \neg v \vee \neg f\text{NAND}_v) \wedge (r_0 \vee v \vee \neg f\text{NAND}_v) \wedge .. \wedge (r_n \vee v \vee \neg f\text{NAND}_v)$.

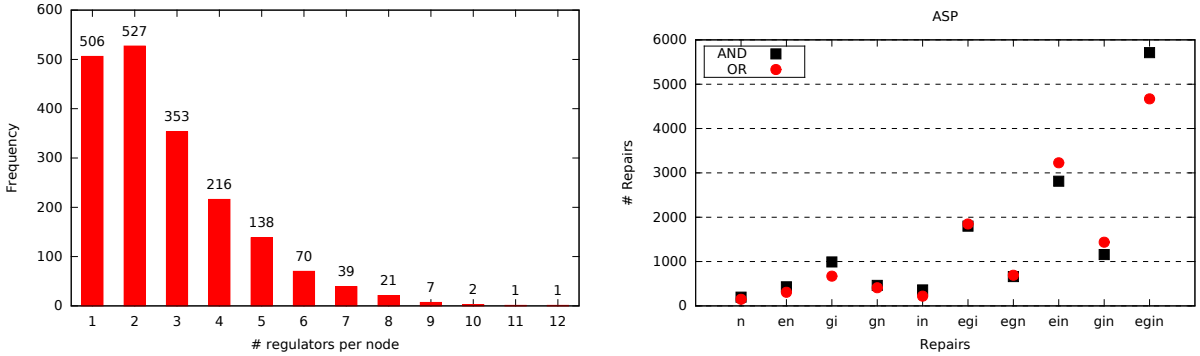
The encoding for the NOR is:

$(r_0 \vee .. \vee r_n \vee v \vee \neg f\text{NOR}_v) \wedge (\neg r_0 \vee \neg v \vee \neg f\text{NOR}_v) \wedge .. \wedge (\neg r_n \vee \neg v \vee \neg f\text{NOR}_v)$.

The encoding previously shown for the AND and OR functions requires the addition of the negative literal corresponding to the function in each clause. The constraints (9) and (10) are needed but considering only the reachable functions with the active repairs (*e.g.* when considering AND as the function for node $v$ and the repair $n$ as the only repair possible NAND and AND are the only reachable functions).

*Combining repairs* - When combining repairs the encoding is basically the instantiation of the combination of the encodings discussed above, with two more little details:

- First, when considering a combination of repairs including both repairs $g$ and $n$ it is required to change the domain of the constraints (9) and (10) given that all functions are reachable.
- Finally, it is harder to change a function from an AND function to a NOR function than an AND function to an OR function. These repairs should have a higher cost and so a new variable is created for the following constraints: $(\neg f\text{NOR}_v / f\text{NAND}_v \vee aux_v) \wedge (f\text{NOR}_v / f\text{NAND}_v \vee \neg aux_v) \wedge (\neg aux_v)$. The first two constraints are soft and so, like in the ASP version, the cost is two.




(a) (b)

Fig. 4: (a) Distribution of the number of regulators per node for *Escherichia coli* model. (b) The number of repairs necessary to correct the Stat vs Exp data set. The types of repairs that are not present did not find a feasible solution. Repair $n$ is the only one to find an optimal solution. The results were obtained using ASP.

## 5 Results

In order to verify if the proposed repairs are sufficient to repair biological regulatory and signalling networks, tests were executed using biological data of *Escherichia coli* and *Candida albicans* (no results shown). The tests were also used to access the performance to obtain the needed repairs.

### 5.1 Experimental Setup

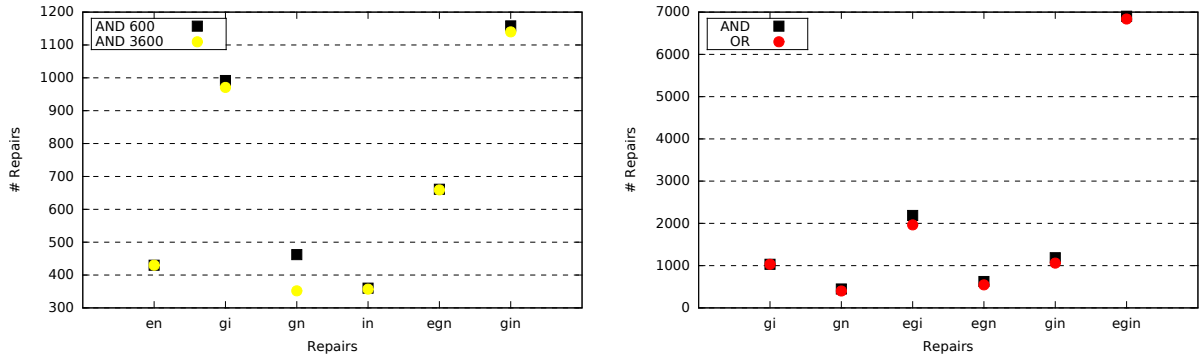
The data sets *HeatShock* [12] and *Stat vs Exp* [13] were used for evaluating the proposed approach. The network of *Escherichia coli* was obtained from `RegulonDB` [14]. The HeatShock data set describes the *Escherichia coli* response to an increase of temperature. The *Stat vs Exp* data set corresponds to the Exponential-Stationary growth shift study of *Escherichia coli*. These data sets were obtained from Gebser *et al* [15]. Since the models were imported from the SCM formalism, and this formalism does not have a Boolean function associated to each node, the models adapted considering that all nodes have the same (default) function. In order to better understand the behaviour of the regulatory components described in these data sets, they were tested considering two different default functions, And and Or, combining the set of regulators without changing the sign of the regulators. This means that for each data set, we consider two models: where all the nodes with more than one regulator have the And function, and another where all the nodes with more than one regulator have the Or function.

The tests were executed using the *runsolver* tool [16] with a time out of 600 seconds and a limit of 3 Gb of memory. The ASP program was executed using *Gringo*(version 4.5.4) [17] and *Clasp*(version 3.1.4) [18]. The MaxSAT encoding was executed using the MaxSAT solver *Open-WBO* (version 1.3.1) [19]. The two different implementations were run on a computer running *Ubuntu 14* equipped with 24 CPUs at 2.6 GHz and 64 Gb of RAM.

It is interesting to see that most nodes have a small number of regulators, limiting the search space of possible functions. This information is shown in Figure 4a. This network has 1915 nodes, not considering the temporary nodes that are required in the encodings (for negating regulators and functions), of which 34 are considered input nodes since they have no incoming edges. The network starts with 1881 default functions and 1327 Not functions.

Since the nodes with one regulator are also connected through the default function, the repair $g$ (which allows changing from And to Or and vice-versa) does not change the output of the function. Moreover, these unary functions have only one possible change, being negated. Either by negating the default function creating a Nand or a Nor (repair $n$), or by negating the only regulator (repair $i$). Considering unary functions as a class of its own may increase the performance, making it possible to change the unary function to a Not function.

The tests were run to find the cardinality minimal repair, although sometimes due to the time and memory limits it was impossible to find an optimal solution.




(a) (b)

Fig. 5: (a) The number of repairs necessary to correct the Stat vs Exp data set for the repairs *en, gi, gn, in, egn* and *gin*, when running the tests with a time limit of 600 and 3600 seconds. The results shown were obtained using ASP. (b) Number of repairs necessary to correct the Stat vs Exp data set, when considering two default functions(And/Or and identity). The types of repairs that are not present did not find a feasible solution. All the repairs are feasible but not optimal. The results were obtained using ASP.

### 5.2 Stat vs Exp case study

Figure 4b shows the smallest number of identified repairs needed to correct the model (not necessarily cardinality minimal) for both default functions obtained using the ASP approach. Note that Figure 4b does not include the combinations of repairs that do not find a solution. All the functions covered by the repairs that find a solution are a superset of the set of functions covered by the repair *n*. In most cases, when considering the Or function as the default function it is necessary to apply less repairs. When considering the complete Stat vs Exp data set one can see that the repair *n* is the optimal solution for this case. It is the only one that achieves an optimal solution within the time limit. The other three types of basic repairs are not able to find any feasible solution.

Repair *n*, contrary to other repair types that find feasible solutions, has the smallest function coverage (smaller number of possible repairs to try) and, as such it finds the optimal solution faster, if it exists. It provides a good solution for nodes with the value `true` that have influencing nodes with different values. These nodes need to be corrected when the default function is an And, but are consistent when considering the Or function as the default function. The data set has some nodes with the value `false` when the influencing nodes are different. Conversely, these nodes are consistent when running with the And function but need correction with the Or function. Since the `false` nodes are less common then the `true` nodes, using the Or function as default function requires less repairs. These networks have also many nodes with the value `true` when their regulators are all `false` which can be repaired by negating the default function.

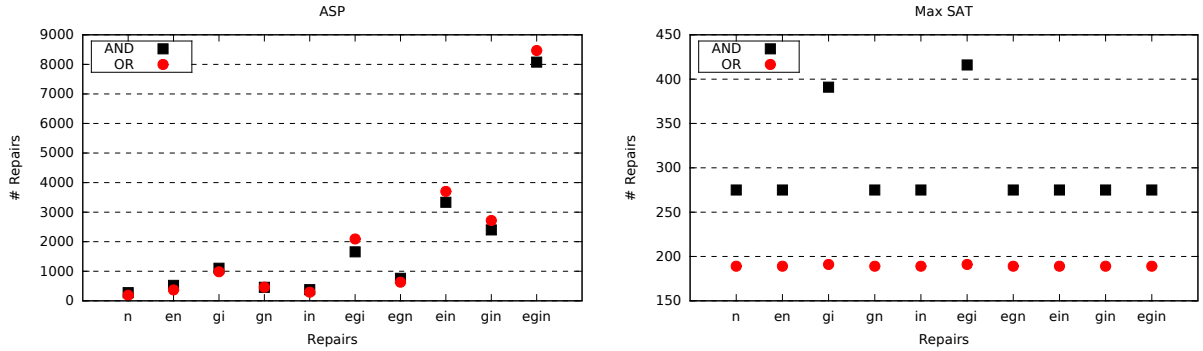
All the combinations of repairs that find a solution include the functions covered by repair *n*. Hence, if the tests are run without constraints, then they should find a solution at least as good as the solution found by repair *n*. The repairs that were closer to the number of repairs obtained by repair *n* and with the default function And, were run again with a time out of 3600 seconds. Figure 5a shows the minimal number of repairs that are needed to correct the model. However, none of these attempts reached an optimal solution. On average, the reduction in the number of repairs was sixteen, and none of the repairs came close to the number of repairs required by repair *n*.

Running the same tests for the whole data using the optimization previously explained (the use of an identity function for unary operation), one can see a few differences. First, there are no optimal solutions mainly because repair *n* is no longer valid. The disappearance of this solution is easy to explain through the fact that it is necessary to correct Identity functions, replacing them with a Not function, where previously they were transformed from And/Or into Nand/Nor. Here, only the repairs containing the repair *g* are possible. The results obtained in this test are presented in Figure 5b. The difference in the required number of repairs between the And function and the Or function as default functions is minimal.

When running exactly the same tests, *i.e.* the data from *Stat vs Exp* data set, with the limit of 600 seconds but using the MaxSAT solution are shown in Table 2. The number of repairs needed is smaller

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| AND | 198 (O) | 198 (O) | 358 | 198 (O) | 198 (O) | 289 | 275 | 285 | 198 (O) | 280 |
| OR | 150 (O) | 150 (O) | 151 (O) | 150 (O) | 150 (O) | 151 (O) | 150 (O) | 150 (O) | 150 (O) | 150 (O) |
| | n | en | gi | gn | in | egi | egn | ein | gin | egin |

Table 2: Number of repairs necessary to correct the Stat vs Exp data set with MaxSAT. The types of repairs that are not present did not find a feasible solution. (O) represent an optimal solution.




(a) (b)

Fig. 6: (a) Number of repairs necessary to correct the HeatShock data set, using ASP. Repair $n$ is the only one to find an optimal solution. (b) Number of repairs necessary to correct the HeatShock data set, using MaxSAT. In this case only the repair *gi* and *egi* are not optimal for the default function And. Note that in both graphs, the types of repairs that are not present did not find a feasible solution.

since in most cases are able to find an optimal solution. Even when an optimal solution is not found the number of repairs is smaller. Considering the repair operations that find an optimal solution one can see that it is easier to repair the model considering the Or functions as the default function.

### 5.3 HeatShock case study

Considering the complete data set of the HeatShock experimental profile, one can see that the result is similar to the one referring to the Stat vs Exp data set (Figure 6a). The repair $n$ is the only one to find an optimal solution, and it requires a lower number of repairs. It could be possible that with a larger time limit, a different combination would improve the results. But as it can be seen in Figure 6b no repair can find a better solution. Figure 6a shows the results for the repairs that find a feasible solution. In this case, the difference between the And function and the Or function as default functions seems marginal. However, it is possible to see in Figure 6b that the Or function as default requires less repairs. The repair $n$ is the repair that finds a minimal number of repairs. Combining it with other repair operations does not improve the result. Similar to what happens in *Stat vs Exp* data set, the repair $n$ is the one requiring the smallest number of repairs. The closest repair without containing repair $n$, repair *gi*, requires a bit more repairs.

## 6 Conclusion and Future Work

We propose two methods to repair Boolean networks, which are capable of repairing functions with any number of regulators. The proposed repairs are based on function transformations. These repairs are able to find feasible solutions to all real biological regulatory and signalling networks tested. From these results it is possible to conclude that, of all types of repairs tested, negating functions is the best approach for these types of networks. This repair is particularly efficient since there are many nodes with the value `false` when their regulators are all `true` and also there are nodes with the value `true`/`false` when the corresponding function is And/Or and their regulators have different values. The MaxSAT approach is clearly faster and is able to find a larger number of optimal answers then the ASP implementation. However, the ASP implementation is easier to change if one wants to add new functionalities.

As future work, the choice between different possible repairs could be explored in order to give priority to biologically relevant repairs. Also, a possible direction is the reduction of the search space by adding biological rules that better characterize the repairs allowed, therefore filtering the space of possible repairs.

Additionally, it should be possible to filter the type of functions allowed in the input, *e.g.* making it necessary to be written in a normal form and easier to evaluate the coverage of the functions when repairing.

Finally, the set of default functions to be considered could also have a biological support, like considering a disjunction over the activators in conjunction with the conjunction of the negated inhibitors, i.e. a target is active if there is at least one activator and none of its inhibitors.

**Acknowledgements** This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013. Alexandre Lemos was supported by FCT project grant EXCL/EEI-ESS/0257/2012.

## References

1. Chaouiya, C.: Petri net modelling of biological networks. Brief. Bioinform. 8(4), 210–219 (2007)
2. Siegel, A., Radulescu, O., Le Borgne, M., Veber, P., Ouy, J., Lagarrigue, S.: Qualitative analysis of the relation between DNA microarray data and behavioral models of regulation networks. Biosystems 84(2), 153–174 (2006)
3. Glass, L., Kauffman, S.: The logical analysis of continuous, non-linear biochemical control networks. Journal of Theoretical Biology 39(1), 103–129 (1973)
4. Thomas, R., Thieffry, D., Kaufman, M.: Dynamical behaviour of biological regulatory networks: I. Biological role of feedback loops and practical use of the concept of the loop-characteristic state. Bull. Math. Biol. 57(2), 247–276 (1995)
5. Didier, G., Remy, E., Chaouiya, C.: Mapping multivalued onto Boolean dynamics. Journal of Theoretical Biology 270(1), 177–184 (2010)
6. Mobilia, N., Rocca, A., Chorlton, S., Fanchon, E., Trilling, L.: Logical modeling and analysis of regulatory genetic networks in a nonmonotonic framework. In: IWBBIO. LNCS, vol. 9043, pp. 599–612 (2015)
7. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer set solving in practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers (2012)
8. Chang, C.L., Lee, R.C.T.: Symbolic logic and mechanical theorem proving. Academic press (2014)
9. Guerra, J., Lynce, I.: Reasoning over biological networks using maximum satisfiability. In: Principles and Practice of Constraint Programming. pp. 941–956. Springer Berlin Heidelberg (2012)
10. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning, pp. 466–483. Symbolic Computation, Springer (1983)
11. Comtet, L.: Advanced combinatorics: The art of finite and infinite expansions. p. 187. Springer, Holland (1974)
12. Allen, T.E., Herrgård, M.J., Liu, M., Qiu, Y., Glasner, J.D., Blattner, F.R., Palsson, B.Ø.: Genome-scale analysis of the uses of the escherichia coli genome: model-driven analysis of heterogeneous data sets. Journal of bacteriology 185(21), 6392–6399 (2003)
13. Bradley, E.H., Curry, L.A., Devers, K.J.: Qualitative data analysis for health services research: Developing taxonomy, themes, and theory. Health Serv Res 42(4), 1758–1772 (2007)
14. Gama-Castro, S., Jiménez-Jacinto, V., Peralta-Gil, M., Santos-Zavaleta, A., Peñaloza-Spinola, M.I., Contreras-Moreira, B., Segura-Salazar, J., Muñiz-Rascado, L., Martínez-Flores, I., Salgado, H., et al.: Regulondb (version 6.0): gene regulation model of escherichia coli k-12 beyond transcription, active (experimental) annotated promoters and textpresso navigation. Nucleic acids research 36(suppl 1), D120–D124 (2008)
15. Gebser, M., Guziolowski, C., Ivanchev, M., Schaub, T., Siegel, A., Thiele, S., Veber, P.: Repair and prediction (under inconsistency) in large biological networks with answer set programming. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13 (2010)
16. Roussel, O.: Controlling a solver execution with the runsolver tool. JSAT 7(4), 139–144 (2011)
17. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In: Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR, Vancouver, Canada, May 16-19. Proceedings. pp. 345–351 (2011)
18. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence 187-188, 52–89 (2012)
19. Martins, R., Manquinho, V.M., Lynce, I.: Open-wbo: A modular maxsat solver,. In: Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. pp. 438–445 (2014)