Repairing Boolean regulatory networks using Answer Set Programming

Alexandre Lemos, Pedro T. Monteiro and Inês Lynce
{alexandre.lemos,pedro.tiago.monteiro,ines.lynce}@tecnico.ulisboa.pt

Last modified: August 10, 2016

Abstract

Models of biological regulatory and signalling networks are increasingly used to formally describe and understand complex biological processes. Such models are often repaired 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.

In this work, we describe biological regulatory and signalling networks using the Boolean formalism, where nodes are represented by Boolean variables denoting biological components and edges denote regulatory interactions between components. The evolution of each variable is defined by a Boolean function depending on the values of the regulators of the component.

Here, 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 is tested using data from *Escherichia coli* and *Candida albicans* organisms. Interestingly, the system finds adequate solutions to ensure consistency for all observations.

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

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 [3], piecewise-linear differential equations [11], Sign Consistency Model (SCM) [14] or the logical formalism [15]. 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).

The SCM formalism describes the relation between components through an influence graph, where edges represent interactions between components labelled by their sign. The value of a component is defined as the product of a regulator value with the sign of the corresponding interaction. Consequently, a component can be fixed at different values whenever different regulators permit it, rendering a given model too permissive to different experimental observations. Nevertheless, Gebser *et al.* [9] successfully applied this formalism for the repair and prediction of biological regulatory and signalling networks at steady state, using Answer Set Programming (ASP). ASP is a form of declarative programming, similar to Prolog, that uses logic semantics to solve search problems.

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 use an ASP-based approach to check its consistency that allows the model to be revised. 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 and ASP. In Section 3 our approach will be described and explained. Section 4 describes the ASP encoding[1]. In Section 5 we test the proposed approach with three real data sets and discuss the results.

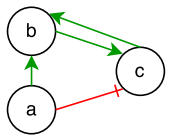
2 Preliminaries

2.1 Boolean regulatory graphs

A Boolean regulatory graph is defined by:

- a set of regulatory components $G = \{g_0, \cdots, g_n\}$, where each is associated with a Boolean variable representing the level (of expression or activity) of the component;
- a set of edges E, where $(g_i,g_j)_{i,j=1,\ldots,n} \in E$ denotes a regulatory interaction between components g_1 and g_2, *i.e.*, g_1 is a regulator of g_2;

[1]The complete encoding is available at `http://web.ist.utl.pt/~alexandre.lemos/rbnasp/`



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

$$K_b(x) = x_a \vee x_c \qquad K_b(x) = \neg(x_a \vee x_c)$$
$$K_c(x) = x_b \wedge \neg x_a \qquad K_c(x) = \neg(x_b \wedge \neg x_a)$$

Figure 1: The representation of 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).

- to each component g_i there is an associated regulatory logical function K_i which defines its value based on the value of its regulators. Components without regulators are denoted inputs and have constant values.

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 can be 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 (see [12] among others). An ASP program is defined by a set of rules, where each 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 $\leftarrow$ denoted as the head of the rule and the right side denoted 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 literal $\sim l_{m+1}$ 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 part of 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 for elements of the Herbrand universe 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.

Input		Functions															
A	B	A	B	$\neg A$	$\neg B$	$A \wedge B$	$\neg A \wedge B$	$A \wedge \neg B$	$\neg A \wedge \neg B$	$A \vee B$	$\neg A \vee B$	$A \vee \neg B$	$\neg A \vee \neg B$	$A \oplus B$	$A \equiv B$	T	F
0	0	0	0	1	1	0	0	0	1	0	1	1	1	0	1	1	0
0	1	0	1	1	0	0	1	0	0	1	1	0	1	1	0	1	0
1	0	1	0	0	1	0	0	1	0	1	0	1	1	1	0	1	0
1	1	1	1	0	0	1	0	0	0	1	1	1	0	0	1	1	0

Table 1: All possible combinations of Boolean functions with two arguments.

3 Repairing Boolean regulatory networks

Figure 1 shows a network (left) with the corresponding Boolean functions (center). The edge *(a,c)* is the only negative influence on the network showed in Figure 1 and so the corresponding Boolean rule has x_a negated.

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

Considering the model described in Figure 1 and the example 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 regulators b and c.

Also, when one combines the repair i, which allows the negation of a function regulators (only allows the negation of regulators that have not been previously negated), with the repair g, the output will be more general than when applying only the repair n. The functions NAND and NOR are a subgroup of the functions produced when combining those repairs.

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

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

Table 2: Possible replacements for function ¬ A ∧ B and which repairs are used to achieve them.

```
funcOr(1,b).        funcOr(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)).
```

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

the number of arguments of the function [4]. Table 1 presents all the possible combinations of functions with two arguments.

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 2 shows the different combinations of repairs needed to convert the binary function $\neg A \wedge B$ into a different one (whenever possible).

4 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 and the predicate `obs_vlabel(P,V,S)` to give to each node an observed value `S` in an experimental profile `P`. Storing profiles, allows us to handle multiple observations at the same time.

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 are shown in Figure 2. The predicate `temp` is used to defined 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 lines (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

```
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)
```

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

the node have the value `true`. The predicate `edge(u,v)` represents an edge between the node `u` and `v`, *i.e.* a node `u` is a regulator of `v`.

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`.

Now, 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.

Line (4) shows the possibility of creating a NAND function; to use this repair it is necessary to have the repair active ($repair_n$), the existence of 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 not removed edges for the given node (line 5).

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

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*. The tests were also used to access the performance to obtain the needed repairs.

5.1 Experimental Setup

The data sets *HeatShock* [1] and *Stat vs Exp* [2] were used for evaluating the proposed approach. The network of *Escherichia coli* was obtained from `RegulonDB` [6]. The *heatShock* data set describes the behaviour of *Escherichia coli* response to an increase of temperature. The *Stat vs Exp* data set corresponds to Exponential-Stationary growth shift study of *Escherichia coli*. These data sets were obtained from Gebser *et al* [9], and

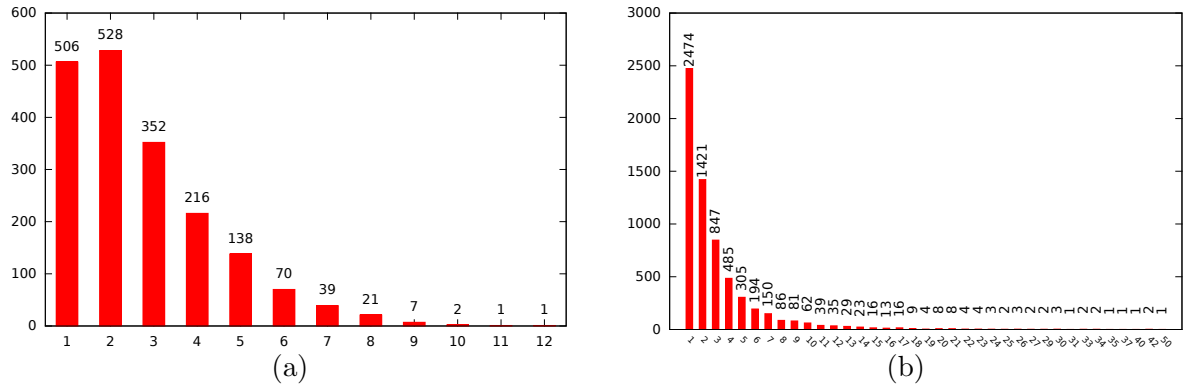


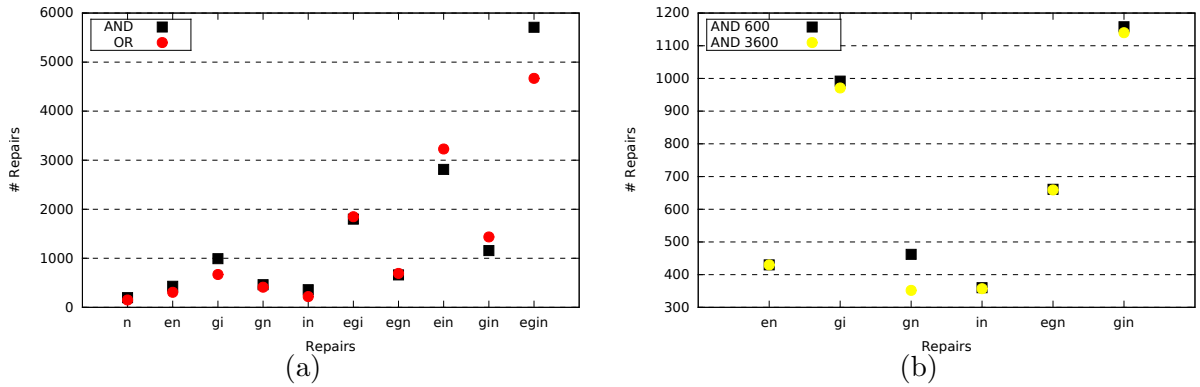
Figure 4: Number of regulators per node for *Escherichia coli* model (left), and the number of regulators per node for *Candida albicans* model (right).

are divided in six different percentages of data: 3%, 6%, 9%, 12%, 15% and 100%. For the first five percentages, there are 200 different files with a randomly picked sample of the data. For the last percentage, there is only one file since the file includes all the observations of the corresponding data set. Using these different percentages one can understand better the behaviour of the different implemented tools considering different amount of data. It is easier to satisfy the experimental data in a 3% then in a 100% file since it has less constraints.The evolution of the repairs can also be studied based on the different results obtain for each percentage. Since the models were imported from the SCM, and this formalism does not have a Boolean function associated to each node, they were adapted by considering that all nodes have the same (default) function. In order to understand better 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 [13] with a time out of 600 seconds and 3 Gb of memory. The program was executed using *Gringo*(version 4.5.4) [10] and *Clasp*(version 3.1.4) [8] 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 temporary vertex that are required in the encodings (for negating regulators and functions), of which 34 are considered input since they have no incoming edges. The encoding in ASP requires the use of additional vertexes achieving the grand total of 3242 vertexes. The network starts with 1881 default functions and 1327 NOT functions.

The regulatory network for *Candida albicans* is larger but still most nodes have a small number of regulators, as it i shown in Figure 4b. This network has 6410 nodes not considering temporary vertex that are required in the encodings (for negating regulators and functions), of which 71 are considered input since they have no incoming edges.The encoding in ASP requires the use of additional vertexes achieving the grand total of 17184



(a) (b)

Figure 5: (a) 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. (b) 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.

vertexes. The network starts with 6339 default functions and 10774 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 result. Moreover, these unary functions have only one possible change, being negated. Either by negating the default function creating a NAND or 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.

5.2 Stat vs Exp case study

Figure 5a shows the number of repairs needed to correct the model (not necessarily cardinality minimal) for both default functions. Note that the Figure 5a 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 OR 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 a solution.

Repair n, contrary to other repair types that find feasible solutions, has the smallest function coverage (smaller number of possible repairs to try), 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 with the default function as AND, but are consistent when considering OR 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 but need

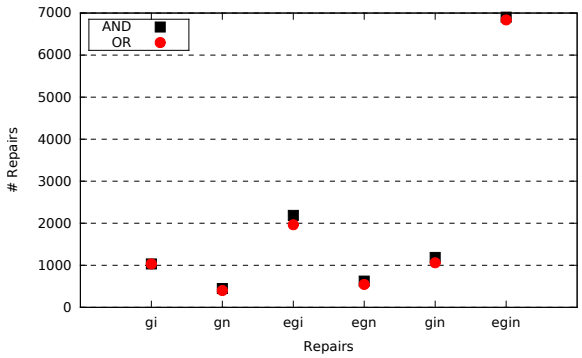


Figure 6: The number of repairs necessary to correct the Stat vs Exp data set, when considering two default functions(And/Or and identity). The types of repairs that are not present did not find a feasible solution. All the repairs are feasible but not optimal.

correction with Or. Since the `false` nodes are less common then the `true` nodes, using Or 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 and so if the tests are run without constraints they should find a solution at least as good as the one 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 retested with a time out of 3600 seconds. Figure 5b 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 these came close to the number of repairs required by repair n.

The repair n is the only repair that obtained the optimal solution and so in an attempt to optimize the solution available in the repairs containing the repair n an upper bound was added to the program. The upper bound was the number of repairs needed to repair the model using only repair n. This reduces the search space, but with the time out of 600 seconds no solution was obtained.

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 by the fact that it is necessary to correct unary functions, making them Not, 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 6. The difference between And and Or as default functions is minimal.

5.3 HeatShock case study

Considering the complete data set of the heatShock experimental profile one can see that the result is every similar to the one resulting from Stat vs Exp data set. The repair n is the only one to find an optimal solution, and it requires a lower number of repairs. It is possible that with a larger time limit, a different combination will improve the results.

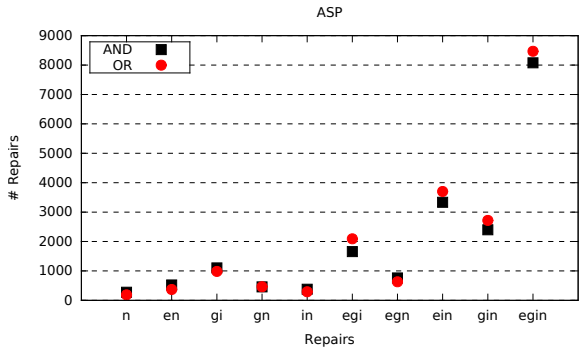


Figure 7: The number of repairs necessary to correct the heatShock data set. The types of repairs that are not present did not find a feasible solution. Repair n is the only one to find an optimal solution.

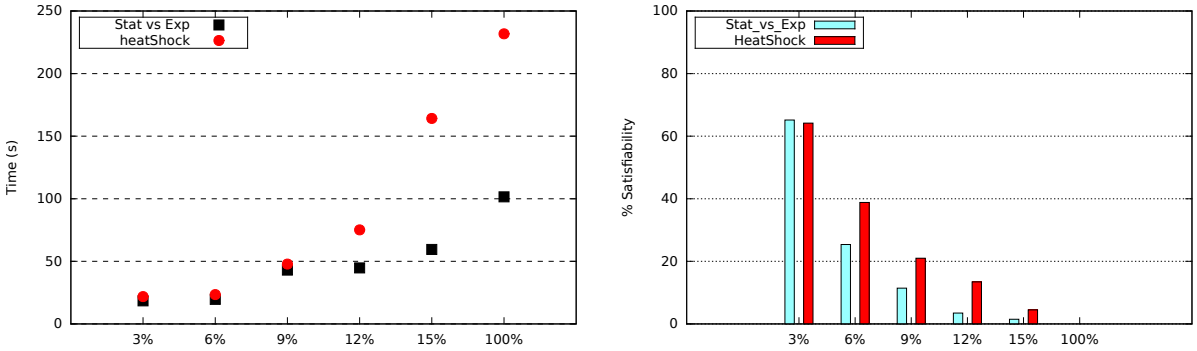


Figure 8: (a) The time required to find an optimal solution for repair n when running with Or as default for the different percentages. (b) The percentage of satisfiable tests for the different percentage of data, using repair g, when considering And as default function.

Figure 7 shows the results for the repairs that find a feasible solution. In this case, the difference between the And and Or as default functions is marginal.

Figure 8a shows the evolution of the time needed to find an optimal solution using repair n using different percentages of data. The time it took to find the optimal solution, when running with Or as the default function, is similar in both data sets but for *heatShock* it is slightly higher when using the full data set. The time needed to find an optimal solution increases proportionally with the percentage of data considered in the data set.

Repair g is not able to find any solutions when considering the full data set but when considering only a fraction it finds solutions. Figure 8b shows the percentage of satisfiable tests for the different percentage of data using repair g and considering And as the default function. One can see that heatShock data set has a bigger satisfiability rate but both data sets reduce the percentage of satisfiable tests when the percentage of data rises.

It is normal that repair g can find some solutions because as it was said before the data set has nodes that are satisfiable with Nand, and part of them are satisfiable with

	egi	ein	egin
AND	925	3225	6126
OR	809	3302	6212

Table 3: The number of repairs necessary to correct the model when considering both data sets (Stat vs Exp and heatShock). Only the combination of repairs that find a feasible solution are shown, none of each is optimal.

an OR; that is why using the OR as default requires less repairs.

When considering both data sets (combining heatShock and Stat vs Exp) there are no repair operations that find an optimal solution within the time limit.As previously discussed using the repair *egi* one can produce exactly the same function as the repair *egin* but it may require more repairs to do it ($\neg(A \wedge B) = \neg A \vee \neg B$). The repair *ein* covers less functions but still achieves a solution within the time limit. Table 3 shows the number of repairs required for repairing the model, none of which are cardinality minimal. The repair *egin* will at least require the same number of repairs as the repair *egi* as it also includes theses repairs. Even that theses two repairs can only achieve the same number of functions the *egin* has more possible repairs to check. Once again one can see that the model with OR as a default function requires less repairs.

5.4 *Candida albicans* case study

In order to test a model of a different organism, we considered the *Candida albicans* fungus. Here, since no experimental observations are available, the program tries to generate data which is consistent with the model. If the program is no able to generate data consistent with the model, the model is incorrect and needs to be revised. It is natural that initially the model does not require any repairs since the model was constructed based on consistent data. In order to test the repairs and see which corrects this network better, some experimental observations were randomly generated considering changes in 50% of the data set. These observations were constructed based on the data from the output of the program when run without any initially profile. The tests were run in the same conditions as for the previous data sets.

Figure 9 shows the average number of repairs for the fifteen randomly generated data sets for both AND and OR as the default function. Only the repair *n* achieves an optimal solution. The repairs *e, i, g, eg* do not find any feasible solutions. The combinations of repairs *ein, gin, egin* exceeded the memory limit.

Considering the Sign Consistency Model (SCM) approach [9] for the heatShock data set, the combination of the three model repairs *aeg* (adding edges, flipping signs and making node inputs) find a non optimal solution in the 600 seconds time limit. Here, the proposed methodology is able to solve all the data sets without compromising much the performance since the majority of the execution time of the tests are comparable with the execution times from the most difficult combination of repairs in the SCM version.

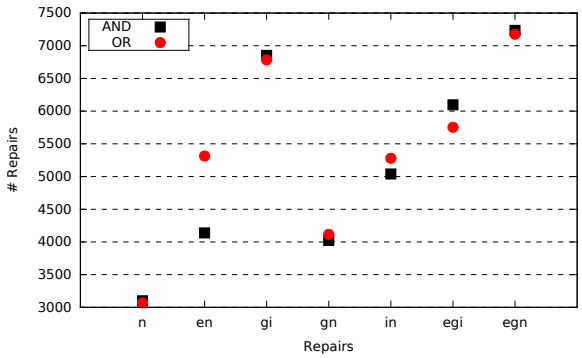


Figure 9: The number of repairs, in average, necessary to correct the *Candida albicans* model. The types of repairs that are not present did not find a feasible solution or exceeded the memory limit and so no data is available. Repair n is the only one to find an optimal solution.

6 Conclusion and Future Work

We propose an ASP-based method to repair Boolean networks, which is 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.

Previous work in repairing Boolean networks, considered the SCM formalism [9], which explains the value of the regulated component in a simplified manner. Here, we show that it is possible to perform repairs with a more fine grained regulatory function, without compromising performance. It was not possible to discover if the combination of repairs proposed would help finding a better solution since the search space was too large for the considered time limit. Giving a value for the maximum number of repairs allowed did not help to significantly reduce the search space.

As future work, the choice between different possible repairs could be explored in order to give priority to biologically relevant repairs. For example, while negating functions may be particularly efficient it may not be the most biologically relevant repair, since it may change the sign of a regulator. 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. Pedro T. Monteiro was supported by FCT grant IF/01333/2013.

References

[1] Timothy E Allen, Markus J Herrgård, Mingzhu Liu, Yu Qiu, Jeremy D Glasner, Frederick R Blattner, and Bernhard Ø Palsson. 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.

[2] Elizabeth H. Bradley, Leslie A. Curry, and Kelly J. Devers. Qualitative data analysis for health services research: Developing taxonomy, themes, and theory. *Health Serv Res*, 42(4):1758–1772, aug 2007.

[3] C. Chaouiya. Petri net modelling of biological networks. *Brief. Bioinform.*, 8(4):210–219, 2007.

[4] Louis Comtet. *Advanced Combinatorics: The art of finite and infinite expansions*. Springer, Holland, 1974.

[5] Gilles Didier, Elisabeth Remy, and Claudine Chaouiya. Mapping multivalued onto Boolean dynamics. *Journal of Theoretical Biology*, 270(1):177–184, 2010.

[6] S Gama-Castro, H Salgado, A Santos-Zavaleta, D Ledezma-Tejeida, L Muñiz Rascado, JS García-Sotelo, K Alquicira-Hernández, I Martínez-Flores, L Pannier, JA Castro-Mondragón, A Medina-Rivera, H Solano-Lira, C Bonavides-Martínez, E Pérez-Rueda, S Alquicira-Hernández, L Porrón-Sotelo, A López-Fuentes, A Hernández-Koutoucheva, VD Moral-Chávez, F Rinaldi, and J Collado-Vides. RegulonDB version 9.0: high-level integration of gene regulation, coexpression, motif clustering and beyond. *Nucleic Acids Research*, 44(D1):D133–43, 2016.

[7] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[8] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.

[9] Martin Gebser, Carito Guziolowski, Mihail Ivanchev, Torsten Schaub, Anne Siegel, Sven Thiele, and Philippe Veber. 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*, 2010.

[10] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* series 3. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR, Vancouver, Canada, May 16-19. Proceedings*, pages 345–351, 2011.

[11] L. Glass and S. Kauffman. The logical analysis of continuous, non-linear biochemical control networks. *Journal of Theoretical Biology*, 39(1):103–129, 1973.

[12] Nicolas Mobilia, Alexandre Rocca, Samuel Chorlton, Eric Fanchon, and Laurent Trilling. Logical modeling and analysis of regulatory genetic networks in a non-monotonic framework. In *IWBBIO*, volume 9043 of *LNCS*, pages 599–612, 2015.

[13] Olivier Roussel. Controlling a solver execution with the runsolver tool. *JSAT*, 7(4):139–144, 2011.

[14] Anne Siegel, Ovidiu Radulescu, Michel Le Borgne, Philippe Veber, Julien Ouy, and Sandrine Lagarrigue. Qualitative analysis of the relation between DNA microarray data and behavioral models of regulation networks. *Biosystems*, 84(2):153–174, 2006.

[15] R. Thomas, D. Thieffry, and M. Kaufman. 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.