UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Model Revision of Boolean Logical Models of Biological Regulatory Networks

João Filipe Rosado Gouveia

Supervisor : Doctor Pedro Tiago Gonçalves Monteiro
Co-Supervisor : Doctor Maria Inês Camarate de Campos Lynce de Faria

Thesis approved in public session to obtain the PhD Degree in Computer Science and Engineering

Jury final classification: Pass with Distinction

2021

TÉCNICO LISBOA

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Model Revision of Boolean Logical Models of Biological Regulatory Networks

João Filipe Rosado Gouveia

Supervisor : **Doctor Pedro Tiago Gonçalves Monteiro**
Co-Supervisor : **Doctor Maria Inês Camarate de Campos Lynce de Faria**

Thesis approved in public session to obtain the PhD Degree in Computer Science and Engineering

Jury final classification: Pass with Distinction

Jury

Chairperson : Doctor Joaquim Armando Pires Jorge,
Instituto Superior Técnico, Universidade de Lisboa

Members of the Committee :

Doctor João Alexandre Carvalho Pinheiro Leite,
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Doctor Carito Guziolowski,
École Centrale de Nantes, France

Doctor Susana de Almeida Mendes Vinga Martins,
Instituto Superior Técnico, Universidade de Lisboa

Doctor Pedro Tiago Gonçalves Monteiro,
Instituto Superior Técnico, Universidade de Lisboa

Funding Institutions

FCT - Fundação para a Ciência e a Tecnologia

2021

Resumo

Processos celulares complexos podem ser representados por redes regulatórias biológicas. Modelos computacionais destas redes são essenciais para um melhor entendimento dos processos biológicos correspondentes, permitindo a reprodução de comportamentos conhecidos, bem como o teste de hipóteses ou simulação de comportamentos. No entanto, a construção destes modelos é ainda uma tarefa principalmente manual, e consequentemente propícia a erros. Para além disso, à medida que nova informação é obtida, os modelos existentes têm de ser revistos.

Neste trabalho propomos um método de revisão de modelos, capaz de determinar conjuntos mínimos de reparações que tornam um modelo lógico Booleano, de uma rede regulatória biológica, consistente com um conjunto de dados experimentais. O método de revisão de modelos proposto tira partido de uma abordagem baseada em lógica para verificar se um modelo é consistente com um conjunto de observações experimentais. Em caso de inconsistência, o método de verificação de consistência, implementado usando Answer Set Programming (ASP), determina o número mínimo de nós do modelo que são inconsistentes e os respetivos motivos de inconsistência. É proposto um algoritmo que, em caso de inconsistência, procura por possíveis operações de reparação que tornam o modelo consistente. Neste trabalho são consideradas quatro operações de reparação: mudar uma função de regulação; trocar um tipo de interação; remover um regulador; e adicionar um regulador. O método de revisão de modelos aqui apresentado considera validar modelos lógicos Booleanos com observações de estados estáveis ou de sequências de estados. No que respeita a observações de sequências de estados, são considerados esquemas de atualizações síncronos e assíncronos.

O método de revisão de modelos proposto é testado usando cinco modelos biológicos publicados. Foram geradas versões corrompidas destes modelos, aplicando mudanças aleatórias nos mesmos. Os modelos corrompidos são depois confrontados com observações de estados estáveis, que correspondem aos estados estáveis dos modelos originais, utilizando o método de revisão de modelos apresentado. Foram também geradas diferentes observações de sequências de estados, consistentes com os modelos originais, com o fim de validar se o método de revisão de modelos é capaz de rever os modelos quando confrontados com este tipo de observações. O método proposto é capaz de reparar a maioria dos modelos corrompidos, considerando tanto observações de estado estável como de sequência de estados. Mais ainda, são produzidas todas as soluções ótimas para reparar os modelos inconsistentes.

Palavras-chave: Revisão de Modelos, Modelos Lógicos Booleanos, Redes Regulatórias, Estados Estáveis, Dinâmica (As)Síncrona

Abstract

Complex cellular processes can be represented by biological regulatory networks. Computational models of such networks are essential to have a better understanding of the corresponding cellular processes, allowing the reproduction of known behaviours, the testing of hypotheses, and the identification of predictions *in silico*. However, the construction of these models is still mainly a manual task, and therefore prone to error. Additionally, as new data is acquired, the existing models must be revised.

Here, we propose a model revision procedure, capable of providing the set of minimal repairs to render a Boolean logical model of a biological regulatory network consistent with a set of experimental observations. The proposed model revision procedure takes advantage of a logic-based approach to verify whether a model is consistent with a given set of experimental observations. In case of inconsistency, the consistency check procedure implemented using Answer Set Programming (ASP) determines the minimum number of inconsistent nodes of the model and corresponding reason of inconsistency. An algorithm to search for possible sets of repair operations to render an inconsistent model consistent is proposed. In this work, four repair operations are considered: changing a regulatory function; changing a type of interaction; removing a regulator; and adding a regulator. The model revision approach presented here, considers confronting a Boolean logical model with stable state or time-series observations. Moreover, for time-series observations, both synchronous and asynchronous update schemes are considered.

The proposed model revision approach is tested on five published well known biological models. Corrupted versions of these models are generated by performing random changes. The corrupted models are confronted with stable state observations, corresponding to the stable states of the original models, using the presented model revision procedure. Moreover, different time-series observations are generated, consistent with the original models, to assess whether the model revision approach is able to revise the corrupted model under time-series observations. The proposed method is able to repair the majority of the corrupted models, considering stable state and time-series observations. Moreover, all the optimal solutions to repair the inconsistent models are produced.

Keywords: Model Revision, Boolean Logical Models, Regulatory Networks, Stable States, (A)Synchronous Dynamics

Acknowledgments

> "It is far, far easier to make a correct program fast than it is to make a fast program correct."
>
> Herb Sutter

First of all, I would like to thank my supervisors, Professor Pedro Monteiro and Professor Inês Lynce, for all the support and guidance throughout this extraordinary journey. I feel really blessed for having you as my supervisors. Thank you for being always there, for all the discussions and ideas, and for your dedication. This fantastic journey would not be possible without you. Thank you for being such extraordinary and kind human beings. It is always a pleasure to work with you, and I hope that I have the opportunity to work with you again in the future.

I would like to thank Professor Susana Vinga and Professor João Leite for all the enlightenment, feedback and suggestions during the thesis proposal.

I would also like to thank Instituto Superior Técnico, and the Department of Computer Science in particular, for all the support and for making this work possible.

A special thanks to INESC-ID and all who are part of it, for welcoming me and providing me all the space and conditions to develop this work. In particular, I would like to thank the SAT group of INESC-ID, for all the support and good moments.

I want to thank Fundação para a Ciência e Tecnologia (FCT) for supporting this work by awarding me a PhD grant (SFRH/BD/130253/2017).

I would like to thank Professor Pedro Monteiro, Professor Inês Lynce, Professor Vasco Manquinho, Professor Francisco Santos, Professor Luis Russo, Professor Alexandre Francisco, Professor Ana Paiva, Professor Nuno Mamede, Professor Luísa Coheur, Sofia Teixeira, Alexandre Lemos, Pedro Varela, Ana de Jesus, and Vanda Fidalgo, with whom I worked, discussed ideas, and who positively contributed in some way to the success of my journey in the last years.

I want to thank my friends, in particular Ana, Afonso, Bárbara, Bernardo, Capelo, Carolina, Fábio, Luís, Rita, Rodrigo, Rui, Rute, and Tiago, for all the support, for all the good and funny moments, for making me happy. You were very important in the last years, and I hope to be there for you as you have been there for me.

Finally, but not least, I want to thank my family, my huge family, for providing me all that I needed and more, for all the support, for all the good moments, for all the endless talks, for all the fun.

Thank you all!

Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

ASP Answer Set Programming.

BCF Blake Canonical Form.

DNF Disjunctive Normal Form.

FY Fission Yeast.

ILP Integer Linear Programming.

MCC Mammalian Cell Cycle.

PKN Prior Knowledge Network.

SCM Sign Consistency Model.

SMT Satisfiability Modulo Theories.

SP Segment Polarity.

STG State Transition Graph.

TCR T-Cell Receptor.

Th T-helper.

Chapter 1

Introduction

Complex cellular processes can be represented through biological regulatory networks which describe interactions between biological compounds such as genes and proteins. Having a computational model of such networks is of great interest, allowing the reproduction of known behaviours, the test of hypotheses, and the identification of predictions *in silico*, therefore providing a better understanding of the associated biological processes.

Different formalisms have been used by the community to model biological regulatory networks, such as Ordinary Differential Equations (ODE) [1], Piecewise-Linear Differential Equations (PLDE) [2], Logical Formalism [3], Sign Consistency Model (SCM) [4], among others [2]. In this work, we consider the (Boolean) logical formalism, which has proven useful to study and analyse dynamical behaviour of biological models [3, 5–7].

The construction of these models is still mainly a manual task, performed by a modeller, and therefore prone to error. Moreover, the construction of these models rely on experimental data that is scarcely available, incomplete or inaccurate, leading to inaccurate models. Considering the Boolean logical formalism, regulatory functions representing how biological compounds interact with each other are defined as Boolean functions. Upon the construction of these models, given the usually few samples of experimental data available, there may be multiple Boolean functions that fit the experimental data, but only one is chosen for each compound, which may not be the most adequate. During the last decades, computational techniques to build networks and models from experimental data have been proposed [8–12]. However, due to few experimental data available, often incomplete or noisy, inaccurate models can still be produced.

Logic-based approaches, such as Answer Set Programming (ASP) [13, 14] and Boolean Satisfiability (SAT) [15], have been successfully used in the context of biological regulatory networks [16, 17]. In this work we consider the use of ASP to aid the analysis of Boolean logical models of biological regulatory networks.

1.1 Motivation

As models of regulatory networks are extended or new experimental data is acquired, it becomes necessary to assess whether the models continue to be consistent with the existing data, and if necessary, to revise them. Typically, model revision is a manual task, performed by a domain expert, and therefore prone to error. The process of building such models is also prone to error, thus accentuating the importance of the model revision process.

There is an inherent combinatorial problem associated to all possible changes that can be applied to a model to render it consistent. Moreover, there is a combinatorial explosion of the dynamical behaviour that can be generated by a regulatory model. This makes the model revision process a difficult task, as the model needs to be consistent with all known dynamical behaviours and properties. One crucial step in the model revision process is the possible redefinition of regulatory functions, a manual process not formally defined, and therefore prone to error. This calls for automated approaches to build and revise such models, especially when incomplete information is considered. The notion of *belief revision* exists for a long time and addresses the problem of having new information in conflict with previously known information [18]. Even though model revision has been addressed by the community for models of regulatory networks [16, 17, 19], few approaches have been proposed for (Boolean) logical models [20, 21].

1.2 Goals

In this work, our goal is to develop a model revision approach for Boolean logical models of biological regulatory networks. For the model revision approach, we introduce a consistency check procedure using ASP able to assess whether a model is consistent with a set of experimental observations. Then, we present an algorithm that is able to produce a set of repair operations that render an inconsistent model consistent.

When revising Boolean logical models of regulatory networks, our goal is to be able to confront a model with a set of stable state and time-series observations. Stable state observations represent important properties of a regulatory network, being stationary states, and time-series observations allow to represent dynamic behaviour. Regarding time-series observations, our goal is to be able to consider different update schemes used to generate the dynamic behaviour of a network. In particular, we consider synchronous and asynchronous update schemes.

As there is a combinatorial problem associated with all the changes that can render an inconsistent model consistent, we intend to define an optimisation criterion to compute optimal sets of repair operations. Moreover, we intend to consider the model construction process to

define the optimisation criterion, as well as the impact that changing regulatory function of a model has on its dynamics.

Finally, our goal is to develop a tool able to receive as input a Boolean logical model and a set of experimental observations and assess whether the model is consistent. In case of inconsistency, the tool is able to produce all the optimal sets of repair operations that render the model consistent. This will allow a modeller, or a domain expert, to use the tool to revise a model. Moreover, as the tool produces all the optimal solutions, the modeller can choose which set of repair operations to apply to an inconsistent model.

1.3 Thesis Outline

This document presents a model revision approach for Boolean logical models.

In Chapter 2 are presented the essential concepts necessary to understand the context of this work. Biological regulatory networks are described in Section 2.1 in which different modelling formalisms are defined. As Boolean functions are a crucial component of Boolean logical models, concepts and properties of Boolean functions are presented in Section 2.2. In Section 2.3, Answer Set Programming (ASP) is described.

An overview of the related work is presented in Chapter 3. Section 3.1 presents network and model inference techniques. Approaches to study and analyse biological models are presented in Section 3.2. Model revision approaches are introduced in Section 3.3.

Logic-based approaches developed during this work are presented in Chapter 4. A consistency check procedure is defined in Section 4.1. A logic-based approach to compute neighbours of monotone non-degenerate Boolean functions is presented in Section 4.2.

The detailed definition of the proposed model revision process is presented in Chapter 5. The developed model revision tool, ModRev, is introduced in Section 5.2.

An experimental evaluation of the proposed model revision approach is presented in Chapter 6. Section 6.1 defines the tests performed as well as the instances used for the evaluation. The obtained results are then shown in Section 6.2. A discussion of the results is done in Section 6.3.

This document concludes in Chapter 7 in which the contributions of this work are presented. A discussion over this work and future work proposal is done in Section 7.2.

Appendix A shows detailed results of the experimental evaluation.

A tutorial of the ModRev tool is presented in Appendix B.

Chapter 2

Background

In this chapter are introduced the main concepts necessary for this work.

Section 2.1 describes biological regulatory networks and presents the inherent concepts. Several formalisms to model such networks are also presented in Section 2.1.

Important properties of Boolean functions that are used in this work are presented in Section 2.2.

This work has an important logic-based component. Therefore, Answer Set Programming (ASP) is described in detail in Section 2.3.

2.1 Biological Regulatory Networks

A biological regulatory network is a collection of biological compounds, such as proteins, genes, metabolites, among others. These biological compounds interact with each other or with other substances in a cell, representing complex biological processes.

Typically, a biological regulatory network is represented by a *regulatory graph*.

Definition 2.1.1. A regulatory graph is a directed graph $\mathcal{G} = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is the set of nodes, and $E \subseteq \{(u, v, s) : u, v \in V, s \in \{+, -\}\}$ is the set of signed edges.

The nodes, or vertices, of a regulatory graph represent biological compounds, and the edges represent interactions between compounds. An edge $(u, v, +)$ represents a positive interaction, or activation, meaning that u activates v. An edge $(u, v, -)$ represents a negative interaction, or inhibition, meaning that u inhibits v. In either case, u is said to be a *regulator* of v. Figure 2.1 illustrates an example of a regulatory graph, where positive interactions are represented by green pointed arrows, and negative interactions are represented by red blunt arrows.

Biological networks shed some light on the corresponding complex biological processes, indicating the interactions between biological compounds. Although biological networks are able to

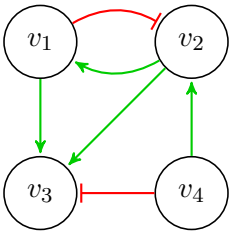


Figure 2.1: Example of a regulatory graph

represent the interaction between compounds, the information regarding how exactly the compounds influence with each other is not represented. Thus, this raises the necessity of having biological models capable of representing how biological compounds interact with each other.

Modelling biological networks is particularly useful to be able to computationally reproduce existing observations, test hypotheses, and identify predictions *in silico*. Computational modelling of regulatory networks constitutes a crucial step towards the functional understanding of the resulting intertwined networks.

Different formalisms have been used by the community to model biological regulatory networks. These models can be roughly classified as quantitative or qualitative models. Quantitative models are typically built using differential equations, requiring specific mathematical skills and a lot of detailed quantitative experimental data collected over time. However, for some biological processes, such detailed quantitative data is not available. Simpler models are necessary to overcome the lack of information: qualitative models. Qualitative models consider a discrete number, typically small (mostly Boolean), of possible states for each component, *e.g.* active / inactive.

In this work we will focus on qualitative models, in particular the (Boolean) logical models [3], described in the following Section 2.1.1.

2.1.1 Modelling

Having a computational model, representing biological regulatory networks, allows to have a better understanding of the corresponding complex biological process.

Quantitative models, usually based on differential equations, required a lot of (often unavailable) detailed quantitative data. Such models use real variables to represent the concentration of each biological compound, and equations to define the changes of the concentration values over time. Some of the quantitative models used are Bayesian Networks [22], Ordinary Differential Equations (ODEs) [1], Partial Differential Equations [2], Piecewise Linear Differential Equations (PLDEs) [2] and Stochastic Master Equation [23].

Qualitative models have proven to be well adapted for the modelling of systems where quan-

titative information is generally incomplete or noisy. Typically, network compounds only affect other compounds above (or below) some concentration threshold. With this in mind, it is possible to consider discrete variables to model regulatory networks, representing different levels of concentration.

In the following, some of the qualitative formalisms used to model biological regulatory networks are described in detail.

Logical Modelling

A simple way to define the state of a compound of a biological network is to consider a Boolean variable, rather than a real number used to define quantity in quantitative models. A Boolean variable can either be *True (1, on, active)* or *False (0, off, inactive)*. If a concentration threshold is defined for a biological compound in a regulatory network, then that compound can be represented by a Boolean variable with value *True* if its concentration is above such threshold, and with value *False* otherwise.

The use of logical models to represent biological regulatory networks was introduced by Kauffman [24] in 1969, and by Thomas [3] in 1973.

In order to have a complete model, it is necessary to define how biological compounds interact with each other. For example, considering Figure 2.1, we know that both v_1 and v_2 activate v_3, and v_4 inhibits v_3, but we lack information regarding how exactly do v_1, v_2 and v_4 affect v_3. For v_3 to become *active* it may require v_1 AND v_2 to be active. Another possibility is to just require one of v_1 OR v_2 to activate v_3. Therefore, in order to have a model of a biological network, regulatory functions are required. In a logical model, regulatory functions are defined as Boolean functions.

Definition 2.1.2. Let $\mathcal{B}$ be the set $\{0, 1\}$ and $\mathcal{B}^n$ be the n-dimensional Cartesian product of the set $\mathcal{B}$. A Boolean function f is then defined as $f : \mathcal{B}^n \to \mathcal{B}$.

Definition 2.1.3. A (Boolean) logical model $\mathcal{M}$ of a regulatory network is defined by a tuple $(V, \mathcal{F})$, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of n variables representing the regulatory compounds of the network, where to each v_i is assigned a Boolean value in $\{0, 1\}$, and $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ is the set of n Boolean functions, where each f_i define the value of v_i.

Remark 2.1.1. Considering a biological compound with k regulators, there are 2^{2^k} different Boolean functions that can be defined as regulatory function.

Figure 2.2 illustrates an example of a logical model, where the regulatory functions of each node are represented on the right side. Note that it would suffice to provide the set of regulatory

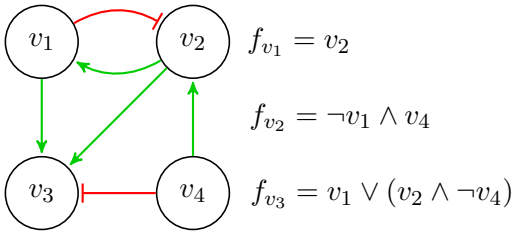


Figure 2.2: Example of a Boolean logical model

functions without the regulatory graph to define the logical model, since all the information regarding nodes and interactions can be inferred from the Boolean functions. Node v_4 does not have a regulatory function as it does not have any regulator. In this case, we denote v_4 as an *input* node.

Definition 2.1.4. An input node is a node whose regulatory function is not applied, typically representing an external stimuli. Its value does not change over time. Nodes without regulators or regulatory function are considered input nodes.

Figure 2.2 presents an example of a representation of a logical model. However, logical models can have different representations. It is possible to represent logical models as a logical circuit using logical gates to represent the Boolean functions. Figure 2.2 uses logical formulas to represent Boolean functions. However, a Boolean function can be represented by different logical formulas. For example f_{v_3} could also be represented as $f_{v_3} = (v_1 \vee v_2) \wedge \neg(\neg v_1 \wedge v_2 \wedge v4)$.

Generalised Logical Modelling

The (Boolean) logical model can be generalised by allowing the variables to have more than two values. Discrete variables can be used to define different levels of concentration, according to some threshold values.

Consider, for example, Figure 2.3, where node a affects node b above a threshold θ_1, and a affects c above a second threshold $\theta_2 > \theta_1$. We can then define three possible values for node a:

- 0: concentration level bellow θ_1

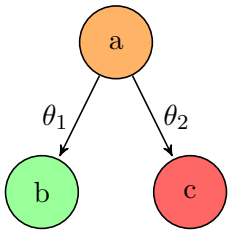


Figure 2.3: Example of a generalized logical model interaction.

- 1: concentration level between θ_1 and θ_2
- 2: concentration level above θ_2

With this in mind, we can generalise a logical model as follows:

Definition 2.1.5. A logical model $\mathcal{M}$ of a regulatory network is defined by a tuple $(V, \mathcal{F})$, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of n variables representing the regulatory compounds of the network, where to each v_i is assigned a non-negative integer value in $\{0, \ldots, max_i\}$, representing the compound concentration level, where max_i is the maximum value possible for variable v_i. $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ is the set of n regulatory functions, where each f_i define the value of v_i.

Remark 2.1.2. Given a model $\mathcal{M} = (V, \mathcal{F})$, if $\forall_{v_i}\ max_i = 1$, then all variables are Boolean variables and we are in the presence of a Boolean logical model.

Probabilistic Boolean Networks

In (Boolean) logical modelling, each compound regulated by k other compounds have 2^{2^k} possible regulatory Boolean functions. In some cases, experimental data is insufficient or there is incomplete knowledge to define such regulatory functions, where several candidates are possible. The logical modelling was extended in order to account for the uncertainty of the regulatory functions [1, 25, 26].

In a Probabilistic Boolean Network (PBN), a compound can have several regulatory functions, each with an associated probability. These probabilities are determined based on the available data, such that it is compatible with the prior knowledge of the network.

Definition 2.1.6. A Probabilistic Boolean Network model $\mathcal{M}$ is defined by a set (V), where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of n variables representing the regulatory compounds of the network, where to each v_i is assigned a Boolean value in $\{0, 1\}$. To each v_i is assigned a set $\mathcal{F}_i = \{f_1^{(i)}, f_2^{(i)}, \ldots, f_l^{(i)}\}$ of Boolean functions that can define the value of v_i. Each function $f_j^{(i)}$ is associated with a probability $c_j^{(i)}$ of being chosen to determine the value of v_i.

Remark 2.1.3. Given a PBN model $\mathcal{M}$, let $l(i)$ be the number of Boolean functions assigned to variable v_i of model $\mathcal{M}$. If $l(i) = 1$ for all $i = 1, \ldots, n$, where n is size of the network, then the PBN model reduces to a Boolean logical model.

During simulation, at each time step, for each compound, a regulatory function is randomly selected according to the corresponding probabilities, in order to determine the next state of that compound. Figure 2.4 [26] illustrates a node x_i of a PBN, associated with a set of functions

$\mathcal{F}_i = \{f_1^{(i)}, f_2^{(i)}, \ldots, f_{l(i)}^{(i)}\}$ and corresponding probabilistic values $c_1^{(i)}, c_2^{(i)}, \ldots, c_{l(i)}^{(i)}$, where only one function is chosen to predict the next state of node x_i.

Note that PBN are stochastic models, where the state of a compound not only depends on the state of the regulatory compounds, but also on a probabilistic factor that defines the regulatory function.

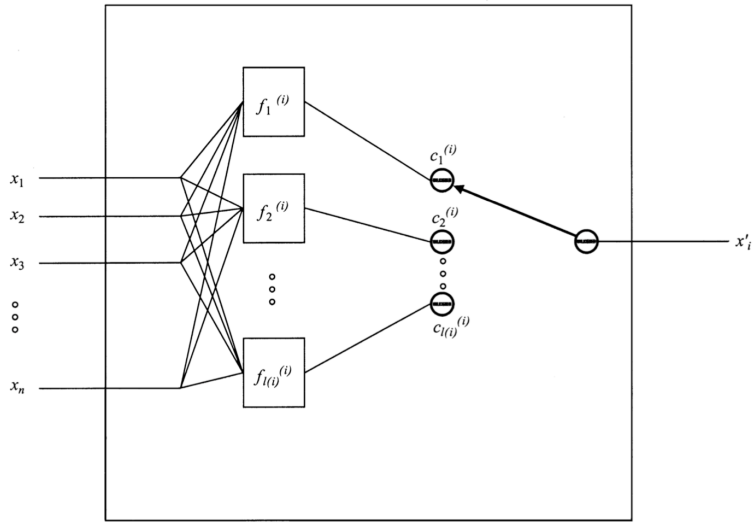


Figure 2.4: Node of a Probabilistic Boolean Network with a set of associated regulatory functions where only one is chosen to predict the next value of the node [26].

Sign Consistency Model

Siegel et al. [4] proposed the Sign Consistency Model (SCM) formalism. In this approach, it is only considered the difference in the expression levels between two situations. The expression level of a compound can increase, decrease or not change significantly.

A directed graph is built from the biological information, where each node represents a biological compound, such as mRNA, proteins or metabolites. The edges of the graph represent interactions between compounds and can be labelled "+" or "−". An edge with label "+" ("−") from a to b means that an increase of the concentration of a increases (decreases) the concentration of b. This model can contain information on indirect interactions. If one observes that a variation of the concentration of a affects the concentration of b, then an edge from a to b can be added to the model, even if the underlying mechanism of the network is not known.

In the SCM, each node of the model can assume values in $\{0, +, -\}$, where "0" means that the expression level does not change significantly, "−" means that the expression level decreases, and "+" represents an increase of the expression level. As we only know the variation sign of each compound and the interaction sign, to find relations between the signs of the variations

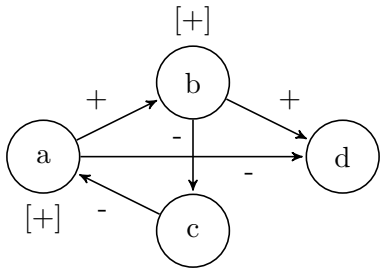


Figure 2.5: Example of a Sign Consistency Model. Observed nodes a and b are labeled with the corresponding observation.

and the signs of the interactions, the following sign algebra must be considered [4]:

$$\{+\} + \{-\} = ?,$$
$$\{+\} + \{+\} = \{+\},$$
$$\{+\} + \{0\} = \{+\},$$
$$\{-\} + \{-\} = \{-\},$$
$$\{-\} + \{0\} = \{-\},$$
$$\{+\} \times \{-\} = \{-\},$$
$$\{+\} \times \{+\} = \{+\},$$
$$\{-\} \times \{-\} = \{+\},$$
$$\{+\} \times \{0\} = \{0\},$$
$$\{-\} \times \{0\} = \{0\},$$

where $? = \{0, +, -\}$.

Although we describe the model as presented by Siegel et al. [4], there are approaches that consider an extended version, considering that each node can only have values in $\{+, -\}$, representing that an expression level tends to increase $(+)$ or decrease $(-)$. A simpler sign algebra than the one presented here is considered in the extended version of the SCM [16, 17, 27].

Figure 2.5 illustrates an example of a Sign Consistency Model of a network. Nodes a and b are observed nodes, where an increase of concentration was observed.

A set of equations can be defined from a network to represent the interactions. Considering

Figure 2.5 we can define the following equations for that network:

$$\begin{aligned} a &= -c \\ b &= a \\ c &= -b \\ d &= -a + b \end{aligned} \tag{2.1}$$

A compatible valuation of the qualitative system satisfies the following:

- All nodes are determined, i.e., have an attributed value of "+" (increased) or "−" (decreased) or "0" (did not change).
- All the *observed* nodes are given the observed values.
- The equations of the interactions are satisfied.

Considering the sign algebra, there can be equations with an unknown result. For example, considering Figure 2.5, the corresponding observation values, and the equations in (2.1), we obtain:

$$\begin{aligned} a &= + \\ b &= + \\ d &= -a + b \Leftrightarrow \\ \Leftrightarrow d &= -(+) + (+) \Leftrightarrow \\ \Leftrightarrow d &= (-) + (+) = ? \end{aligned}$$

This indetermination is called *competition*. The biological interpretation is that node d is submitted to competitive actions of different signs. If the variation of the node with competition is zero, the competition is neutral. If the variation is positive (negative), one say that the positive (negative) path wins the competition or that the competition leans towards the positive (negative) direction.

Therefore, for each node with value in $\{+, -\}$, we can say that it is sign consistent if it receives at least one influence with the corresponding sign. If there is no competition, all the influences justify the value of the node (e.g. the node receives only positive influences and therefore its value is positive). If there is competition, we can say that the path with the corresponding sign won the competition (e.g. in the previous example if d has value "−" we can say that it is justified by the negative path from a).

Others

There are other models to represent biological networks such as Automata Networks [28, 29] and Most Permissive Boolean Networks [30]. Automata Networks subsumes Boolean and multi-valued networks and are defined by a finite set of finite state machines having transitions between their local states conditioned by the state of other automata in the network [28].

Most recently, Paulevé et al. [30] introduced the Most Permissive Boolean Networks (MPBNs) paradigm, which reduces the complexity of dynamical analysis. In this paradigm, it is considered that a compound can exist in four states: inactive, increasing, decreasing, or active. MPBNs guarantee not to miss any behaviour achievable by a quantitative model following the same logic.

2.1.2 Dynamics

Biological processes evolve through time, changing the state of its compounds. Computational models of biological regulatory networks are particularly useful to analyse and simulate such changes. To simulate these behaviours, it is necessary to consider the model dynamics. The state of a network can change over time by applying the regulatory functions associated with each compound, updating its corresponding value. In a qualitative (non-hybrid) formalism, this can be done assuming discrete time-steps.

Definition 2.1.7. A state of a network with n biological compounds is a vector $S = [v_1, v_2, \ldots, v_n]$, where v_i represents the value of the variable associated with the i-th compound of the network.

Remark 2.1.4. Considering a (Boolean) logical model, the space of possible states is $S = \{0, 1\}^n$. There are 2^n different states.

The generated dynamic of a network can be represented as a State Transition Graph (STG).

Definition 2.1.8. A State Transition Graph (STG) is a directed graph $\mathcal{G}_{\mathcal{STG}} = (S, T)$, where S is the set of vertices representing the states of the network, and T is the set of edges representing possible state transitions between states.

The dynamic behaviour of a biological process can be generated through a computational model considering different update schemes. The most commonly used are the synchronous, and asynchronous update schemes. In the synchronous update scheme, all the regulatory functions are applied simultaneously, updating the value of the corresponding variable representing a biological compound. Considering the model in Figure 2.2 and represented regulatory functions, and considering that the state of the network is represented by the value of each node in ascending order of index ($S[v_1, v_2, v_3, v_4]$), the resulting STG under a synchronous update scheme

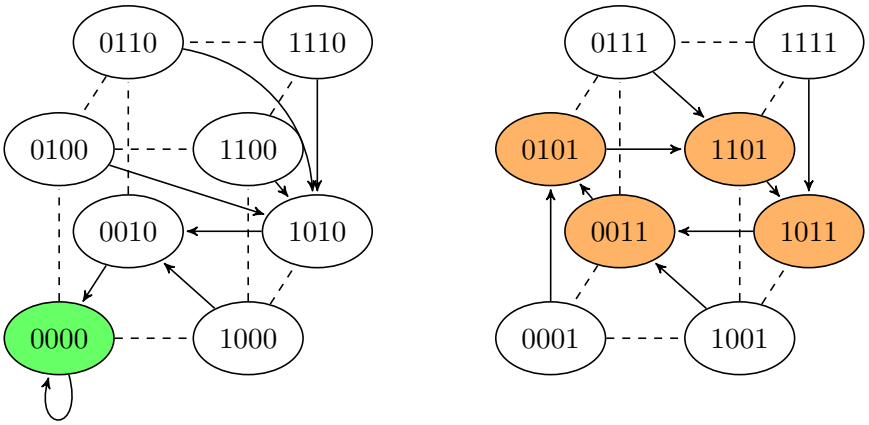


Figure 2.6: Example of a State Transition Graph for a Logical Model with four nodes, considering a synchronous update scheme. Vertices represent states of the model and arrows represent possible state transitions.

is illustrated in Figure 2.6. Note that in the synchronous update scheme, since all regulatory functions are applied simultaneously, there is exactly one state transition from each state.

Biological processes occur in nature in a non-synchronous way, where changes occur at variable speeds and rarely simultaneously. Although generating computational dynamics using a synchronous update scheme allows to study and analyse some properties of a biological regulatory network, using an asynchronous update scheme is arguably more realistic. In an asynchronous update scheme, it is assumed that all the changes and interactions occur at different times, and therefore only one regulatory function can be applied at each time-step. Note that, contrary to the synchronous update scheme, it is possible to have multiple transitions from any state. In fact, in a network with n compounds, a state can have up to n different transitions to different states. The combinatorial explosion of possible dynamic behaviours, particularly when considering asynchronous update scheme, makes the task of analyse and study the network dynamic behaviour and properties a challenging task. Considering again the Boolean logical model in Figure 2.2, and an asynchronous update scheme, the generated STG is represented in Figure 2.7.

An important property of the network dynamic is the existence of *attractors*, which denote meaningful biological properties. An attractor is a set of states of the STG such that: from each state belonging to the attractor it is possible to reach any state of that attractor through one or more state transitions; and there is no state transition from any state of the attractor to any other state outside the attractor. An attractor that contains only one state is called a *point attractor* or *stable state* and correspond to an equilibrium state of the network. Otherwise, an attractor is denoted as a *complex* or *cyclic attractor*, representing changes between the states of the network that repeat in a cycle or loop.

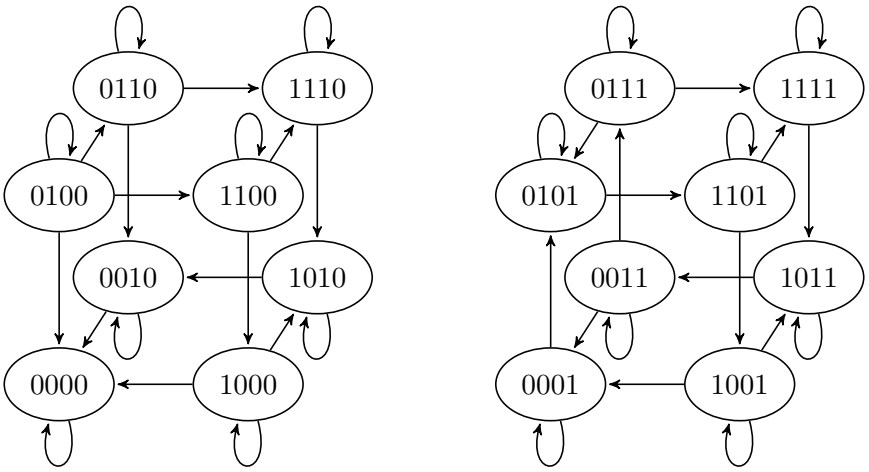


Figure 2.7: Example of a State Transition Graph for a Logical Model with four nodes, considering an asynchronous update scheme. Vertices represent states of the model and arrows represent possible state transitions.

In Figure 2.6 it is represented in green the stable state $\{\{0000\}\}$, to which the network converges when v_4 is absent (has value 0), and from which is not possible to transition to any other state. When v_4 is present (has value 1), the network converges to the cyclic attractor $\{\{0101\}, \{1101\}, \{1011\}, \{0011\}\}$, represented in orange. Note that the states of the cyclic attractor repeat in a loop, not existing any transition to any state outside the attractor, and from any state of the attractor it is possible to reach any other state of that same attractor.

Determining the attractors of a network, specially considering asynchronous update scheme, can be a difficult task due to the combinatorial explosion of possible dynamic behaviours. A more loose concept of attractors is sometimes considered: *trap spaces*. A trap space is a set of states from which there is no state transition to any other state outside the trap space [31]. A trap space is not exactly an attractor since a state of the trap space may not be reachable from other states of the same trap space. However, trap spaces can be exploited to investigate attractor properties as well as model reduction techniques, and therefore are of great interest.

Definition 2.1.9. Let S be the set of all possible network states and T be the set of state transitions. A trap space TS is defined as $TS \subseteq S : TS \neq \emptyset,\ \nexists (u, v) \in T : u \in TS, v \notin TS$.

Definition 2.1.10. An attractor is a minimal trap space. $A \subseteq S$ is an attractor if it is a trap space and $\forall_{s \in A} A \setminus \{s\}$ is not a trap space.

Remark 2.1.5. Any trap space contains at least one attractor.

Remark 2.1.6. An attractor corresponds to a terminal Strongly Connected Component (SCC) in a STG.

2.2 Boolean Functions

In this work we will consider the Boolean logical formalism to model biological regulatory networks as described in Section 2.1.1. These (Boolean) logical models have regulatory functions defined as Boolean functions (see definition 2.1.2). In order to have a better understanding of the impact of these regulatory functions in a model, some of the properties of Boolean functions are going to be taken into account.

A Boolean function can be represented in Disjunctive Normal Form (DNF) as a disjunction (OR, $\vee$) of terms, where each term is a conjunction (AND, $\wedge$) of literals and each literal is a variable or its negation (NOT, $\neg$) [15]. However, a Boolean function can have multiple equivalent DNF representations. To uniquely represent a Boolean Function, the Blake Canonical Form (BCF) can be used [32]. The BCF is a special case of DNF where a Boolean function is represented by the disjunction of all its *prime implicants*. An *implicant* of a Boolean function f is an elementary conjunction C over the variables of f such that if $C = 1$ implies $f = 1$, and is called *prime* if it is not absorbed by any other implicant (for details, see Crama and Hammer [33]). As a Boolean function is uniquely identified by the list of its prime implicants, any given Boolean function has a unique BCF representation. From now on we will consider that any Boolean function is represented in BCF.

In this work, we restrict the domain of the Boolean regulatory functions to the set of monotone non-degenerate Boolean functions. In the context of biological networks, in a monotone regulatory function each regulator only has one role: either activator, or inhibitor, but not both simultaneously. In a non-degenerate Boolean function only its regulators are present as variables of the function. If a given variable does not affect in any way the output of the regulatory function, then the variable should not be considered a regulator.

Let $\mathcal{B}$ be the set $\{0, 1\}$ and $\mathcal{B}^n$ be the n-dimensional Cartesian product of the set $\mathcal{B}$. Given $X = (x_1, \ldots, x_n) \in \mathcal{B}^n$, with $1 \leq i \leq n$, a Boolean function $f : \mathcal{B}^n \to \mathcal{B}$ is *positive* (resp. *negative*) in x_i if $f|_{x_i=0} \leq f|_{x_i=1}$ (resp. x_i if $f|_{x_i=0} \geq f|_{x_i=1}$), where $f|_{x_i=0}$ (resp. $f|_{x_i=1}$) denotes the value of function $f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$ (resp. $f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$). A given function f is *monotone* if it is either *positive* or *negative* for every x_i [33]. In the BCF representation of a monotone Boolean function, each variable x_i always appears either as a positive (x_i) or as a negative ($\neg x_i$) literal. A function f is non-degenerate if all of its variables are *essential*. A variable x_i is *essential* in f if $f|_{x_i=0}(X) \neq f|_{x_i=1}(X)$ for some $X \in \mathcal{B}^{n-1}$, and is *inessential* otherwise [34, 35].

Let f and f' be two monotone non-degenerate Boolean functions in $\mathcal{B}^n \to \mathcal{B}$, $X = (x_1, \ldots, x_n) \in$

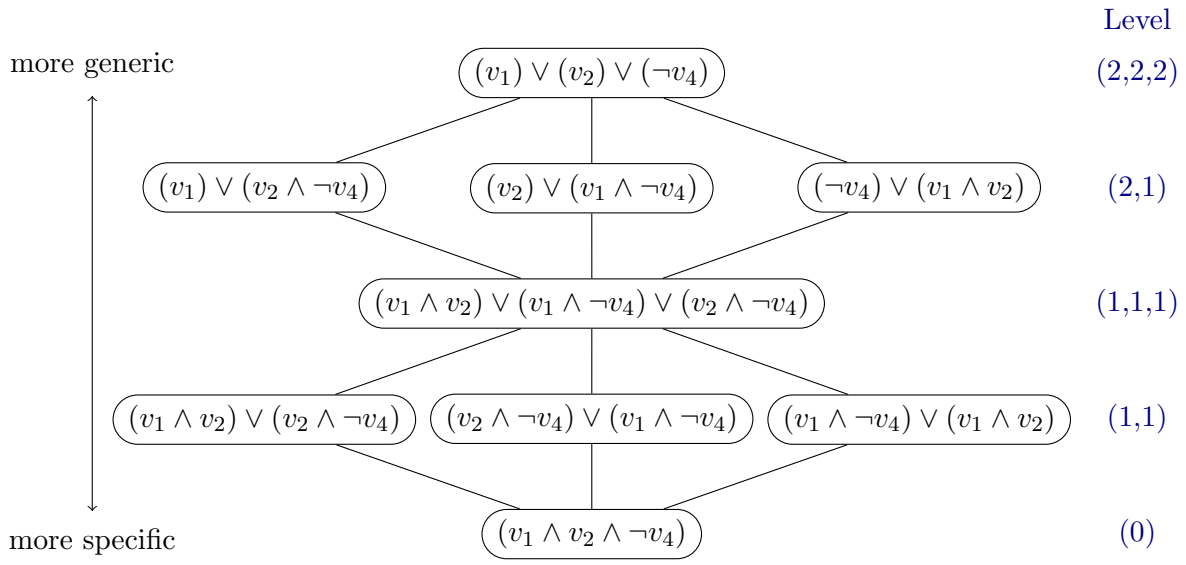


Figure 2.8: Hasse diagram for monotone non-degenerate Boolean functions of three arguments (regulators v_1, v_2, and $\neg v_4$). On the right side is represented the level of the functions.

$\mathcal{B}^n$, with the relation $\preceq$ being defined as:

$$f \preceq f' \iff f(X) = 1 \Rightarrow f'(X) = 1. \tag{2.2}$$

A partial order set can be defined over the set of all monotone non-degenerate Boolean functions in $\mathcal{B}^n \to \mathcal{B}$ with the relation $\preceq$ [33, 34], and can be represented by an Hasse diagram. Moreover, given two functions f and f', f' is a *parent* of f if and only if $f \preceq f'$ and $\nexists f''$ such that $f \preceq f''$ and $f'' \preceq f'$. In this case f is said to be a *child* of f'. Two functions are said to be *comparable* if one is an ancestor of the other, *i.e.* if there is a sequence of parents connecting the two.

Figure 2.8 illustrates an Hasse diagram over the set of monotone non-degenerate Boolean functions with three variables (v_1, v_2, and $\neg v_4$). Each node of the diagram represents a Boolean function in BCF, and each connection represents a parent-child relation. The function represented in the top of the diagram is the most generic function, which truth table contains only one entry that evaluates 0. Analogously, the function represented in the bottom of the diagram is the most specific function, which truth table contains only one entry that evaluates 1.

In [34] a set of rules is proposed to compute the *parents/children* of a given function monotone non-degenarate Boolean f without the need to compute the whole function space.

Let a monotone Boolean function f be represented by a set S where each element represents a term of the BCF representation of the function. Each element of S is a set of variables that are present in the corresponding term. For example, function $f = (v_1 \wedge v_2) \vee (v_2 \wedge v_4)$ can be represented by $S = \{\{v_1, v_2\}, \{v_2, v_4\}\}$. Let T be the set of all possible non-empty terms with n variables. Note that $|T| = 2^n - 1$.

Given a function S, a **parent** S' can be obtained by applying one of the following rules [34]:

1. $S' = S \cup \{c\}$, with $c \in \max_{\sigma \in T}(|\sigma| : \forall_{s \in S}, \sigma \not\subset s \wedge \sigma \not\supset s)$;
2. $S' = \min(S'' \cup \{\sigma\})$, with $S'' \subsetneq S$ and $\sigma \in T$ such that:
   (a) $\exists \sigma' \in S : \sigma \subset \sigma'$;
   (b) $\nexists \sigma' \in T$ and $\sigma'' \in S$ such that $\sigma \subsetneq \sigma' \subsetneq \sigma''$;
   (c) $\forall c$ satisfying rule 1, $\sigma \not\subseteq c$;
   (d) S' contains all variables;
3. $S' = \min(S'' \cup \{\sigma\} \cup \{\sigma'\})$, with $S'' \subsetneq S$ and $\sigma, \sigma' \in T$ such that:
   (a) σ and σ' satisfy all the conditions of rule 2 but condition (c);
   (b) S' contains all variables.

Given a function S, a **child** S' can be obtained by applying one of the following rules [34]:

1. $S' = S \backslash \{c\}$ with c such that $\nexists \sigma : \forall_{s \in S \backslash \{c\}}, \sigma \not\subset s \wedge \sigma \not\supset s$ such that $c \subset \sigma$;
2. $S' = (S \backslash \{c\}) \cup C$, for any c and C such that:
   (a) $C = \min_{\sigma \in T}(\{|\sigma| : forall_{s \in S \backslash \{c\}}, \sigma \not\subset s \wedge \sigma \not\supset s \; and \; \sigma \supset c\})$;
3. $S' = (S \backslash \{c, c'\}) \cup \{c \cup c'\}$ for c,c' not satisfying rules 1 or 2 and such that $c \cap c' \neq \emptyset$.

Considering the BCF representation of monotone non-degenerate Boolean functions, the notion of level of a function can be defined to establish a relation between functions allowing a comparison between functions that are not descendants or ancestors of each other [34]. The level of a function f, represented by $l(f)$, can be defined as an ordered m-tuple, where m is the number of terms in the BCF representation of the function f, and each element of the m-tuple represents the number of variables missing in the corresponding term, assuming the terms are in non-decreasing order on the size of each term. For example, the level of the Boolean function $f = (v_1) \vee (v_2 \wedge v_3)$, having three variables (v_1, v_2, and v_3), is $l(f) = (2, 1)$. Note that the function f has two terms, and its level is represented by an non-increasing ordered 2-tuple. Figure 2.8 shows the levels of the represented functions on the right side.

A total order relation $\leq$ can be defined between levels of monotone non-degenerate Boolean functions of the same dimension. Given two functions f and f' and the corresponding levels represented by $l(f) = (l_1, \ldots, l_m)$ and $l(f') = (l'_1, \ldots, l'_{m'})$, $l(f) \leq l(f')$ if and only if one of the following conditions holds:

- exists $k \in \{1, \ldots, min(m, m')\}$ such that $l_k < l'_k$, or
- $l_k = l'_k$ for all $k \in \{1, \ldots, min(m, m')\}$ and $m \leq m'$.

2.3 Answer Set Programming

ASP [13, 14, 36–38] is a declarative problem solving approach for combinatorial satisfaction problems. ASP is known to be efficient for enumerating solutions of NP problems while providing a user-friendly language to encode the problem [39].

ASP is based on the stable model (or answer set) semantics of logic programming [40]. An ASP program Π is a logic program defined as a set of *rules*. A *rule* r is defined as:

$$a_0 \leftarrow a_1, \ldots, a_n, not\ a_{n+1}, \ldots, not\ a_m.$$

where a_i is a (variable-free) literal and $not\ a_i$ is the negation of a_i. A (positive) literal is a predicate in first order logic and is also called an *atom*. The left part of the rule (left of $\leftarrow$) is called the *head* of the rule, and the right part of the rule (right of $\leftarrow$) is called the *body* of the rule.

The body of the rule r is true if all $a_1, \ldots, a_n$ are true and none of $a_{n+1}, \ldots, a_m$ can be proven to be true. Note that it is considered negation by omission, which means that an atom is considered to be false if it cannot be proven true. If the body of the rule is true, then the head of the rule (a_0) must also be true. Informally we can read the symbol "$\leftarrow$" as "if", which means that a_0 is true **if** the body of the rule is true.

Whenever a_0 is false ($\bot$), the rule has no head and is called an integrity constraint:

$$\leftarrow a_1, \ldots, a_n, not\ a_{n+1}, \ldots, not\ a_m.$$

An integrity constraint means that the body of the rule must never be true in order for the rule to be satisfied.

We can also have a rule without a body, which means that the head of the rule is always true ($a_0 \leftarrow \top$). Such rules are called *facts* and can be represented by dropping the symbol "$\leftarrow$":

$$a_0.$$

An ASP program can have multiple answer sets (or stable models). Informally, an *answer set* of a logic program is a minimal set of (positive) literals that satisfies all the rules of the program, and each (positive) literal can be proven either by a *fact* or by a *body* of a rule in

which that literal appears in the *head* of the rule. Consider, for example, the following program:

$$p \leftarrow q.$$
$$q \leftarrow not\ r.$$

This program has one answer set that is $\{q, p\}$. Considering negation by omission, it is not possible to justify r as true, and therefore r is assumed false. From the second rule of the program, as r is false, the body of the rule is true and q must be true and, therefore, present in the answer set. Then, similarly, from the first rule, p must be true as the body of the rule is true.

More formally, Gelfond and Lifschitz [40] defined a stable model as follows:

Definition 2.3.1. Let Π be a logic program. For any set M of atoms from Π, let Π_M be the program obtained from Π by deleting

1. Each rule that has a negative literal $not\ B$ in its body with $B \in M$, and

2. All negative literals in the bodies of the remaining rules.

Clearly, Π_M is negation-free, so that Π_M has a unique minimal Herbrand model. If this model coincides with M, then we say that M is a stable model of Π [40].

ASP allows the use of variables (starting with an upper-case letter). This allows the user to define a logic program in a higher level of abstraction, defining literals recurring to variables and defining domains for the variables. Consider for example the following program:

$$p(a, b).$$
$$q(X) \leftarrow p(X, Y).$$

This program has the answer set $\{p(a, b), q(a)\}$. The third rule considers that any first argument of a (true) predicate p is a true assignment for predicate q.

To solve an ASP program, *i.e.*, to compute the answer sets, it is necessary obtain a variable-free logic program. To this end, a *grounder* is used to replace the variables in the rules with all possible instantiations. Considering the previous example, a grounder would produce the

following program:

$$p(a, b).$$
$$q(a) \leftarrow p(a, b).$$
$$q(a) \leftarrow p(a, a).$$
$$q(b) \leftarrow p(b, a).$$
$$q(b) \leftarrow p(b, b).$$

The general process of solving an ASP program is to give a logic program as input to a grounder, then the grounder produces the corresponding variable-free program which is given as input to an ASP solver, which, in turn, produces the answer sets of the program, as illustrated in Figure 2.9.

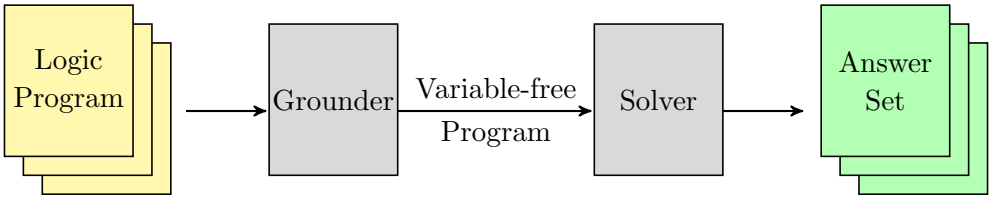


Figure 2.9: General process of solving an ASP program.

In this work we consider *clingo* (version 5.4.0) [41, 42] as the ASP system to ground and solve logic programs. Clingo uses *gringo* [43] as a grounder and *clasp* [13] as a solver.

Clingo supports several aggregate directives, such as *#count* and *#sum*, allowing the definition of more expressive rules. These directives are used to define conditions over a group of variables, and have the following general form:

$$s_1 \prec_1 \alpha\{t_1 : L_1; \ldots; t_n : L_n\} \prec_2 s_2$$

where L_i is a literal and t_i is the corresponding value associated, α is the aggregate directive, $\prec_1$ and $\prec_2$ are comparison operations ($\leq$ by default), and s_1 and s_2 are the lower and upper bound, respectively. The lower and upper bounds can be omitted. As an example of the use of the *#sum* directive, consider the following program:

$$p(1).$$
$$p(2).$$
$$q \leftarrow \#sum\{X : p(X)\} <= 5.$$

This program has the answer set $\{p(1), p(2), q\}$, since the sum of the arguments of the predicate

p is $3 \leq 5$, q must be true. However, if in the last rule, the upper bound is changed from 5 to 2, the body of the last rule fails and q cannot be explained, thus the answer set of the program will not contain q. Another interesting use of this directive is illustrated in the following program:

$$p(1).$$
$$p(2).$$
$$q(S) \leftarrow S = \#sum\{X : p(X)\}.$$

The answer set of this program is $\{p(1), p(2), q(3)\}$ where the $\#sum$ directive is used to define the argument of predicate q.

The use of *choice rules* is also supported to simulate disjunctions. Choice constructs are elements of the form:

$$s_1\{L1; \ldots; L_n\}s_2$$

that represents a disjunction of literals, where L_i are literals, and s_1 and s_2 are lower and upper bounds (and can be omitted). As an example, consider the following program:

$$p.$$
$$1\{q; r\}2 \leftarrow p.$$

The last rule of this program is called a *choice rule*. Since the body of that rule is true, the head must also be true. For the choice construct element $1\{q; r\}2$ to be true, at least one literal and at most two literals must be true, according to the bounds defined. Therefore, the program has three answer sets: $\{p, q\}$, $\{p, r\}$, and $\{p, q, r\}$.

Clingo supports the use of optimisation statements ($\#minize$ and $\#maximize$) to compute optimal answer sets. An answer set is optimal if the obtained cost is minimal (or maximal) among all answer sets, according to the optimisation statement defined. The optimisation statement is of the form:

$$\#minimize\{w_1@p_1, t_1 : L_1, \ldots, w_n@p_n, t_n : L_n\}.$$

where w_i represents the weight associated with value t_i of the literal L_i with a priority p_i (that can be omitted). The optimisation statement considers the sum of all the weights. As an

example, consider the following program:

$$1\{hotel(a); hotel(b); hotel(c)\}1.$$
$$cost(a, 80).\ star(a, 3).$$
$$cost(b, 80).\ star(a, 4).$$
$$cost(c, 90).\ star(a, 5).$$
$$\#minimize\{C@2, H : hotel(H), cost(H, C)\}.$$
$$\#maximize\{S@1, H : hotel(H), star(H, S)\}.$$

In this program, there are three hotels defined, and we want to choose only one, as indicated by the first rule. Each hotel has a cost and a star associated. The $\#minimize$ statement has the higher priority value (2), and indicates that we desire to minimise the cost C associated with the hotel. The $\#maximize$ statement, with a lower priority (1), states that we want to maximise the star value S of the hotel. The program has three answer sets, associated with each of the hotels, however there is only one optimal answer set. The hotels with the lowest cost are a and b and only these will be considered according to the first optimisation statement. Then, we want to maximise the star value, and in this case, between hotels a and b, hotel b is the one with the highest star value (4). Thus, the (only) optimal answer set for this program is the one containing hotel b as the chosen hotel.

For more detailed information regarding supported directives, such as aggregate and optimisation directives, consult the *clingo*'s user guide[1].

[1] https://potassco.org/clingo/

Chapter 3

Related Work

There are many problems and areas of interest related to biological regulatory networks. From model construction, to model analysis and revision, several approaches have been presented in the past years [44, 45].

Tools in systems biology to address some of these problems, and facilitate the work of biologists, have also been developed. Some of these tools are GINsim [46], MaBoSS [47, 48], BiNoM [49, 50], BoolNet [51], CellNOpt [52], EpiLog [53], among others [45].

In this section we describe some of the existing approaches and applications regarding biological models, from the inference of a model, to identification and verification of network properties, to the model revision process.

3.1 Network and Model Inference

Computational models of biological regulatory networks allow the study, reason over, and having a better understanding of the corresponding complex biological processes. Therefore, the construction of models, able to correctly represent biological processes, is a task of great importance.

In order to build a biological model, experimental data must be processed and analysed. However, usually few samples of experimental data are available, and it may be incomplete and inaccurate data, making the model inference a difficult task.

The model inference process depends on the type of model desired (see Section 2.1.1 for more details on different models). In this work we are focused on qualitative models.

To infer a logical model is necessary to not only infer the network, *i.e.* the interactions between compounds, but also the regulatory functions associated with each compound. The process of network inference aims to determine the regulatory graph (see Definition 2.1.1) from

a set of experimental observations and information available in literature.

Statistical learning techniques are commonly used in regulatory networks inference, such as mutual information [54–61], regression methods [62, 63], support vector machines [63], expectation-maximisation algorithms [63, 64], and Bayesian networks [8, 22, 65–67].

Having a regulatory graph, indicating the interactions and corresponding type between compounds, gives some information that can be useful. However, it is necessary to also know how exactly does each interaction affect each compound, *i.e.* what is the regulatory function of each compound. For example, a regulatory graph can indicate that compound A activates C, and that B also activates C. But how is C activated? C can be activated when both A and B are active (logical AND), or C can be activated when either of A or B is active (logical OR).

Some formalism and approaches assume a fixed regulatory function for every compound. In the SCM the regulatory function of each compound is based on the sign algebra [4]. Another regulatory rules that are commonly used take into account the number of activators versus the number of inhibitors. If a compound receives more active activations than active inhibitions, then it becomes active; if a compound receives more active inhibitions than active activations, then it becomes inactive; otherwise, if a compound receives the same number of active activations and active inhibitions, then it maintains the previous state [68, 69]. These type of regulatory functions may be useful depending on the type of model, information that it is available, or the type of analysis that is necessary to perform.

However, more expressive regulatory functions are usually necessary to accurately model a biological regulatory process. The process of inferring, from experimental data, the regulatory functions associated with each compound is not an easy task. We have to take into account that, considering Boolean logical models, there are 2^{2^k} different Boolean functions with k regulators.

Some approaches have been proposed over the years to infer logical models using different techniques. Most approaches use a previously defined regulatory graph, commonly called Prior Knowledge Network (PKN), and a set experimental data, in order to infer the regulatory functions of each compound. The inferred model must be compatible with the given PKN and consistent with the available data. However, the construction of such models from experimental data and a PKN often leads to several candidates, requiring some choices which can lead to biased predictions and inaccurate models [39].

Logic programming have been used in recent years in systems biology methods, in particular in model inference. ASP have been used to determine all the feasible models compatible with a given PKN, and characterise such models, inferring what is common or what is mutually exclusive between models [9].

Approaches to infer logical models by exhaustively enumerate all models compatible with a given set of experimental observations or a PKN using ASP have been proposed [70, 71]. Videla et al. [70] uses an hypergraph as a representation of a logical model, where the regulatory functions are represented by AND, OR and NOT gates in a regulatory graph. Then all the minimal models (hypergraphs) compatible with a given PKN are determined, *i.e.*, the hypergraphs with a minimal number of hyperedges, or models with a minimal canonical length of logical formulas.

Time-series data can also be used to infer Boolean logical models, given a PKN. Ostrowski et al. [10] considers a prior knowledge network and time-series data to determine all the compatible models using ASP. In order to do so, it considers an abstraction of a models dynamic in order to be able to consider both synchronous and asynchronous update schemes for the time-series data.

Satisfiability Modulo Theories (SMT) has also been used in the model inference procedure. In Réda and Wilczyński [72] SMT is used to infer gene regulatory networks, determining plausible regulatory scenarios that explain the observed experimental results.

Approaches using Integer Linear Programming (ILP) have been proposed and proven adequate in model inference [73, 74]. Sharan and Karp [74] presents a model inference approach that, given a PKN and experimental data, determines the best fitting model under a least squares criterion. It also allows to have incomplete PKN in the sense that it allows edges with unknown sign, *i.e.* unknown types of interactions.

3.2 Model Analysis

Computational models of biological regulatory networks are extremely useful to study network properties, make predictions, or analyse biological behaviours, providing a better understanding of the corresponding biological processes. The study of a model, in particular the study of its generated dynamic behaviours and the comparison between such behaviours and experimental observations, is of great interest. Due to the STG combinatorial explosion several formal verification techniques have been proposed to analyse dynamic behaviour.

3.2.1 Attractors Determination

The state of a network evolves through time, according to its dynamics, and tend to reach an equilibrium. These equilibriums can be stable states or sets of recurring states, and are called *attractors* (see Section 2.1.2).

Determining the attractors of a network is a very important task since attractors correspond to meaningful biological properties of great relevance. However, this is not an easy task [75].

The task of attractors determination depends on the type or size of the attractors that are being considered (point or cycle attractors), depends on the number of attractors that are being determined (all, a subset, or just one), and depends on the dynamic generations that is being considered (*e.g.* synchronous or asynchronous).

Considering only the identification of point attractors may be an easier task when compared with the identification of all the attractors of a network. Point attractors are the nodes in the STG which only have a transition to itself. However, defining the STG of a network, specially when we have large and complex networks, is a difficult task. There are some approaches that consider transforming a Boolean logical model into a SAT-network, which is a one-to-one correspondence to a SAT formula, and then exhaustively considering all possible assignments to the variables of the SAT formula in order to determine all point attractors of the network [75, 76].

Some approaches to determine attractors have been proposed using Decision Diagrams [77–79]. There is a state space explosion with the increasing size of a network, which greatly impacts the performance of some approaches to compute attractors of a network. Garg et al. [77] showed that using Binary Decision Diagrams (BDDs) to represent Boolean logical models facilitates the computation of the corresponding attractors. Naldi et al. [78] proposed an algorithm using a generalised logical formalism by means of Multi-valued Decision Diagrams, in order to determine all the point attractors of a model.

Other works define point attractors, or stable states, in a slightly different way, and use weighted MaxSAT [80] and ASP [81] to find such likely stable states. A point attractor of a network usually refers to all species in the network being at their equilibrium concentrations. The work presented in [80] and [81] considers that such attractor refers to a subset of interactions that can be consistently "on" or active. In other words, instead of defining variables for compounds of the network being present or absent, the variables are defined for the interactions between compounds being active or inactive.

Determining all attractors, including cycle attractors is a more challenging task. Dubrova and Teslenko [82] proposed a SAT-based approach to find all attractors of Boolean logical models considering synchronous update scheme. It determines any path of a given length in the STG. If the path contains at least two equal states, then an attractor is found and marked as such. If the path found did not contain an attractor, it is considered a higher path length. The process terminates when no path is found and all attractors have been identified. The work of Fayruzov et al. [69] uses Answer Set Programming (ASP) and allows the determination of all attractors considering a Markovian program in order to overcome the challenge of determining the number of time-steps needed to achieve an attractor.

Approaches to enumerate all attractors in a Boolean network were proposed by Khaled and Benhamou [83, 84]. In [83], an exhaustive search for cycles in state transitions is performed using an ASP framework, considering both synchronous and asynchronous update schemes. The presence of a cycle is detected by checking if the last seen state appears at least twice in the path considered, where the states in between any two occurrences of the last state belong to a cycle. This method allows to find any stable or unstable cycle attractor. Whenever a cycle is found, it is eliminated in the next iterations, allowing to enumerate all solutions. This approach can be memory consuming when dealing with large networks. More recently, an approach was proposed to enumerate all attractors in circular networks under asynchronous update scheme [84]. The idea is to use the notion of a circuit in the regulatory graph to determine attractors. A circuit is a path on the regulatory graph that starts and ends in the same node. Note that this circuit is on the regulatory graph and not on the state transition graph. A circuit is said to be positive if it as an even number of negative edges, and is said to be negative otherwise. Then, a positive circuit admits two attractors, and a negative circuit admits only one attractor under asynchronous update scheme [85].

3.2.2 Reachability Verification

Given a model and a set of experimental data, it is interesting to verify if the model can explain the results obtained in the experiment. In other words, if the experimental data indicates that we started at a given state and ended in other state, it is interesting to verify in the model if there is a set of state transitions from the starting state to the ending state.

Model checking [86–89] consists in verifying if experimental observations are consistent with the known model of the network. An experimental observation is consistent with the corresponding model if the model can reproduce the observation, indicating the corresponding state transitions. With larger and more complex networks it becomes intractable to verify the consistency between the experimental data and the model manually and therefore many automated techniques have been used [89].

Sometimes, verifying if an observation is consistent with a given model is not enough and it is necessary to go further. It is useful to obtain some explanation for a given observation, *i.e.*, what are the interactions triggered. It is frequently necessary to consider incomplete models. The work in [90] considers reasoning with incomplete and partial information about signal networks in order to obtain explanations and predictions. For this effect, the work of Baral et al. [90] uses the declarative programming language AnsProlog.

It is also interesting to know how can a system be influenced in order to avoid reaching

unsafe or undesired states. The work in [91, 92] introduces the notion of *bifurcation*, transitions after which a given goal is no longer reachable. The work presents a method using ASP to identify bifurcations given a model represented as a discrete finite-state of interacting compounds. However, this method, due to approximations, is not complete, *i.e.*, does not guarantee the identification of all the bifurcations, but only a subset.

3.2.3 Reduction Techniques

Analysing reachability properties or determining attractors on a logical model is computationally difficult. The state space explosion makes the generation of a model dynamic behaviour a difficult task, specially when considering large networks and/or asynchronous update schemes. For example, if we consider a logical model of a network with 20 compounds, with Boolean variables, we have $2^{20} = 1\,048\,576$ possible states. If we want to analyse its dynamic properties, considering asynchronous update scheme, then we have to consider tens of millions of possible state transitions. To overcome this problem, reduction techniques have been proposed in order to simplify the networks prior to dynamic analysis [93–95].

Naldi et al. [93] proposed a reduction method for logical models in which regulatory components are removed iteratively, updating the corresponding regulatory rules taking into account indirect effects of its regulators. The basic idea is to remove a node by connecting its regulators directly to the nodes that are regulated by the removed node. The reduced model preserve important dynamical properties of the original model such as all attractors. As trajectories in a reduced model can be related to trajectories in the corresponding original model, with this reduction method it is possible to verify reachability properties of a logical model by considering the corresponding reduced model.

Zañudo and Albert [94] presents a reduction approach that uses a topological criterion in an augmented representation of the network to identify network components that stabilise in a fixed state. These components can then be used to reduce the network size, simplifying the network.

Automata networks, which subsumes Boolean and multi-valued logical networks, have been used to reduce qualitative models to enhance the analysis of network properties [28, 29]. The reduction technique proposed by Pauleve [29] does not change the dimension of the model, but limits its degree of freedom. It identifies state transitions that do not contribute to a given reachability property, and therefore can be ignored. This will reduce the number of transitions to consider when analysing reachability problems under asynchronous dynamic.

3.3 Model Revision

When new experimental data is acquired, it is necessary to validate if the computational model is consistent with the new information, *i.e.*, if the computational model can reproduce the same experimental data. However, most models are manually built by a modeller (a domain expert) and therefore are prone to error. Even considering automated approaches to built such models, only known and most likely incomplete information is considered, and an incorrect model may still be generated (although consistent with the knowledge at the time). Given new information, a model may no longer be consistent, and therefore it should be revised and updated. The notion of *belief revision* exists for a long time and addresses the problem of having new information in conflict with previously known information [18]. However, there are few approaches to model revision of Boolean models of biological regulatory networks.

Model revision approaches were proposed over the SCM formalism [4, 16, 17]. The SCM relies on the difference between concentration levels, being different from the Boolean logical model. In the SCM, each variable has value "+" (resp. "−") if it represents an increase (resp. decrease) in the concentration level of the biological compound. Moreover, the regulation of each compound is based on sign algebra, being the sum of the products between the value of each regulator and the sign of the corresponding interaction. This type of models lack expressiveness of the regulatory functions compared with the logical model.

A first approach to model revision over Thomas' logical formalism [3] was proposed by Mobilia et al. [20]. In this work, it is considered that a model is inconsistent when it is over-constrained. Therefore, to repair an inconsistent model, it is considered the removal of some constraints, under a minimisation criterion, until the model is no longer inconsistent. This may lead to under-constrained models, not correctly representing the corresponding biological process.

Model revision usually operates under a minimal assumption as there can be several ways to repair a model. Most approaches define atomic repair operations and minimise the number of operations applied to render a model consistent. Merhej et al. [19] proposed the use of rules of thumbs, which are properties found in the literature, in order to repair inconsistent Boolean models. This work considers Boolean models where the regulatory function of each compound is as follows: it becomes active if there is at least one active activator and there is no active inhibitor; it becomes inactive if there is at least one active inhibitor and there is no active activator; otherwise its value is not changed. When repairing such models, the synchronous update scheme is considered, and the possible repair operations consist in adding or removing edges between nodes of the model.

More recently, Lemos et al. [21] proposed a model revision procedure over Boolean regulatory networks, where more expressive regulatory functions are considered when compared with the previously mentioned approaches. Boolean functions are considered as regulatory functions, represented by the combination of three basic Boolean functions (AND, OR, and IDENTITY). Moreover, the set of defined repair operations comprise: changing the type of interaction of a regulator; changing a Boolean operator from OR to AND and vice-versa; and removing a regulator. This approach does not consider the impact that changing a regulatory function has on the network dynamics. Moreover, regulatory functions are defined in a gate-like manner, where changes operate over AND and OR operators. As a consequence, not all the possible Boolean functions are considered when repairing an inconsistent model.

Chapter 4

Logic-based Approaches

Logic-based approaches, such as ASP [13, 14] and Boolean Satisfiability (SAT) [15], have been successfully used in the context of biological regulatory networks [16, 17]. In this Chapter we present an ASP approach to verify the consistency of Boolean logical models given a set of experimental observations in Section 4.1. In Section 4.2 we present an ASP approach to determine neighbours of monotone non-degenerate Boolean functions.

4.1 Consistency of Boolean Logical Models

During the iterative model construction procedure, as new data is acquired, the current model may not be able to explain the new data, and therefore needs to be revised. Typically, model construction and model revision is a manual task, performed by a domain expert, and therefore prone to error. Moreover, there is an inherent combinatorial problem associated to all possible changes to render a model consistent.

In this work, we developed a model revision procedure capable of producing all possible set of atomic repair operations that render an inconsistent model consistent. We focused on qualitative models, in specific in Boolean logical models, as described in Section 2.1.1.

In order to revise a model, a first necessary step is to check whether a given model is consistent with a set of experimental observations. In the following sections we define the model input format, the consistency check procedure developed, and the determination of inconsistencies.

4.1.1 Boolean Logical Model and Observations

The Answer Set Programming (ASP) paradigm has been used to represent biological models and proven to be useful in model inference, consistency check, and model revision approaches [4, 16, 19]. We defined the consistency check procedure as an ASP program, and therefore, the

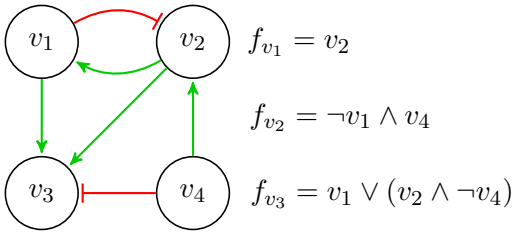


Figure 2.2: Example of a Boolean logical model (repeated from page 8)

input model and observations are defined by predicates in ASP format.

A Boolean logical model is defined by set of nodes, corresponding to biological compounds, a set of edges, corresponding to biological interactions, and a set of Boolean functions, corresponding to the regulatory functions of each compound.

To represent the nodes, the predicate `vertex(V)` is used, indicating that `V` is a node of the network. Interactions between compounds, represented as directed edges in regulatory graphs, are defined by a predicate `edge(V1, V2, S)`, corresponding to an edge from node `V1` to node `V2`, with sign `S` $\in \{0, 1\}$. An edge with sign 0 represents a negative interaction (or inhibition), and an edge with sign 1 represents a positive interaction (or activation), as defined in Section 2.1. With this predicates, one can define the topology, or structure, of the network.

For a model to be complete, it is necessary to define the regulatory functions associated with each biological compound. In the case of a Boolean logical model, these regulatory functions are Boolean functions.

To represent Boolean functions, in BCF, two predicates are defined: `functionOr(V, 1..N)` and `functionAnd(V, T, R)`. The predicate `functionOr(V, 1...N)` indicates that the regulatory function of node `V` has `N` terms. Then, `functionAnd(V, T, R)` indicates that the term `T` of the regulatory function of `V` contains the regulator (node) `R`. The sign associated with each regulator is omitted in the regulatory function predicates as that information is present in the corresponding edge.

The following Listing 4.1 represents the model illustrated in Figure 2.2, using the above defined ASP predicates.

```
1 vertex(v1).
2 vertex(v2).
3 vertex(v3).
4 vertex(v4).
5
6 edge(v1, v2, 0).
7 edge(v1, v3, 1).
8 edge(v2, v1, 1).
```

```
9  edge(v2, v3, 1).
10 edge(v4, v3, 0).
11 edge(v4, v2, 1).
12
13 functionOr(v1, 1..1).
14 functionAnd(v1, 1, v2).
15
16 functionOr(v2, 1..1).
17 functionAnd(v2, 1, v1).
18 functionAnd(v2, 1, v4).
19
20 functionOr(v3, 1..2).
21 functionAnd(v3, 1, v1).
22 functionAnd(v3, 2, v2).
23 functionAnd(v3, 2, v4).
```

Listing 4.1: Example of the representation of the model in Figure 2.2.

For example, consider that f_{v_3} is the regulatory function of biological compound represented by v_3, defined as $f_{v_3} = v_1 \vee (v_2 \wedge \neg v_4)$. Function f_{v_3} has two terms as indicated by line 20 of Listing 4.1. The first term of the function, defined in line 21, contains only regulator v_1, and the second term is a conjunction of v_2 and $\neg v_4$, as indicated in lines 22 and 23. Note that the negation of v_4 does not appear in the predicate `functionAnd`, as this sign is defined by the edge from v_4 to v_3 with sign 0, in line 10.

With the described ASP predicates, a Boolean logical model can be well defined. Now, in order to define an experimental observation, more predicates are introduced. To allow multiple experimental observations of the same regulatory network, we define the predicate `exp(E)`, identifying `E` as an experimental observation. Then, the observed values of the experiment must be defined. This, however, depend on the type of experimental observation. In this work we considered two types of observations:

- Stable state observations;
- Time-series observations.

The stable state observations, represent point attractors, as defined in Section 2.1.2, and therefore do not take into account the dynamic of the network. To encode these stable state observations, the predicate `obs_vlabel(E, V, S)` is used to represent that in experimental observation `E`, node `V` has an observed value of `S` $\in \{0, 1\}$.

For example, Listing 4.2 represents the stable state $S[v_1, v_2, v_3, v_4] = \{0000\}$ of the model in Figure 2.2. To represent multiple observations, a new experimental must be defined (e.g. `exp(e2).`) with the corresponding set of observed values (`obs_vlabel`).

```
exp(e1).
obs_vlabel(e1, v1, 0).
obs_vlabel(e1, v2, 0).
obs_vlabel(e1, v3, 0).
obs_vlabel(e1, v4, 0).
```

Listing 4.2: Stable state of the model in Figure 2.2.

When time-series observations are being considered, a time component must be taken into account. Time-series observations define a set of observed values in a sequence of time steps. In order to define such observations, the predicate `obs_vlabel(E, T, V, S)` is used, similarly to the previously described predicate. In this predicate, the time component `T` is added indicating that in experiment `E`, node `V`, has an observed value of `S` in time step `T`. The time value is defined as a non-negative integer.

For example, to define the sequence of observed states $S_0 = \{1100\}$, $S_1 = \{1010\}$, and $S_2 = \{0010\}$, for the model in Figure 2.2, the Listing 4.3 is used.

```
exp(e2).
obs_vlabel(e2, 0, v1, 1).
obs_vlabel(e2, 0, v2, 1).
obs_vlabel(e2, 0, v3, 0).
obs_vlabel(e2, 0, v4, 0).
obs_vlabel(e2, 1, v1, 1).
obs_vlabel(e2, 1, v2, 0).
obs_vlabel(e2, 1, v3, 1).
obs_vlabel(e2, 1, v4, 0).
obs_vlabel(e2, 2, v1, 0).
obs_vlabel(e2, 2, v2, 0).
obs_vlabel(e2, 2, v3, 1).
obs_vlabel(e2, 2, v4, 0).
```

Listing 4.3: Time-series observation of the model in Figure 2.2.

Remark 4.1.1. Note that all the instances of the variables in ASP predicates must start with a lowercase letter or number.

4.1.2 Consistency Check

Given a model and a set of experimental observations, we want to verify if the model is consistent, *i.e.*, if the model is able to reproduce the given observations. If the model is consistent, then there is no need to repair it. Otherwise, if the model cannot reproduce the observed values, it is necessary to identify possible reasons for inconsistency.

We say that a model is *inconsistent* if it has at least one inconsistent node, and *consistent* otherwise. A node is *inconsistent* if its regulatory function cannot reproduce the corresponding observed value. In the case of a stable state observation, each node must reproduce the corresponding observed value, considering the value of its regulators. In the case of a time-series observation, each node must reproduce, at each time step, the observed value, given the value of its regulators in the previous time step.

In order to consider different types of observations, different ASP programs are defined to check the consistency of the models, and identify inconsistent nodes in case of inconsistency. In this work, we are able to check the consistency of a logical model given either a set of stable state observations, or a set of time-series data, but not both simultaneously. Note that it is possible to consider multiple observations of the same type simultaneously.

Also, having complete observational data may not be possible. To address this, we consider that missing values may exist and can assume any value. We take advantage of the ASP paradigm properties, to consider all the possible combinations of missing values.

Stable State Observations

In order to check the consistency of a logical model, given a set of stable states, it is necessary to verify if each node is able to reproduce the corresponding observed value, given the value of its regulators.

The following Listing 4.4 defines the ASP program that verifies whether a model is consistent, and determines which nodes are inconsistent in case of inconsistency.

```
1 sign(0;1).
2 complement(T,S) :- sign(S), sign(T), T!=S.
3
4 vertex(V) :- edge(V,_,_).
5 vertex(V) :- edge(_,V,_).
6
7 % generate
8 1{vlabel(E,V,S):sign(S)}1 :- vertex(V), exp(E).
9
```

```
10 :-vlabel(E,V,S), obs_vlabel(E,V,T), complement(S,T).
11
12 % functions
13 % one positive or negative contribution in a term
14 onePositive(E,V,Id) :- functionAnd(V,Id, V2), edge(V2,V,S),
       vlabel(E,V2,S), exp(E).
15 oneNegative(E,V,Id) :- functionAnd(V,Id, V2), edge(V2,V,S),
       vlabel(E,V2,T), complement(S,T), exp(E).
16
17 % none negative in a term
18 noneNegative(E,V,Id) :- onePositive(E,V,Id), not oneNegative(E,V,Id).
19
20 vlabel(E,V,1) :- 1{noneNegative(E,V,Id):functionOr(V,Id)}, vertex(V),
       exp(E), not r_part(V).
21 vlabel(E,V,0) :- {noneNegative(E,V,Id):functionOr(V,Id)}0, vertex(V),
       exp(E), functionOr(V,_), not r_gen(V).
22
23 % inconsistencies
24 {r_gen(V)} :- vertex(V), not fixed(V).
25 {r_part(V)} :- vertex(V), not fixed(V).
26
27 % inconsistent nodes
28 repair(V) :- r_gen(V).
29 repair(V) :- r_part(V).
30
31 #minimize {1,V : repair(V)}.
32 #minimize {1,g,V : r_gen(V)}.
33 #minimize {1,p,V : r_part(V)}.
34
35 #show r_gen/1.
36 #show r_part/1.
37 #show vlabel/3.
38 #show repair/1.
```

Listing 4.4: Consistency check ASP program for stable state observations.

Lines starting with % indicate comment lines.

Lines 1 and 2 of Listing 4.4 are auxiliary predicates that define the possible values of the variables and that two values are complement of each other if they are different.

The rules defined in line 4 and 5 indicate that if there is an edge from, or to some compound `V`, then `V` is a node of the network. These lines are not necessary if all the nodes of the network are defined. However, as it is possible to infer the nodes from the edges, these rules ensure that if the nodes are not defined, then they are inferred.

As we support missing values, the predicate `vlabel` is defined to assign a value to each node. Line 8 indicates that every node `V` of every experiment `E` must have at least one and at most one (exactly one) assigned value `S` $\in \{0, 1\}$.

If a given observed value is defined, then the assign value of the corresponding node must be equal. In other words, there cannot exist an assigned value different of the corresponding observed value (if it exists), which is defined with the rule in line 10.

In order to check the consistency, the evaluation of the regulatory functions must be made. Given a Boolean function in DNF, its value is evaluated to 1 (true) if at least one of its terms evaluates to 1. Otherwise, the function evaluates to 0 (false). For a term to be evaluated to 1, all of its literals must be evaluated to 1, since a term is a conjunction (AND) of literals. Otherwise, the term is evaluated to 0. Line 14 and 15 define that a term has a positive or negative evaluated literal, respectively. Note that, in order to evaluate a literal it is taken into account its assigned value and the sign of the corresponding interaction. Then, on line 18 we define that a term does not contain any negative literal if it contains at least one positive literal and none negative, therefore that term evaluates to 1. Finally, lines 20 and 21, indicate that the assigned value of a node must be 1 if there is a term of the corresponding function that evaluates to 1. Otherwise, the assigned value must be 0.

To determine possible inconsistent nodes, *i.e.*, nodes that have functions that do not evaluate to the corresponding assigned value, two predicates are introduced in lines 24 and 25: `r_gen` and `r_part`. The former means that a node has an assigned value 1 but its function evaluates to 0, and the latter indicates that a node has an assigned value of 0 but its regulatory function evaluates to 1. This will allow to have inconsistent assignments while identifying those nodes at the same time. Lines 24 and 25 indicate that a node may or may not be an inconsistent node. Note that the rules on lines 20 and 21 take into account these predicates, allowing, for example, a function to evaluate to 1 and the assigned value not to be 1 if that node has the `r_part` predicate defined.

The `fixed` predicate, that appears on lines 24 and 25, prevents a given node to be considered inconsistent. Preventing a node to be considered inconsistent is of great interest, as it allows to avoid applying repair operations to such node in case of inconsistency. This, however, will make this consistency check program to justify a possible inconsistency with other nodes. With the

definition of fixed nodes, it may become impossible to justify or identify reasons of inconsistency, due to the fact that the problem may become overconstrained.

Lines 28 and 29 of Listing 4.4 define that an node with an inconsistent assigned value (`r_gen` or `r_part`) is an inconsistent node in need of repair.

Finally, lines 31 to 33 are minimisation ASP directives in order to minimise the number of inconsistent nodes. This will force the ASP program to assign the missing values such that the most number of assigned values are consistent with the model. Note that this only applies to unobserved values as line 10 prevents assignments that differ from the observed values.

The last four lines of Listing 4.4 are directives for the program to produce as output only the corresponding predicates. The output produced is the essential information in case of inconsistency: complete assignments to each node, the inconsistent nodes, and the type of inconsistency. Moreover, it is possible to retrieve all the possible solutions for this ASP program, allowing to consider all the possible assignments of values.

Time-Series Observations

To check the consistency of a model confronted with time-series observations, it is necessary to take into account the evolution of the network through time. Although the program is similar to the consistency check procedure for stable state observations, some adjustments are necessary.

Listing 4.5 defines the core ASP program to check the consistency of a model under dynamic observations.

```
1 time(0..t).
2 sign(0;1).
3 complement(T,S) :- sign(S), sign(T), T!=S.
4
5 vertex(V) :- edge(V,_,_).
6 vertex(V) :- edge(_,V,_).
7
8 %generate
9 1{vlabel(E,T,V,S):sign(S)}1 :- vertex(V), exp(E), time(T).
10
11 :-vlabel(E,T,V,S1), obs_vlabel(E,T,V,S2), complement(S1,S2).
12
13 % functions
14 % one positive or negative contribution in a term
15 onePositive(E,T,V,Id) :- functionAnd(V,Id, V2), edge(V2,V,S),
     vlabel(E,T,V2,S), exp(E), time(T).
```

```
16 oneNegative(E,T,V,Id) :- functionAnd(V,Id, V2), edge(V2,V,S1),
       vlabel(E,T,V2,S2), complement(S1,S2), exp(E), time(T).
17
18 % none negative in a term
19 noneNegative(E,T,V,Id) :- onePositive(E,T,V,Id), not
       oneNegative(E,T,V,Id).
20
21 %input nodes
22 input(V) :- not functionOr(V,_), vertex(V).
23 vlabel(E,T+1,V,S) :- input(V), vlabel(E,T,V,S), exp(E), time(T), T<t,
       not repair(V).
24
25 % inconsistencies
26 {r_gen(V)} :- vertex(V), not fixed(V).
27 {r_part(V)} :- vertex(V), not fixed(V).
28
29 % inconsistent nodes
30 repair(V) :- r_gen(V).
31 repair(V) :- r_part(V).
32
33 #minimize {1@1,V : repair(V)}.
34 #minimize {1@1,g,V : r_gen(V)}.
35 #minimize {1@1,p,V : r_part(V)}.
36
37 #show repair/1.
38 #show r_gen/1.
39 #show r_part/1.
40 #show vlabel/4.
```

Listing 4.5: Consistency check ASP program for time-series observations.

Most of the rules presented in Listing 4.5 are similar to the ones previously described in Listing 4.4, where only the notion of time was added.

The first line of Listing 4.5 indicates the number of time steps that will be considered, defined by a constant `t`. In lines 22 and 23, it is defined input nodes as nodes without regulatory function associated. This is important to ensure that an input node maintain its value through time. The rule defined in line 23 indicates that an input node as the same value as its value in the previous time step, if it is not being considered an inconsistent node.

However, in Listing 4.5 it is not defined how the values are assigned in each time step besides

input nodes. The assignment of values depends on the considered update scheme. In this work, we consider both synchronous and asynchronous update schemes. In a synchronous update scheme, all of the regulatory functions are applied at each time step, updating the values of the corresponding nodes. To do so, the rules in Listing 4.6 are defined.

```
1 vlabel(E,T+1,V,1) :- 1{noneNegative(E,T,V,Id):functionOr(V,Id)},
      vertex(V), exp(E), not r_part(V), not topologicalerror(V),
      time(T), T<t.
2 vlabel(E,T+1,V,0) :- {noneNegative(E,T,V,Id):functionOr(V,Id)}0,
      vertex(V), exp(E), functionOr(V,_), not r_gen(V), not
      topologicalerror(V), time(T), T<t.
3
4 % there cannot exist two times with the same update, which function's
      inputs are the same and the output is different, either in two
      different time points or in different experiments
5 % any function applied in two time points with the same inputs cannot
      have different outputs
6 topologicalerror(V) :- time(T1), time(T2), T1 != T2, T1 < t, T2 < t,
      vertex(V), {vlabel(P1,T1,V1,S1): vlabel(P2,T2,V1,S2), S1!=S2,
      functionAnd(V,Id, V1)}0, vlabel(P1,T1+1,V,S3),
      vlabel(P2,T2+1,V,S4), S3 != S4, not input(V).
7 topologicalerror(V) :- time(T), T < t, exp(P1), exp(P2), P1 != P2,
      vertex(V), {vlabel(P1,T,V1,S1): vlabel(P2,T,V1,S2), S1!=S2,
      functionAnd(V,Id, V1)}0, vlabel(P1,T+1,V,S3), vlabel(P2,T+1,V,S4),
      S3 != S4, not input(V).
8
9 repair(V) :- topologicalerror(V).
10
11 #minimize {1@2,top,V : topologicalerror(V)}.
12
13 #show topologicalerror/1.
```

Listing 4.6: Synchronous update scheme.

The first two lines of Listing 4.6 are similar to lines 20 and 21 of Listing 4.4 previously explained. When considering time-series observations, a special case of inconsistency may occur. In fact, we may have two occurrences of the same network state in different time steps, with transitions to different states. As the regulatory functions are deterministic, it is not possible to produce different values, given the same input values. This is a special type of inconsistency where the topology, or structures, of the network has an error, because there is no Boolean

function that can produce different values in the same conditions. The rules in line 6 and 7 of Listing 4.6 define that if an assignment is made where different values are produced in different time steps in the same conditions, we are in the presence of a node with a topological error. Moreover, as we support the existence of missing observation values, line 11 avoids the assignments where such topological error occurs, without preventing them as they can be, in fact, the reason for an inconsistent model.

In the asynchronous update scheme, only one regulatory function is applied at each time step. Therefore, different rules are necessary and are presented in Listing 4.7.

```
1 % only one node updates at each time
2 1{update(E,T,V):vertex(V)}1 :- exp(E), time(T), T<t.
3
4 vlabel(E,T+1,V,1) :- update(E,T,V),
      1{noneNegative(E,T,V,Id):functionOr(V,Id)}, vertex(V), exp(E), not
      r_part(V), not topologicalerror(V), time(T), T<t.
5 vlabel(E,T+1,V,0) :- update(E,T,V),
      {noneNegative(E,T,V,Id):functionOr(V,Id)}0, vertex(V), exp(E),
      functionOr(V,_), not r_gen(V), not topologicalerror(V), time(T),
      T<t.
6
7 % keep all others node values
8 vlabel(E,T+1,V,S) :- not update(E,T,V), vlabel(E,T,V,S), time(T), T<t.
9
10 % prevent updates that do not change the state of the network
11 :- update(E,T,V), vlabel(E,T,V,S), vlabel(E,T+1,V,S), time(T), T<t.
12
13 % there cannot exist two times with the same update, which function's
      inputs are the same and the output is different, either in two
      different time points or in different experiments
14 % any function applied in two time points with the same inputs cannot
      have different outputs
15 topologicalerror(V) :- time(T1), time(T2), T1 != T2, T1 < t, T2 < t,
      update(P1, T1, V), update(P2, T2, V), {vlabel(P1,T1,V1,S1) :
      vlabel(P2,T2,V1,S2), functionAnd(V,Id, V1), S1!=S2}0,
      vlabel(P1,T1+1,V,S3), vlabel(P2,T2+1,V,S4), S3 != S4, not input(V).
16 topologicalerror(V) :- time(T), T < t, update(P1, T, V), update(P2,
      T, V), P1 != P2, {vlabel(P1,T,V1,S1) : vlabel(P2,T,V1,S2), S1!=S2,
       functionAnd(V,Id, V1)}0, vlabel(P1,T+1,V,S3),
      vlabel(P2,T+1,V,S4), S3 != S4, not input(V).
```

```
17
18 repair(V) :- topologicalerror(V).
19
20 #minimize {1@2,top,V : topologicalerror(V)}.
21
22 #show update/3.
23 #show topologicalerror/1.
```

Listing 4.7: Asynchronous update scheme.

In order to consider that only one regulatory function is applied at each time step, the predicate `update` is defined in line 2 of Listing 4.7.

The rules in line 4 and 5 are similar to previously described rules (Listing 4.6) where the value of each node is assigned according to the corresponding regulatory function. However, the assignment is only done if the corresponding node is the node updated at that time step. Then, all the nodes that do not update at a given time step, will keep the corresponding value in the next time step, according to the rule defined in line 8.

We prevent updates that to not change the state of the network with the rule in line 11. Therefore, only updates that change the state of the network are considered in the dynamic behaviour. The rest of the rules in Listing 4.7 are similar to the ones in Listing 4.6 to identify nodes with a topological error.

These ASP programs check the consistency of a Boolean logical model when confronted with either stable state observations, or time-series observations. Moreover, considering time-series observation, both synchronous and asynchronous update schemes are supported. It is also supported the existence of missing values in observational data. In case of an inconsistent model, it is possible to retrieve information regarding the inconsistent nodes and reasons of inconsistency.

Consider the previous model example in Figure 2.2. If we consider an incomplete stable state observation where node $v_1 = 0$ and node $v_4 = 0$, and use the answer set program in Listing 4.2, we can get an answer set (or solution) with no inconsistent nodes, where nodes v_2 and v_3 are assigned with value 0. Note that $S[v_1, v_2, v_3, v_4] = \{0000\}$ is a stable state of this model (see corresponding STG of this model in Figure 2.6). Internally, the program considers all the possible assignments for nodes v_2 and v_3, and as there is an optimisation directive, the optimum solution with no inconsistent nodes is produced. However, if we now consider a stable state observation of node $v_1 = 1$ and node $v_4 = 0$, we obtain two optimum solutions, each with one inconsistent node. One of the solutions is to consider that $v_2 = 1$ and $v_3 = 1$ and we have

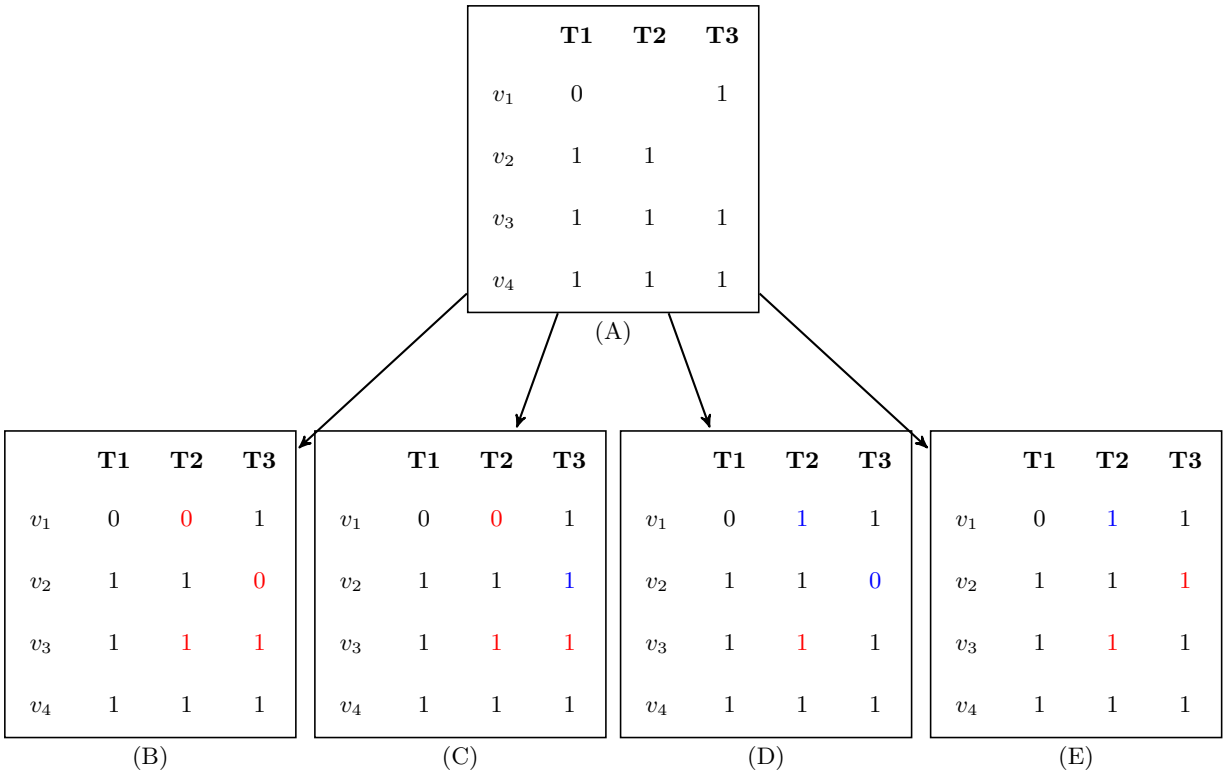


Figure 4.2: Consistency check of incomplete time-series observations under synchronous update scheme considering the model in Figure 2.2.

an inconsistent node v_2. In this solution, the program produces the predicates `r_gen(v2)` and `repair(v2)` indicating that node v_2 is not consistent where its regulatory function produces value 0, but value 1 is expected. If we apply every regulatory function, assuming the state $S[v_1, v_2, v_3, v_4] = \{1110\}$ we obtain the following:

$$f_{v_1} = v_2 = 1$$

$$f_{v_2} = \neg v_1 \wedge v_4 = 0$$

$$f_{v_3} = v_1 \vee (v_2 \wedge \neg v_4) = 1$$

Therefore, for this state to be a stable state, node v_2 is considered inconsistent. Another solution is to consider node $v_2 = 0$ and node $v_3 = 1$ where a similar reasoning can be made, and in this case node v_1 is considered inconsistent. Note that the other two possibilities ($S[v_1, v_2, v_3, v_4] = \{1000\}$ and $S[v_1, v_2, v_3, v_4] = \{1100\}$) for the compatible states with the given partial observation are not optimum solutions, with more than one inconsistent node, and are, therefore, not produced by the program.

Consider now the (incomplete) time-series observation in Figure 4.2 (A) under synchronous update scheme, with time steps T1, T2 and T3, and the model in Figure 2.2. Tables (B), (C), (D), and (E) represent the four compatible complete time-series observations, where all the

combinations of possible assignments of the missing values are considered. Marked in red are the values which cannot be explained by the model. Note that, since all the state information is fixed, each node can be checked independently to verify if its value can be reproduced at each time step considering the (fixed) state of the network in the previous time step. This means that at each time step, we consider that the input of each regulatory function is the expected value (the fixed values) rather than the currently produced values, as we assume that we may be in the presence of inconsistent nodes. For example, considering table (C), to check if the regulatory function of node v_2 can reproduce the expected value (1) at time step T3, we consider that node v_1 has value 0 at time step T2. Note that the regulatory function of v_1, at time step T2 produces value 1, which is not the expected value, and we assume that node v_1 is inconsistent and for future computation we consider the expected value of 0. The complete compatible time-series observation represented in table (B) contains three inconsistent nodes (v_1, v_2, and v_3), since each of these nodes contain at least one value that cannot be reproduced by the current model. Tables (C) and (E) have two inconsistent nodes, and table (D) has only one inconsistent node (v_3). Therefore, the consistency check procedure in Listings 4.5 and 4.6, produces as solution the complete time-series observation represented by table (D). This is the complete observation, compatible with the given time-series observation in table (A), with the minimum number of inconsistent nodes. Note that in case of multiple complete observations with the minimum number of inconsistent nodes, all the optimum solutions are produced by the consistency check procedure.

4.2 Boolean Function Relations

The study of Boolean functions and their properties became a very important task when considering the model revision process of Boolean logical models. One of the reasons for a model to be inconsistent is the incorrectness in regulatory functions. Therefore, it is desired to be able to change a regulatory function if necessary, without changing the topology (or structure) of the network. However, changing a regulatory function will also change the inherent dynamic behaviour of the model. In this work we limited the domain of the regulatory functions to monotone non-degenerate Boolean functions. Two Boolean functions can be compared using the relation $\preceq$ defined by Cury et al. [34] and presented in Section 2.2. Considering such relation and the set of all monotone non-degenerate Boolean functions, a partial order set can be defined and represented by an Hasse diagram (see Figure 2.8). Considering this partial order set of monotone non-degenerate Boolean functions, the proximity of two functions is proportional to the number of different entries in the corresponding truth tables. As such, the closer two functions are in the

Hasse diagram, the minimal impact exists in the regulatory structure, and therefore, minimal changes of the dynamic behaviour [34]. Therefore, in this work, we consider the rules to compute neighbours of a given monotone non-degenerate Boolean function presented by Cury et al. [34] as described in Section 2.2. In the following, an ASP encoding of these rules is presented.

Listing 4.8 is the base ASP program to help compute the parents and children of a given Boolean function. We consider that a Boolean function in BCF is represented as a set S where each element is a set of variables representing a term of the function. To compute a neighbour function, we first generate all the possible non-empty sets over n variables, where n is the function dimension (*i.e.* the number of regulators). Note that there are $p = 2^n - 1$ different sets. Lines 2 and 3 of Listing 4.8 indicate that there are n elements (function dimension) and p different sets, where n and p are given as constants of the program. We define the predicate `powerSet(N,E)` to represent the possible sets, where `N` $\in \{1, \ldots, p\}$ identifies the set, and variable (or element) `E` is present in that set. A Boolean function is then represented with the aid of predicate `term(T,E)`, meaning that variable `E` is present in term `T` of the function in BCF.

Line 5 of Listing 4.8 introduces predicate `selected(N)` to represent that the set `N`, defined by predicate `powerSet`, is a set representing a term of the function. To manipulate the p sets of possible terms (`powerSet`), some ASP rules are defined. In line 8, we define predicate `containsPS(N1,N2)` if there is no element in set `N2` that it is not present in set `N1`, *i.e.*, if set `N1` contains `N2`. Then in line 11 we define that a set `N1` is a `parent` of `N2` if `N1` contains `N2`, and `N1` contains only one element that it is not present in `N2`. Line 13 define predicate `countPS(N,S)` meaning that set `N` contains `S` elements.

As we are only considering non-degenerate Boolean functions, the rule on line 15 prevents the definition of functions that do not contain all the n variables in the BCF representation.

```
1 %number of elements
2 element(1..n).
3 id(1..p).
4
5 selected(N1) :- id(N1), id(N2), term(N2,_), {element(E) :
    powerSet(N1,E), not term(N2,E)}0, {element(E) : term(N2,E), not
    powerSet(N1,E)}0.
6
7 % N1 contains N2
8 containsPS(N1,N2) :- id(N1), id(N2), {element(E): powerSet(N2,E), not
    powerSet(N1,E)}0.
9
10 % N1 is parent of N2 (contains N2) with only one more element
```

```
11 parent(N1,N2) :- containsPS(N1,N2), id(N1), id(N2), C = #count
       {element(E): powerSet(N1,E), not powerSet(N2,E)}, C=1.
12
13 countPS(N,S) :- id(N), S = #count {element(E):powerSet(N,E)}.
14
15 :- element(E), {powerSet(N,E):selected(N)}0.
16
17 1{rule(r1;r2;r3)}1.
18
19 #show rule/1.
```

Listing 4.8: ASP base program to compute parents or children of Boolean functions.

There are three rules to compute the parents of a given function, and three rules to compute the children of a given function (see Section 2.2). We consider that we either only compute the parents, or we only compute de children. Moreover, we consider that exactly one rule can be applied at a time, as defined in line 17 of Listing 4.8, producing only one parent (or child). Then, to compute all the parents (resp. children) of a given function, an ASP solver can be used to retrieve all possible solutions.

To compute the parents, following the rules 1, 2, and 3 defined by Cury et al. [34], we defined the ASP programs in Listing 4.9, 4.10, and 4.11, respectively. To compute a parent of a given Boolean function, Listings 4.8, 4.9, 4.10, and 4.11 must be used simultaneously. Listing 4.9 define the rule 1 to compute a parent of a function. This rule adds to the set of terms of a function, a new term that is independent of all the terms in the function (is not contained nor contains), and is of maximum size. Lines 1, 2, and 3 of Listing 4.9 define as `dominate` the sets that are contained or contain a term present in the function. Then in line 5 it is defined that a set is `independent` if it is not `dominate`. The maximum independent set is identified in line 8. The program produces the predicate `selectedR1(N)` that indicates that the term `N` is present in a parent of the original function when rule 1 is applied. As this rule only adds one term, all the original terms are present in the parent function, as indicated by the rule in line 11. Then, in line 12, exactly one new term is selected to be present in the parent function. To ensure that all variables are present in the parent function, the restriction in line 14 is defined.

```
1 dominate(N) :- selected(N).
2 dominate(N1) :- id(N1), id(N2), selected(N2), containsPS(N1,N2).
3 dominate(N1) :- id(N1), id(N2), selected(N2), containsPS(N2,N1).
4
5 independent(N) :- id(N), not dominate(N).
```

```
6 :- id(N), independent(N), dominate(N).
7
8 maxIndependent(N) :- id(N), independent(N), countPS(N,S),
      {countPS(N2,S2): independent(N2), S < S2}0.
9
10 :- rule(r1), not maxIndependent(_).
11 selectedR1(N) :- selected(N), id(N), rule(r1),
      1{maxIndependent(N2):id(N2)}.
12 1{selectedR1(N):maxIndependent(N)}1 :- rule(r1).
13
14 :- element(E), rule(r1), {powerSet(N,E):selectedR1(N)}0.
15
16 :- rule(r1), not selectedR1(_).
17 :- selectedR1(_), not rule(r1).
18
19 #show selectedR1/1.
```

Listing 4.9: ASP program to compute the parents of a Boolean function - Rule 1.

The rule 2 to compute a parent function is defined in Listing 4.10. In line 1 we define a potential set candidate to be added to the function (`potentialR2`). According to rule 2, this potential candidate must be contained by at least one term of the original function and must not exist another set that is also contained by that term and contains the potential set. The potential candidate must not be contained by any set that satisfies rule 1. This potential candidate, if exists, will replace the terms of the original function that contains the candidate. The predicate `potentialR2(N1,N2)` in line 3 is defined to identify the sets `N2` that will be replaced by the candidate `N1`. Then lines 5 and 7 select the terms that are present in the parent function by rule 2 (`selectedR2`). To ensure that all variables are present in the parent function, the restriction in line 9 is defined.

```
1 potentialR2(N1) :- id(N1), id(N2), selected(N2), parent(N2,N1),
      {containsPS(N3,N1): independent(N3)}0.
2
3 potentialR2(N1,N2) :- id(N1), id(N2), potentialR2(N1), parent(N2,N1),
      {potentialR2(N3):parent(N3,N1)}0.
4
5 {selectedR2(N):potentialR2(N,_)}1.
6
7 selectedR2(N) :- id(N), selected(N),
```

```
    1{selectedR2(N3):potentialR2(N3,_)},
    {selectedR2(N2):parent(N,N2),id(N2)}0.

:- element(E), rule(r2), {powerSet(N,E):selectedR2(N)}0.

:- rule(r2), not selectedR2(_).

:- selectedR2(_), not rule(r2).

#show selectedR2/1.
```

Listing 4.10: ASP program to compute the parents of a Boolean function - Rule 2.

The rule 3 to compute a parent function is defined in Listing 4.11. This rule indicates that we add two terms to the original function that satisfy rule 2 except restriction (c) (see Section 2.2). The program identifies a potential term candidate that will be removed from the original function (`potentialRemR3`) in line 1 of Listing 4.11. Then, in line 3 are selected two terms that satisfy rule 2. In line 7 and 8 are defined ASP rules to satisfy the exception mentioned above. In these rules we say that each new term that will be added to the function must contain at least one element that is also present in a different term of the parent function. To ensure that all variables are present in the parent function, the restriction in line 12 is defined.

```
1 1{potentialRemR3(N):selected(N)}1 :- rule(r3).

2
3 2{selectedR3(N):potentialR2(N,N2),potentialRemR3(N2)} :- rule(r3).

4
5 selectedR3(N) :- selected(N), not potentialRemR3(N), rule(r3).

6
7 duplicateElementR3(E,N1) :- powerSet(N1,E), selectedR3(N1),
    potentialR2(N1,N2), potentialRemR3(N2), 1{selectedR3(N3):id(N3),
    N3!=N1, powerSet(N3,E)}.
8 :- selectedR3(N1), potentialR2(N1,N2), potentialRemR3(N2),
    {element(E):powerSet(N1,E), not duplicateElementR3(E,N1)}0.

9
10 :- selectedR3(N), id(N), selected(N),
    1{selectedR3(N2):parent(N,N2),id(N2)}.

11
12 :- element(E), selectedR3(N1), id(N1), {powerSet(N,E):selectedR3(N)}0.

13
14 :- rule(r3), not selectedR3(_).
```

```
15
16 :- selectedR3(_), not rule(r3).
17
18 #show selectedR3/1.
```

Listing 4.11: ASP program to compute the parents of a Boolean function - Rule 3.

Similarly, following the rules 1, 2, and 3 to compute the children of a given Boolean function defined by Cury et al. [34], we define the ASP programs in Listing 4.12, 4.13, and 4.14 respectively. To compute a child of a given Boolean functions, Listings 4.8, 4.12, 4.13, and 4.14 must be used simultaneously. Rule 1 says that we remove a term of the function such that there is no independent set of the resulting function that contains the removed term. As this rule is related to rule 1 of computing parents, Listing 4.12 has similar definitions of Listing 4.9 described above. In line 1 of Listing 4.12 we define one potential term to be removed. Line 6 define the selected terms for the child function with the predicate `selectedR1`. Then in lines 12 and 13 we define that cannot exist an independent term that contains the removed term.

```
1 1{potentialRemR1(N):selected(N)}1 :- rule(r1).
2 :- rule(r1), not potentialRemR1(_).
3
4 potentialRem(N) :- potentialRemR1(N).
5
6 selectedR1(N) :- selected(N), not potentialRemR1(N), rule(r1).
7
8 dominate(N1,N) :- selected(N), potentialRem(N1), N!=N1.
9 dominate(N,N1) :- id(N1), id(N2), selected(N2), containsPS(N1,N2),
      potentialRem(N), N2!=N, N1!=N.
10 dominate(N,N1) :- id(N1), id(N2), selected(N2), containsPS(N2,N1),
      potentialRem(N), N2!=N, N1!=N.
11
12 independent(N1,N) :- id(N), not dominate(N1,N), potentialRem(N1),
      N!=N1.
13 :- id(N), independent(N1,N), dominate(N1,N).
14
15 :- independent(N,_), not potentialRem(N).
16 :- dominate(N,_), not potentialRem(N).
17
18 :- potentialRem(N),
      {potentialRemR1(N);potentialRemR2(N);potentialRemR3(N)}0.
19
```

```
20 :- potentialRemR1(N), id(N2), independent(N,N2), containsPS(N2,N),
     rule(r1).
21
22 :- rule(r1), element(E), potentialRemR1(N2),
     {powerSet(N,E):selectedR1(N)}0.
23
24 :- rule(r1), not selectedR1(_).
25 :- selectedR1(_), not rule(r1).
26
27 #show selectedR1/1.
```

Listing 4.12: ASP program to compute the children of a Boolean function - Rule 1.

The rule 2 to compute a child function is defined in Listing 4.13. In this rule, we first remove a term of the function. Then, we add a new term that is a minimum size independent set that contains the removed term. In line 1 of Listing 4.13, we define a potential term to be removed (`potentialRemR2`). In line 6 we say that each term that is not removed is present in the child function (`selectedR2`). Lines 8 define a potential new term to be added that is independent and contains the removed term. Then, in line 10, we add the new term that does not contain any other potential set. To ensure that all variables are present in the child function, the restriction in line 12 is defined.

```
1 1{potentialRemR2(N):selected(N)}1 :- rule(r2).
2 :- rule(r2), not potentialRemR2(_).
3
4 potentialRem(N) :- potentialRemR2(N).
5
6 selectedR2(N) :- selected(N), not potentialRemR2(N), rule(r2).
7
8 potentialR2(N) :- independent(N2,N), potentialRemR2(N2),
     containsPS(N,N2), N!=N2.
9
10 selectedR2(N) :- potentialR2(N), rule(r2),
     {potentialR2(N2):containsPS(N,N2), N!=N2}0.
11
12 :- potentialRemR2(N2), element(E), rule(r2),
     {powerSet(N,E):selectedR2(N)}0.
13
14 :- rule(r2), {potentialR2(N):selectedR2(N)}0.
15 :- potentialR2(N), potentialRemR2(N2), not independent(N2,N), N2!=N.
```

```
16
17 :- rule(r2), not selectedR2(_).
18 :- selectedR2(_), not rule(r2).
19
20 #show selectedR2/1.
```

Listing 4.13: ASP program to compute the children of a Boolean function - Rule 2.

The rule 3 to compute a child function is defined in Listing 4.14. This rule is the inverse of rule 3 to compute a parent function. In this rule two terms of the function are removed and a new term, that is the union of the removed terms, is added. The removed terms must not satisfy rules 1 and 2. In line 1 of Listing 4.14 two terms are chosen to be removed (`potentialRemR3`). The terms that are not removed are present in the child function (line 6). Lines 9-12 make sure that the removed terms do not satisfy rule 1. Lines 16 and 17 do the same for rule 2. Then, lines 20-22 select the new term to be added that is the union of the removed terms. To ensure that all variables are present in the child function, the restriction in line 24 is defined.

```
1 2{potentialRemR3(N):selected(N)}2 :- rule(r3).
2 :- rule(r3), not potentialRemR3(_).
3
4 potentialRem(N) :- potentialRemR3(N).
5
6 selectedR3(N) :- selected(N), not potentialRemR3(N), rule(r3).
7
8 %not r1
9 invalidR1(N) :- potentialRemR3(N), id(N2), independent(N,N2),
      containsPS(N2,N), rule(r3).
10 validR1(N) :- not invalidR1(N), potentialRemR3(N), rule(r3).
11 invalidR1(N1) :- rule(r3), potentialRemR3(N1), element(E),
      {powerSet(N,E):selected(N), N!=N1}0.
12 :- rule(r3), potentialRemR3(N), validR1(N).
13
14
15 %not r2
16 potentialR32(N) :- independent(N2,N), potentialRemR3(N2),
      containsPS(N,N2), N!=N2.
17 :- rule(r3), 1{potentialR32(N):id(N)}.
18
19 %union of terms
20 potentialR3(N) :- id(N), potentialRemR3(N1), potentialRemR3(N2),
```

```
    N1!=N2, containsPS(N,N1), containsPS(N,N2), N!=N1, N!=N2.
21 duplicateElementR3(E,N) :- element(E), potentialR3(N),
    potentialRemR3(N1), N1!=N, powerSet(N,E), powerSet(N1,E).
22 selectedR3(N) :- potentialR3(N), {element(E):powerSet(N,E), not
    duplicateElementR3(E,N)}0.
23
24 :- rule(r3), element(E), potentialRemR3(N2),
    {powerSet(N,E):selectedR3(N)}0.
25
26 :- rule(r3), not selectedR3(_).
27 :- selectedR3(_), not rule(r3).
28
29 #show selectedR3/1.
```

Listing 4.14: ASP program to compute the children of a Boolean function - Rule 3.

As it was mentioned before, it is possible to compute all the parents (or children) of a given function by using an ASP solver to compute all the possible solutions. Although it is possible to compute the parents and children of a monotone non-degenerate Boolean function using these ASP programs, it is not the most efficient approach. With the presented definition of the problem, and given the characteristics of ASP, all the possible combinations of functions are considered by the ASP solver when computing parents/children of functions. The number of Boolean functions with dimension n is 2^{2^n}. Although we restrict the domain to monotone non-degenerate boolean functions, the number of functions is still very large [34]. In fact, the number of monotone Boolean functions is known as the Dedekind number [96, 97] and there is no closed-form expression for it. It is only known the exact number of monotone Boolean functions up to $n = 8$, as shown in Table 4.1.

To improve performance and avoid considering all the possible functions when computing

Table 4.1: Number of monotone ($M(p)$) and monotone non-degenerate ($N(p)$) Boolean functions of dimension p.

p	$M(p)$	$N(p)$
1	3	1
2	6	2
3	20	9
4	168	114
5	7 581	6 894
6	7 828 354	7 785 062
7	2 414 682 040 998	2 414 627 396 434
8	56 130 437 228 687 557 907 788	56 130 437 209 370 320 359 968

parents and children of a given Boolean function, a C++ procedure was developed, following the same rules described in Section 2.2. This procedure revealed to be faster to compute parents and children of a given Boolean function than the ASP approach, as shown in the results section (Section 6.2). The implementation of these rules as a C++ library is available online[1].

[1] `https://github.com/FilipeGouveia/BooleanFunction`

Chapter 5

Model Revision

Models of biological regulatory networks are not always in line with existing experimental observations, that is, the model may not be able to reproduce some experimental observations. In this case we say that the model is inconsistent and must be revised.

In this Chapter a Model Revision approach for Boolean logical models of biological regulatory networks is presented. Section 5.1 details the model revision approach, and in Section 5.2 is presented the developed ModRev tool.

5.1 Model Revision Approach

As new data is acquired, models may become inconsistent, not being able to reproduce the new data, and therefore a model revision process is needed. In this work we will consider Boolean logical models, as defined in Section 2.1.1.

Figure 5.1 illustrates the general idea of the model revision process. Given a Boolean logical model and a set of experimental observations, in a first phase it is necessary to check the consistency of the model. This consistency check procedure is described in Section 4.1. If a model is consistent with the experimental observations, there is no need for repair. However, if the model is inconsistent, it is necessary to search for possible repair operations in order to render the model consistent. The goal is to be able to produce sets of repair operations that can be applied to an inconsistent model. There may exist a lot of possible combinations of repair operations that can render an inconsistent model consistent, and therefore, it is desired to filter possible solutions. To that end, and to avoid exploring all the possible combinations of repair operations, we defined an optimisation criterion to select optimal solutions as a product of the model revision approach.

We defined four possible causes for inconsistency and corresponding repair operations, as

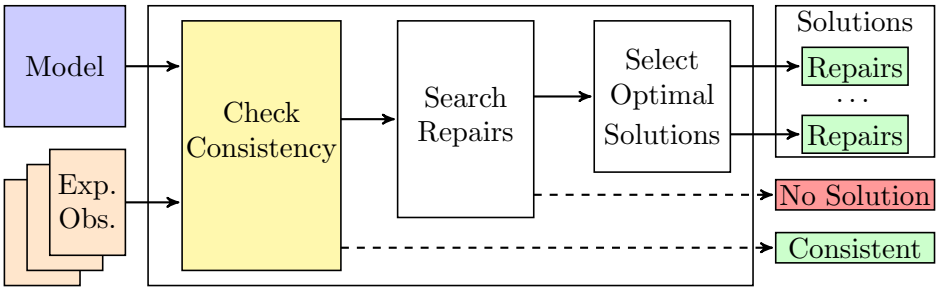


Figure 5.1: Overview of the model revision process.

Table 5.1: Causes of inconsistency and corresponding repair operations. Class F stands for Function repair and class T stands for Topology repair.

Cause	Repair Operation	Class
Wrong Regulatory Function	Function change	F
Wrong Interaction Type	Edge sign flip	T
Wrong Regulator	Edge removal	T
Missing Regulator	Edge addition	T

shown in Table 5.1. A model may have a wrong regulatory function defined, and in this case, we want to be able to change the regulatory function. An interaction type (activation or inhibition) may be incorrect, and in this case, we may want to change the interaction type by changing the sign of the corresponding edge in the regulatory graph (see Section 2.1.1). A model may also be inconsistent due to the presence of a wrong regulator or due to the absence of a missing regulator. In these cases, we may want to consider remove the wrong regulator by removing its edge, or add the missing regulator by adding the corresponding edge. These repair operations can be classified as function repairs (class F) thus changing a given regulatory function, or as topological repairs (class T) thus changing the topology (or structure) of the regulatory graph.

As previously mentioned, considering the four repair operations presented in Table 5.1, different possible combinations of such operations can render an inconsistent model consistent. Consider the model in Figure 2.2 with four nodes as an example, and assume that node v_3 is inconsistent. To repair the model, we can apply any combination of operations over node v_3, *i.e.* changing its regulatory function, flipping the sign of any of its incoming edges (regulators), remove any of its regulators and/or add the missing regulator (node v_3 itself). Taking also into account that there are 114 monotone non-degenerate Boolean functions with four arguments, 9 with 3 arguments, 2 with two arguments, and 1 with one argument, and considering that we can change the regulatory function to any function within the corresponding dimension (taking into account possible addition or removal of regulators), there are a total of 2 168 different combinations of repair operations that can be applied over node v_3, as shown in Table 5.2. Each line represents a combination of the number of operations of each type that can be applied.

Table 5.2: Combinations of all possible repair operations that can be applied over nove v_3 of the model in Figure 2.2. For each operation type (remove regulator; add regulator; flip the sign of edge; and change function) is indicated the number of operations (#Oper.) and the corresponding number of possible combinations/options (#Opt.), and the corresponding regulatory function dimension (Dim.) and number of functions (#F).

Removal		**Addition**		**Flip Sign**		**Function**		**Total**
#Oper.	**#Opt.**	**#Oper.**	**#Opt.**	**#Oper.**	**#Opt.**	**Dim.**	**#F**	
0	1	0	1	0	1	3	8	8
0	1	0	1	1	3	3	9	27
0	1	0	1	2	3	3	9	27
0	1	0	1	3	1	3	9	9
0	1	1	2	0	1	4	114	228
0	1	1	2	1	3	4	114	684
0	1	1	2	2	3	4	114	684
0	1	1	2	3	1	4	114	228
1	3	0	1	0	1	2	2	6
1	3	0	1	1	2	2	2	12
1	3	0	1	2	1	2	2	6
1	3	1	2	0	1	3	9	54
1	3	1	2	1	2	3	9	108
1	3	1	2	2	1	3	9	54
2	3	0	1	0	1	1	1	3
2	3	0	1	1	1	1	1	3
2	3	1	2	0	1	2	2	12
2	3	1	2	1	1	2	2	12
3	1	0	1	0	1	0	1	1
3	1	1	2	0	1	1	1	2
							Total	2 168

Note that whenever a regulator is added or removed, the regulatory function will necessarily change, and the change of function dimension have to be taken into account. Moreover, when we consider adding a missing regulator, it can be added with a positive or negative interaction, thus adding two possibilities. In general, if we have a node with n regulators and m missing regulators, considering the removal of r regulators we have $\binom{n}{r}$ possibilities, considering the addition of a regulators that can be added with either positive or negative sign we have $2^a \times \binom{m}{a}$ possibilities, considering flipping the sign of e edges we have $\binom{n-r}{e}$ possibilities, and we can change the regulatory function to any function with dimension $n - r + a$.

Considering that a model can have more than one node in need of repair, that models typically have many dozens of nodes, and that the number of possible monotone Boolean functions increases exponentially with the number of regulators, there is a combinatorial explosion in the search space for possible repair operations.

To filter all the possible solutions of repair operations that can be applied to render an inconsistent model consistent, and to avoid exploring all the possible combinations of such

operations, we defined the following lexicographic optimisation criterion:

1. Minimise the number of add/remove edge operations;
2. Minimise the number of flip sign of an edge operations;
3. Minimise the number of function change operations.

The defined optimisation criterion takes into consideration how a logical model is built. In particular, multiple Boolean functions are likely to be consistent with the existing data, but only one is chosen as the regulatory function of a node. Therefore, we assume that we have a higher level of confidence in the correctness of the topology of the network than in the regulatory functions of the model. Note that in the defined criterion, the top priority is to minimise the number of add/remove edge operations which changes the topology of the network. Whereas in the lowest priority is the number of function change operations. This means that it is preferred solutions with only function changes, over solutions with some topological operations.

Given the consistency check procedure described in Section 4.1, in case of an inconsistent model it is provided not only all the minimum set of inconsistent nodes, but also the corresponding complete data (stable state or time-series) with all the expected values (see Figure 4.2). With this in mind, we can try to repair each node individually without having to take into account the interactions with other nodes in the network dynamics, as we only have to be sure that each node can produce the expected value (at each time step in case of time-series data). Note that, by fixing the complete experimental data first, it is not necessary to generate the model's dynamic when a repair operation is applied, but rather just make sure that the model can reproduce the fixed set of complete experimental observations, as all state transitions are defined by the complete observations. Moreover, this can be done by considering each inconsistent node independently.

Given a model and the set of complete experimental data with the corresponding inconsistent nodes given by the consistency check procedure described in Section 4.1, we defined the model revision procedure in Algorithm 1 and 2. This is an iterative and incremental approach. The procedure iterates over all the complete data and corresponding inconsistent nodes, and then iterates over every inconsistent node to find possible repair operations. To repair an inconsistent node, the procedure uses an incremental approach where it is considered an incremental number of repair operations, following the above defined optimisation criterion. The search for repair operations starts by considering only function changes, as it is desired to avoid changing the topology of the network. If no consistent function is found, topological repairs are considered. We try one operation of edge sign flip, and then, if one operation is not enough to render the

Algorithm 1: Model Revision

Input: Model, ExpData [ExpData1, ExpData 2, ...]
Output: Set of optimal solutions found

```
1 begin
    // Check the consistency of the model given the set of experimental
       observation with the ASP program defined in Section 4.1
2   [(ExpData1,InconsistentNodes1), ...] ← ConsistencyCheck(Model, ExpData);
3   foreach pair of experimental data and corresponding inconsistent nodes do
4     foreach Inconsistent node n do
5       RepairNode(n, Model, ExpData);                    // Algorithm 2
6     end
7   end
8   Filter non-optimal solutions
9 end
```

model consistent, we increment the number of allowed operations and consider all combinations of two operations. We consider combinations of increasing number of edges until all the edge combinations are considered. For each attempt at flipping the sign of an edge, all possible functions changes are considered again. If no solution is found, the procedure advances to the next stage of topological changes, by repeating this process considering the addition or removal of one edge. If the model is still not consistent, then we consider the same process with two edges. All the combinations are considered increasingly until no more edges can be added or removed. Having this incremental approach allows to find optimal solutions, avoiding exploring all the possible combinations of repair operations. Once a solution to repair an inconsistent node is found, the procedure considers it as an upper bound. All the optimal solutions will have the same number of operations (considering the lexicographic criterion) or less, but never more.

In the end of the model revision procedure, we have optimal solutions for each set of complete data, that may differ in optimality. Hence, in the end only the optimal solutions are produced.

Figure 5.2 illustrates an example of a model revision process for a Boolean logical model confronted with an incomplete time-series observation under synchronous update scheme. The consistency check procedure (see Section 4.1) considers all the possible combinations of compatible time-series data, *i.e.* all the combinations of the missing values, as shown in the four tables illustrated. The values marked in red in each table represent the values that cannot be explained by the given model. The consistency check procedure produces all the compatible complete data with the minimum amount of inconsistent nodes. Note that in this case there is only one table (third table) with the minimum number of inconsistent nodes (one node). However, if there were more tables with the same (minimum) number of inconsistent nodes, all would be produced and considered for the search of repair operations. Moreover, the consistency

Algorithm 2: Repair Node

```
   Input: Node, Model, ExpData
 1 begin
 2     for n = 0 to MaxAddRemoveOperations do
 3         foreach combination of n add and remove ops. (to Node) do
 4             for m = 0 to NumberOfEdges do
 5                 foreach combination of m flipping edge sign ops. (of Node) do
 6                     Check the consistency of Node if is not consistent then
 7                         Search for consistent function (function change op.)
 8                     end
 9                     if solution is found then
10                         Finish the current iteration to search for other solutions with the
                            same number of operations and exit the cycle
11                     end
12                 end
13             end
14             if solution is found then
15                 Finish the current iteration to search for other solutions with the same
                    number of operations and exit the cycle
16             end
17         end
18     end
19 end
```

check procedure produces the inconsistent nodes and corresponding reason(s) of inconsistency. Then, for each complete time-series data produced by the consistency check procedure, for each inconsistent node, Algorithm 2 is called. In the end, the model revision procedure produces all the optimal set of repair operations that can render the model consistent. In the example of Figure 5.2, there is only one optimal solution, with one repair operation, which is to change the regulatory function of v_3.

5.1.1 Validation of Node Consistency

For the model revision procedure to be able to repair an inconsistent node, it is necessary to verify if the node became consistent after each repair operation being considered. This validation of consistency depends on the type of experimental observations and update scheme considered. In this work it is considered stable state observations and time-series data, under synchronous and asynchronous update schemes.

Considering a set of complete stable state observations, for a node to be consistent it must be able to reproduce its value considering the value of its regulators, for each stable state observation. Note that the consistency check procedure gives a complete stable state data in case of missing observed values.

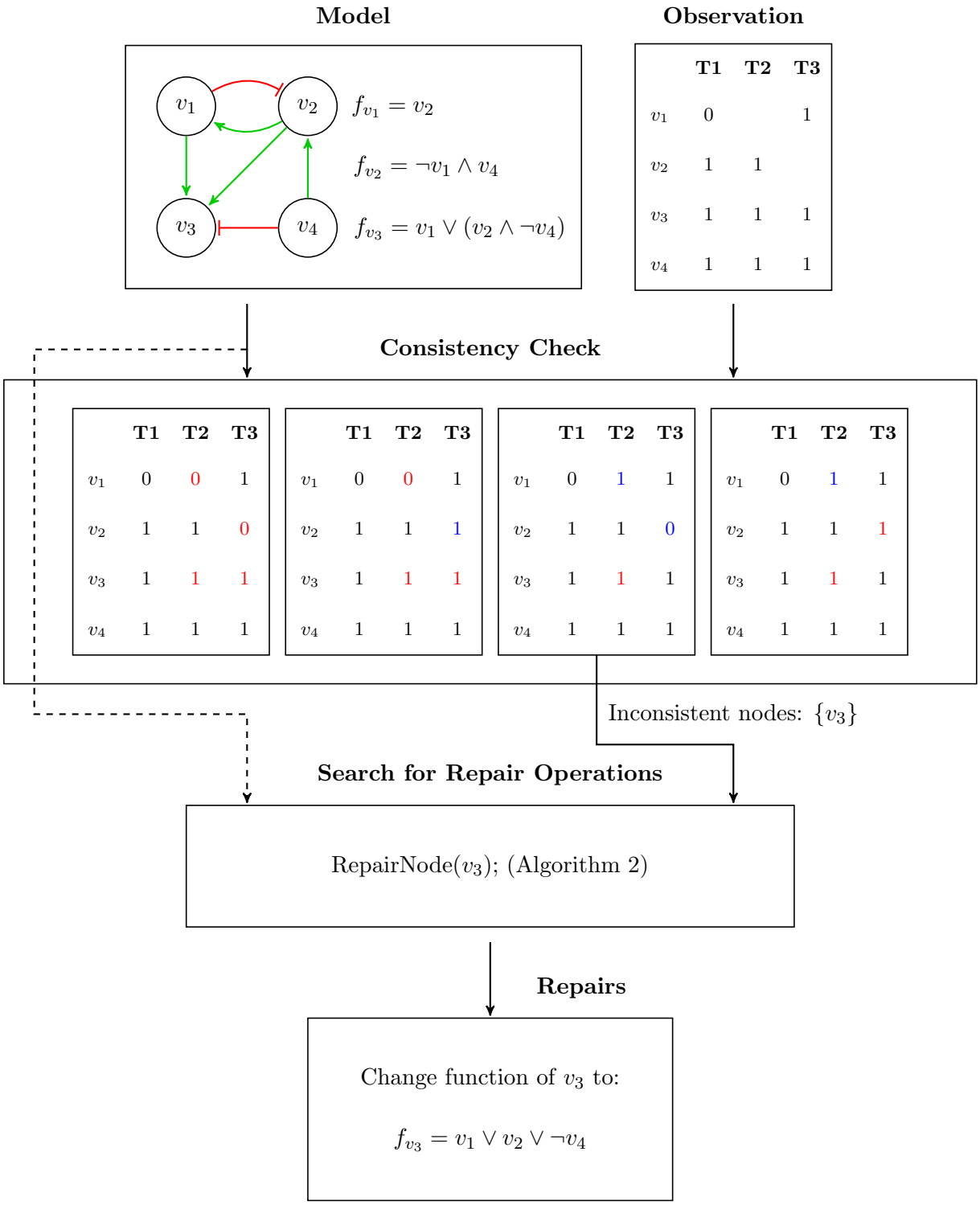


Figure 5.2: Model revision process of a Boolean logical model confronted with a time-series observation under synchronous update scheme.

To validate the consistency of a node given a set of time-series observations, the process is similar to the one considering stable state observations. The difference relies on the necessity of verifying if a node is able to reproduce the expected value at every time step. However, we have to take into account the update scheme considered. Considering the synchronous update scheme, a node is consistent only if it can reproduce the expected value at every time-step given the expected values of its regulators in the previous time step. In an asynchronous update scheme, only one node is updated at each time step. To verify if a node is repaired, it is necessary to verify if it can reproduce its expected value given the expected values of its regulators, not at every time step, but only at the time steps in which that node is updated. Note that the consistency check procedure defined in Section 4.1 not only produces the complete observations (all the expected values), but also the information regarding the updated nodes.

5.1.2 Function Repair Search

The search for possible function repairs is a crucial step in our model revision approach. Changing a regulatory function will change the model dynamic behaviour. When repairing an inconsistent model, it is desired to make the least amount of changes possible, with minimal impact on the model dynamic behaviour. When considering monotone non-degenerate Boolean functions, the impact of a function change can be minimised considering an Hasse diagram (see Section 2.2 and Figure 2.8 for details). The closer two functions are in an Hasse diagram, the less impact exists in the dynamic behaviour when changing a regulatory function [34].

In order to search for possible function repairs, information regarding the type of inconsistency is used to avoid exploring the whole function space. The consistency check procedure defined in Section 4.1 gives some information that can be useful in the search for function repairs. The consistency check procedure indicates if a node needs a more generic function (`r_gen`), *i.e.* if a node is expected to have value 1 but the function evaluates to 0, or if it needs a more specific function (`r_part`), *i.e.* if a node is expected to have value 0 but the function evaluates to 1.

Considering the relation between monotone non-degenerate Boolean functions defined in Section 2.2, the ancestors (parents) of a given function are more generic functions, having more entries with value 1 in the corresponding truth table. Analogously, the descendants (children) of a given function are more specific functions, having more entries with value 0 in the corresponding truth table. Note that the top function in the Hasse diagram is the most generic function in which all the entries of the truth table have value 1 except one entry, and the bottom function in the Hasse diagram is the most specific function in which all the entries of the truth table have value 0 except for one entry.

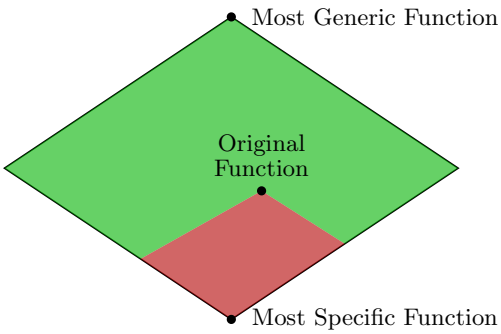


Figure 5.3: Search space of Boolean functions assuming the original function has an entry that evaluates to 0 but a 1 is needed. The diamond shape represents the space of monotone non-degenerate Boolean functions. The area in red represents the space of impossible function candidates. The area in green represents the space of possible function candidates.

For the sake of clarity, here we will explain the search methods for function repairs considered when we have a node to be expected value 1 but the function evaluates to 0 (`r_gen`), but the process is analogous for the opposite case (`r_part`). If the regulatory node evaluates to 0 but a 1 is expected, we know that the descendants of that function will not be possible function candidates since they all have the same entries in the truth table with value 0 (and more). Figure 5.3 illustrates the search space of monotone non-degenerate Boolean functions, with the area in red being the descendants of the original function that are not valid function candidates for replacement when considering the `r_gen` type of inconsistency. The area in green represents the search space of possible function candidates.

To search for possible functions repairs, we can consider if the original function is in the bottom half or in the top half of the Hasse diagram and start the search for possible function repairs from the most specific function (resp. the most generic function), and then go up (resp. down) the Hasse diagram. In order to consider the functions closer to the original function for the repair operation, the search is performed in a breath-first style ensuring that all the closest functions are considered, taking into account the comparison between levels of functions as defined in Section 2.2. Moreover, with this method, the search can stop earlier once a consistent function candidate is found: with the same level as the original function; with the highest level lower than the original function (and there is no candidate with the same level of the original); or with the lowest level higher than the original function. Figure 5.4 illustrates the search for a function repair in this case. In Figure 5.4 the original function is in the bottom half of the Hasse diagram and, therefore, the search starts at the most specific function and goes up in a breath-first search. The search can stop at the dashed line, as consistent functions are found, and the search is going farther from the original function. Other consistent functions may exist, above the dashed line, but are not optimal and therefore are not considered.

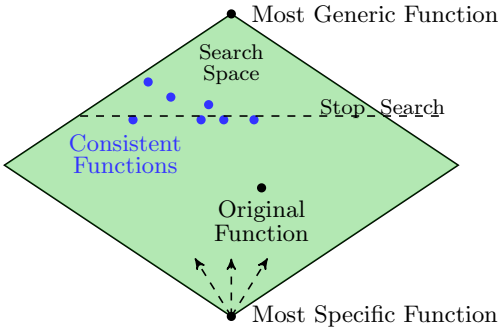


Figure 5.4: Search of Boolean functions assuming the original function has an entry that evaluates to 0 but a 1 is needed. The diamond shape represents the space of monotone non-degenerate Boolean functions. Direction of the search is indicated by dashed arrows. Search stops at the dashed line.

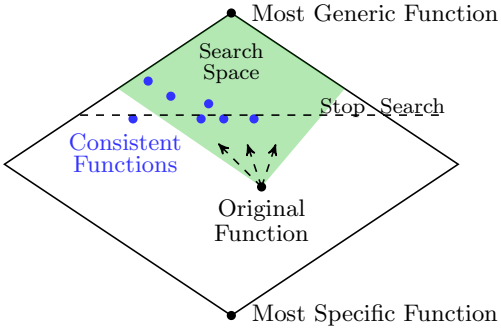


Figure 5.5: Near optimal search of Boolean functions assuming the original function has an entry that evaluates to 0 but a 1 is needed. The diamond shape represents the space of monotone non-degenerate Boolean functions. Direction of the search is indicated by dashed arrows. Search stops at the dashed line. In green is the search space considered.

However, with this search method, although all optimal solutions (if exist) are found, we may end up considering a huge number of function candidates. To avoid such searching we consider a near optimal search approach. If we are in a case where the value 1 is expected but a function evaluates to 0, it is probable to find a consistent candidate only looking at the ancestors of that function. Therefore, we can start at the original function and go up the Hasse diagram in a breath-first search. Note that with this method it is possible to not find a consistent candidate although it may exist, being not an ancestor of the original function. Moreover, even if a consistent function is found, it may not be the optimum candidate. Figure 5.5 illustrates the search for function repairs considering the near optimal approach. The search starts at the original function and goes up the diagram, and the search space considered is smaller than the one described previously (see Figure 5.4).

Note that these search methods for function repair operations are analogous when considering `r_part` inconsistencies, where a function evaluates to 1 but a value 0 is expected.

Considering the possible reasons of inconsistency (`r_gen` and `r_part`), so far it was described

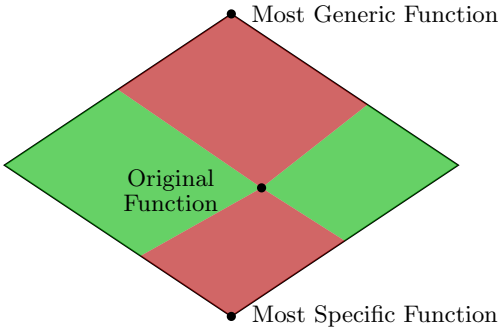


Figure 5.6: Search space of Boolean functions when in presence of a double inconsistency. The diamond shape represents the space of monotone non-degenerate Boolean functions. The area in red represents the space of impossible function candidates. The area in green represents the space of possible function candidates.

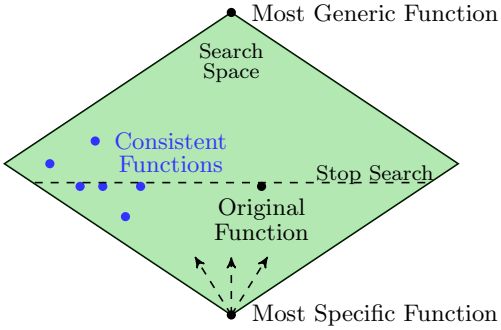


Figure 5.7: Search of Boolean functions when in the presence of a double inconsistency. The diamond shape represents the space of monotone non-degenerate Boolean functions. Direction of the search is indicated by dashed arrows. Search stops at the dashed line.

two search methods for repair operations when in the presence of only one reason (*i.e.* either `r_gen` or `r_part`). However, it may be the case where we are in the presence of the two types of inconsistency simultaneously. We call this case, a *double inconsistency*. This happens when there are two states of the network from which in one of them a regulatory function evaluates to 0 but a 1 is expected, and in the other the function evaluates to 1 but a 0 is expected. In this case we know that if a consistent regulatory function exists, it is neither an ancestor nor a descendant of the original function. Figure 5.6 illustrates the space of monotone non-degenerate Boolean functions in case of double inconsistency. The area in red represents the ancestors and descendants of the original function that are not possible function candidates of replacement.

To search for possible function repairs when in presence of a double inconsistency, a search can be made, similarly to the optimal search described before (Figure 5.4). In this case, if the original function is in the bottom half of the diagram (resp. top half), the search starts at the most specific function (resp. most generic function) and goes up (resp. down) the diagram. Figures 5.7 illustrates the search in a double inconsistency case, where the possible candidates are not ancestors nor descendants of the original function.

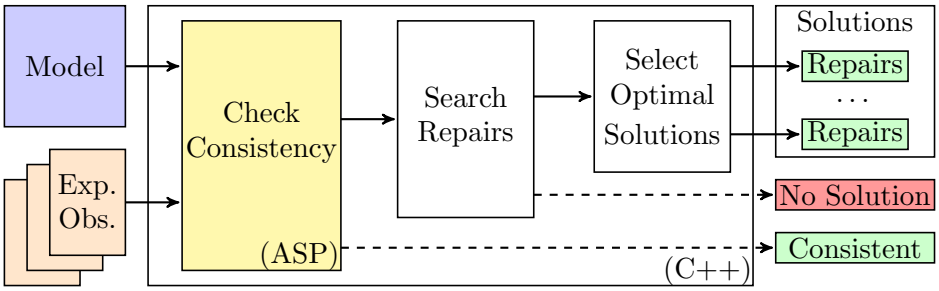


Figure 5.8: ModRev tool arquitecture. A model and a set of experimental observations are the input of the tool. The tool is mainly implemented in C++, and in yellow is represented the ASP component. The tool produces optimal sets of repair operations. Dashed arrows represent alternative flows.

5.2 ModRev - Model Revision Tool

ModRev is a freely available model revision tool for Boolean logical models of biological regulatory networks[1]. We developed ModRev, a command line tool, that implements the model revision methods presented in Chapter 4 and Chapter 5 (Section 5.1) to repair inconsistent models under stable state and time-series observations.

Considering a Boolean logical model and a set of experimental observations, ModRev determines whether the model is consistent with the observations. In case of inconsistency, it determines the minimum set of nodes that must be repaired. Four possible repair operations are considered: change a regulatory function; change the type of interaction (from activation to inhibition and vice-versa); remove a regulator; and add a regulator.

In order to repair an inconsistent model, the following lexicographic optimisation criterion, as defined in Section 5.1, is considered:

1. Minimise the number of add/remove edge operations;

2. Minimise the number of flip sign of an edge operations;

3. Minimise the number of function change operations.

This lexicographic criterion allows to give preference to function changes over changes in the structure of the network.

Figure 5.8 illustrates the tool architecture. Dashed arrows represent alternative flows, where it is not possible to repair a model, or the model is already consistent and no repair is needed. In white is represented the components of the tool implemented in C++, and in yellow the component (Check Consistency) implemented using ASP (see Section 4.1).

[1] https://filipegouveia.github.io/ModelRevisionASP/

Regulatory functions supported by ModRev are monotone non-degenerate Boolean functions. Biologically, a monotone function means that each regulator only has one role, either an activator, or an inhibitor, but not both. A non-degenerate function means that each regulator influences the output of the regulatory function. Otherwise, it should not be a regulator. This model revision process requires the regulatory functions to be represented in BCF [33, 34].

The ModRev tool is based on ASP [14], and the input is defined using ASP predicates, as described in Section 4.1.1. To represent a Boolean logical model we use the predicate `vertex(V)`, to indicate that `V` is a node of the regulatory graph, and the predicate `edge(V1,V2,S)` to represent an edge from `V1` to `V2` with a sign `S` $\in \{0,1\}$, where 0 (1) represents a negative (positive) interaction. The predicate `vertex` may be omitted if the node can be inferred from `edge` predicates. To represent regulatory functions, we use the predicate `functionOr(V,1..N)` that indicates that the regulatory function of `V` is a disjunction of `N` terms. The predicate `functionAnd(V,T,R)` is then used to represent that node `R` is a regulator of `V` and is present in the term `T` of the regulatory function.

ModRev is able to confront a model with a set of experimental observations, either in stable state, or a time-series data. To represent the set of experimental observations, the predicate `exp(E)` is used to identify an experimental observation `E`. To represent the observed values of an experiment `E`, we use the predicate `obs_vlabel(E,V,S)`, which means that node `V` in experiment `E` has an observed value `S` $\in \{0,1\}$ considering stable state observations. If time-series observations are to be considered instead, a similar predicate (`obs_vlabel(E,T,V,S)`) is used, where `T` represents the time-step of the observed value.

If ModRev identifies that a given model is not consistent with a set of observations, it produces all the optimum solutions (repairs) that render the model consistent. Considering the optimisation criterion defined above, the optimum set of repair operations are produced. However, one may want to prevent some of the repair operations. For example, if one is certain that a given interaction in a model is correct, solutions where that interaction is removed or its type is changed, are not desired. To give the user some control over the solutions produced, it is possible to classify any edge (or even any node) of the network as `fixed`. This can be done by adding to the definition of the model, given as input, a predicate `fixed(V1,V2)` meaning that the edge from `V1` to `V2` is fixed, or by adding a predicate `fixed(V)` meaning that node `V` is fixed. With this, solutions that affect `fixed` edges or `fixed` nodes are not considered. Note that with the addition of such definitions, an inconsistent model may become impossible to repair, due to over-constraining the problem.

When searching for function repair operations, ModRev uses by default the near optimal

search described in detail in Section 5.1.2. However, an option to force the optimum search is implemented, allowing the user to control the search method used for function repairs.

A tutorial with detailed examples of how to use the tool is presented in Appendix B.

Chapter 6

Experimental Evaluation

In this chapter we present the evaluation method used to test the proposed model revision approach. Section 6.1 presents the instances used in the evaluation method. The results are described in Section 6.2. This chapter ends with a discussion of the results in Section 6.3.

To the best of our knowledge, there is no other model revision approach over Boolean logical models with the same considerations and repair operations as the proposed approach. Therefore, there is no possible fair comparison with other model revision approaches to be made allowing us to gather conclusions. Some of the existent approaches consider the SCM which has specific rules for regulations based on sign algebra that cannot be changed [4, 16, 17]. Other model revision approaches over Boolean logical models consider fixed regulatory functions [19], different repair operations (or subset of operations) [20, 21], or do not take into account the impact of changing a regulatory function in the networks dynamic behaviour as the proposed model revision approach does.

6.1 Instances

To evaluate the proposed model revision approach, five well-known biological logical models are considered, representative of different processes and organisms: the cell-cycle regulatory network of Fission Yeast (FY) [68], with only 10 nodes and 27 edges being a very connected network, and 12 stable states; the Segment Polarity (SP) network which plays a role in the fly embryo segmentation [98], with the biggest regulatory function of the models considered with eight regulators; the T-Cell Receptor (TCR) signalling network [99], with 40 nodes and 57 edges, being a less connected network with an average regulatory function dimension of $1,425$; the core network controlling the Mammalian Cell Cycle (MCC) [100], with only 10 nodes and 35 edges, being the most connected network considered with an average regulatory function dimension

Table 6.1: Boolean models used for evaluation with corresponding: abbreviation (Abbr.), number of nodes (#N), number of edges (#E), number of stable states (#SS), average number of regulators (Avg.Reg.), maximum number of regulators (Max.Reg.), and associated reference (Ref.).

Abbr.	**Model**	**#N**	**#E**	**#SS**	**Avg.Reg.**	**Max.Reg.**	**Ref.**
FY	Fission Yeast	10	27	12	2,7	5	[68]
SP	Segment Polarity (1 Cell)	19	57	7	3	8	[98]
TCR	TCR Signalisation	40	57	7	1,425	5	[99]
MCC	Mammalian Cell Cycle	10	35	1	3,5	6	[100]
Th	Th Cell Differentiation	23	35	3	1,52	5	[101]

of 3,5; and the regulatory network controlling T-helper (Th) cell differentiation [101], with 23 nodes and 35 edges, being an average model when compared to the other models considered. Table 6.1 shows some characteristics of the five models used. These models range from 10 to 40 nodes, having different degrees of connectivity, and different number of stable states, allowing comparisons between models to assess the impact of some model properties in the model revision process. For example, TCR with 40 nodes and 57 edges is less connected than SP, which has the same number of edges but only 19 nodes, which can be verified by the dimension of the regulatory functions (average and maximum number of regulators).

Furthermore, in order to evaluate our method, we generate corrupted models from the original ones. We consider different types of corruptions: changing a regulatory function (F), flipping a sign of an edge (E), removing a regulator (R), or adding a regulator (A). Then, we generate corrupted models giving a probability of applying each of these types of corruption. We consider 24 different configurations of these corruption types (F, E, R, and A), as shown in Table 6.2. First, we consider configurations with single types of corruptions. With changes only on: the regulatory functions; the sign of edges; the regulators by removal; and the regulators by addition. Then, some configurations with more than one type of corruption are also considered. For each configuration, 100 corrupted models are generated, giving a total of 2 400 corrupted models of each original model.

To evaluate the proposed model revision approach, we confront the corrupted models with observations that are consistent with the corresponding original model to verify if the approach is able to repair the corrupted models in case of inconsistency with the given observations, or produce no repairs if the corrupted model is consistent with the observations. To evaluate the proposed method under stable state observations, the original stable states of each model are considered. Table 6.1 indicates the number of stable states of each model.

To evaluate the model revision process under time-series data, different sets of observations are randomly generated. Considering the original models, 10 different observations with 20 time

Table 6.2: Percentage (%) values of F, E, R, and A parameters, of the 24 corruption configurations (# Config.). Probability of changing a function(F). Probability of flipping the sign of an edge (E). Probability of removing a regulator (R). Probability of adding a regulator (A).

# Config.	F	E	R	A
1	5	0	0	0
2	25	0	0	0
3	50	0	0	0
4	100	0	0	0
5	0	5	0	0
6	0	10	0	0
7	0	15	0	0
8	0	20	0	0
9	0	25	0	0
10	0	50	0	0
11	0	75	0	0
12	0	0	1	0
13	0	0	5	0
14	0	0	10	0
15	0	0	15	0
16	0	0	0	1
17	0	0	0	5
18	0	0	0	10
19	0	0	0	15
20	25	5	0	0
21	50	25	0	0
22	100	50	0	0
23	5	25	5	5
24	10	10	5	5

steps for each model are generated. Of these 10 observations, 5 consider the synchronous update scheme and 5 the asynchronous update scheme. The observations generated are consistent with the original models. This permits the evaluation of the proposed approach, by feeding our method with a corrupted model and the corresponding set of observations, in order to verify if it can repair the corrupted model rendering it consistent with the observations given. This evaluation considers both synchronous and/or asynchronous update schemes. To assess the impact of the number of observations on the method performance, we consider different subsets of these generated observations. For each corrupted model we test our method with only one time-series observation, and with all the five observations simultaneously. Moreover, we consider observations with a different number of time steps, three and twenty time steps, in order to study the impact of the size of the observations on the method performance.

6.2 Results

In this section we present the results of different tests performed in order to evaluate our model revision approach. In the model revision process, the near optimal search of function repairs operations as described in Section 5.1.2 is considered for the evaluation. Note that the near optimal search of function repairs may result in non optimum solutions where a possible function repair may exist but is not found and a topological repair is considered instead. However, despite the exponential growth, this search method considers a smaller function space, with respect to the exhaustive search, improving the efficiency of the model revision approach.

All the experiments were run on an Intel(R) Xeon(R) 2.1GHz Linux machine with a memory limit of 2GB. A selection of these results are discussed in Section 6.3. More detailed results are presented in Appendix A.

6.2.1 Stable State Observations

To evaluate the model revision approach under stable state observations, we feed our method with each corrupted model and the corresponding set of stable state observations. In order to compare the impact of using the ASP program described in Section 4.2 for the function repair search, and the equivalent C++ implementation (library available online [1]), both approaches are tested.

Table 6.3 shows the average and median times in seconds for solved instances per model per corruption configuration (see Table 6.2), using ASP for function repair search. A time limit of 600 seconds is considered, and the corresponding number of timeouts are also shown in Table 6.3.

Table 6.4 shows the average and median times in seconds for solved instances per model per corruption configuration (see Table 6.2), using the C++ library developed for function repair search. A time limit of 600 seconds is also considered, and the corresponding number of timeouts are shown in Table 6.4.

To shed more light on the obtained results, Figure 6.1 illustrates a time comparison of solved instances per model using ASP for function repair operations search. Similarly, Figure 6.2 illustrates a time comparison of solved instances per model using the C++ library implemented for function repair operation search. Figures 6.1 and 6.2 show the SP model as the one with less number of solved instances, with higher solving times, and MCC as the model with the lowest solving times, where almost all of the 2 400 are solved. Moreover, Figures 6.1 and 6.2 clearly show that the TCR model has lower solving times and more solved instances than the SP model,

[1] `https://github.com/FilipeGouveia/BooleanFunction`

which has the same number of edges (57) and the same number of stable state observations (7), but is more connected, with only 19 nodes whereas the TCR model has 40 nodes.

As the results show, using the C++ library to compute the parents and children of monotone Boolean functions is strictly better than using the ASP approach. The results in Table 6.4 (C++ approach) show a higher number of solved instances than the results shown in Table 6.3 (ASP approach). Moreover, all the solved instances using ASP are solved faster by the C++ approach. Therefore, in the remaining tests only the C++ approach is considered.

In order to verify the impact of increasing the time limit considered in the number of solved instances, the time limit is increased to 3 600 seconds. The detailed results are presented in Appendix A (see Table A.1). Table 6.5 summarises the number of unsolved instances for all the approaches considered under stable state observations: using ASP for function search with a 600 second time limit, and using C++ for function search with 600 and 3 600 seconds of time limit. See Tables 6.1, 6.2, and A.1 for more details.

Results show when only functional corruptions are considered (type F corruptions), the model revision approach is able to repair all the models under the time limit of 600 seconds. Moreover, the average solving time is under 1 second. Timeouts begin to occur when topological corruptions are considered, *i.e.* corruptions of type E, R, and A, which change the topology of the network. Instances where new edges are added to the original model tend to take longer to solve, leading to a higher number of timeouts, than the instances where edges are removed, as shown in Tables 6.3 and 6.4. In general, when the number of unsolved instances increase, there is a bigger difference between the average and the median solving times, where some models quickly begin to have higher solving times. This phenomenon can also be observed in Figures 6.1 and 6.2.

Table 6.3: Results for model revision of FY, SP, TCR, MCC, and Th, under stable state observations using ASP for function repair search. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,036	0,028	0	0,035	0,030	0	0,044	0,037	0	0,019	0,016	0	0,031	0,031	0
25	0	0	0	0,043	0,041	0	0,057	0,056	0	0,054	0,053	0	0,022	0,016	0	0,047	0,047	0
50	0	0	0	0,055	0,045	0	0,084	0,083	0	0,067	0,064	0	0,030	0,027	0	0,076	0,076	0
100	0	0	0	0,073	0,058	0	0,138	0,136	0	0,102	0,099	0	0,041	0,035	0	0,133	0,129	0
0	5	0	0	0,352	0,051	7	30,018	0,686	2	0,137	0,041	0	0,049	0,014	0	3,900	0,046	5
0	10	0	0	1,919	0,098	19	66,462	21,577	18	0,238	0,056	0	1,099	0,034	0	18,742	0,099	9
0	15	0	0	4,567	0,157	31	86,897	32,201	28	0,312	0,090	0	4,198	0,064	0	34,991	0,187	14
0	20	0	0	8,014	0,186	43	99,264	34,315	54	0,407	0,127	0	9,901	0,108	2	41,761	0,235	20
0	25	0	0	10,355	0,378	49	108,946	43,391	65	0,445	0,599	0	20,201	0,183	2	54,127	0,597	27
0	50	0	0	26,150	10,956	82	515,652	515,652	99	0,700	0,713	0	130,428	79,366	20	124,088	1,606	55
0	75	0	0	16,091	0,409	97			100	1,001	0,874	0	227,534	221,829	82	150,014	21,866	85
0	0	1	0	0,171	0,034	6	1,845	0,034	4	2,826	0,031	0	0,014	0,010	0	1,235	0,018	3
0	0	5	0	0,427	0,049	14	4,638	1,192	15	12,044	0,038	0	0,016	0,011	0	3,537	0,371	7
0	0	10	0	0,833	0,091	20	10,554	2,526	25	14,046	0,061	0	0,018	0,011	0	3,665	0,414	10
0	0	15	0	0,895	0,109	27	11,126	3,037	30	20,102	5,329	0	0,018	0,011	0	5,458	0,807	11
0	0	0	1	0,584	0,036	0	22,019	0,451	23	5,589	0,101	0	0,048	0,011	0	0,411	0,050	13
0	0	0	5	22,414	0,383	5	107,806	6,329	76	70,951	3,268	57	0,628	0,036	1	60,581	0,590	68
0	0	0	10	25,627	3,169	18	377,806	416,840	97	4,400	4,400	98	4,749	0,227	3	133,721	16,471	93
0	0	0	15	53,629	3,522	34	-	-	100	-	-	100	7,255	0,281	7			100
25	5	0	0	0,169	0,057	11	18,438	0,289	5	0,163	0,063	0	0,042	0,025	0	0,833	0,066	7
50	25	0	0	13,936	0,767	31	123,227	86,083	65	0,589	0,302	0	31,957	0,229	3	23,477	0,419	24
100	50	0	0	33,740	7,280	69	229,134	269,408	93	0,874	0,941	0	97,079	39,688	30	44,347	2,021	56
5	25	5	5	11,651	0,806	53	142,766	62,721	97	33,351	2,712	82	39,836	0,610	14	86,067	10,743	72
10	10	5	5	7,927	0,490	33	108,483	31,067	87	37,640	3,872	76	7,178	0,124	7	62,835	1,868	69

Table 6.4: Results for model revision of FY, SP, TCR, MCC, and Th, under stable state observations using C++ for function repair search. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,021	0,021	0	0,022	0,022	0	0,032	0,033	0	0,010	0,011	0	0,014	0,014	0
25	0	0	0	0,021	0,022	0	0,022	0,023	0	0,031	0,032	0	0,011	0,011	0	0,014	0,014	0
50	0	0	0	0,021	0,021	0	0,023	0,024	0	0,033	0,034	0	0,011	0,010	0	0,014	0,014	0
100	0	0	0	0,021	0,021	0	0,026	0,027	0	0,034	0,035	0	0,010	0,009	0	0,015	0,015	0
0	5	0	0	9,922	0,023	5	2,368	0,030	1	0,032	0,033	0	0,011	0,008	0	3,184	0,016	5
0	10	0	0	23,355	0,026	15	10,703	0,195	6	0,033	0,034	0	0,019	0,012	0	21,346	0,015	7
0	15	0	0	49,343	0,030	23	19,322	0,290	10	0,034	0,034	0	0,256	0,013	0	42,021	0,018	9
0	20	0	0	67,822	0,030	34	28,865	0,988	16	0,034	0,035	0	6,141	0,015	0	50,489	0,022	12
0	25	0	0	82,597	0,034	39	33,039	1,093	20	0,035	0,035	0	6,566	0,019	0	78,785	0,023	17
0	50	0	0	60,904	2,621	80	100,783	20,569	67	0,038	0,038	0	33,729	0,960	1	90,984	0,571	22
0	75	0	0	119,655	3,647	96	32,784	19,486	96	0,040	0,041	0	114,468	110,227	38	53,441	0,955	10
0	0	1	0	0,026	0,023	6	0,179	0,023	0	0,041	0,033	0	0,010	0,009	0	0,019	0,015	3
0	0	5	0	0,243	0,023	13	0,435	0,030	0	0,065	0,034	0	0,010	0,009	0	0,122	0,017	7
0	0	10	0	0,243	0,023	17	0,711	0,165	0	0,061	0,031	0	0,010	0,008	0	0,221	0,020	10
0	0	15	0	0,742	0,024	18	0,759	0,129	0	0,085	0,055	0	0,010	0,009	0	0,266	0,020	11
0	0	0	1	0,043	0,024	0	2,673	0,035	11	0,053	0,036	0	0,010	0,009	0	2,702	0,017	2
0	0	0	5	7,533	0,046	0	24,186	16,020	36	31,402	0,959	29	3,559	0,012	0	27,395	1,375	40
0	0	0	10	21,785	0,146	6	69,102	21,879	87	146,019	47,995	87	3,919	0,016	0	93,597	25,672	72
0	0	0	15	35,451	0,213	12	216,824	216,824	99	-	-	100	6,247	0,020	0	183,331	261,087	97
25	5	0	0	23,203	0,025	6	2,913	0,029	1	0,034	0,034	0	0,011	0,010	0	6,270	0,018	6
50	25	0	0	76,736	0,034	20	30,432	1,183	31	0,037	0,037	0	4,807	0,022	0	57,293	0,023	6
100	50	0	0	185,835	7,146	51	87,609	31,990	54	0,044	0,044	0	32,196	2,097	3	59,685	1,604	20
5	25	5	5	42,181	0,219	36	87,096	28,554	86	16,517	0,387	61	8,092	0,035	5	61,837	1,302	46
10	10	5	5	34,585	0,045	16	29,215	6,134	64	29,280	1,854	35	0,839	0,016	2	33,992	1,252	42

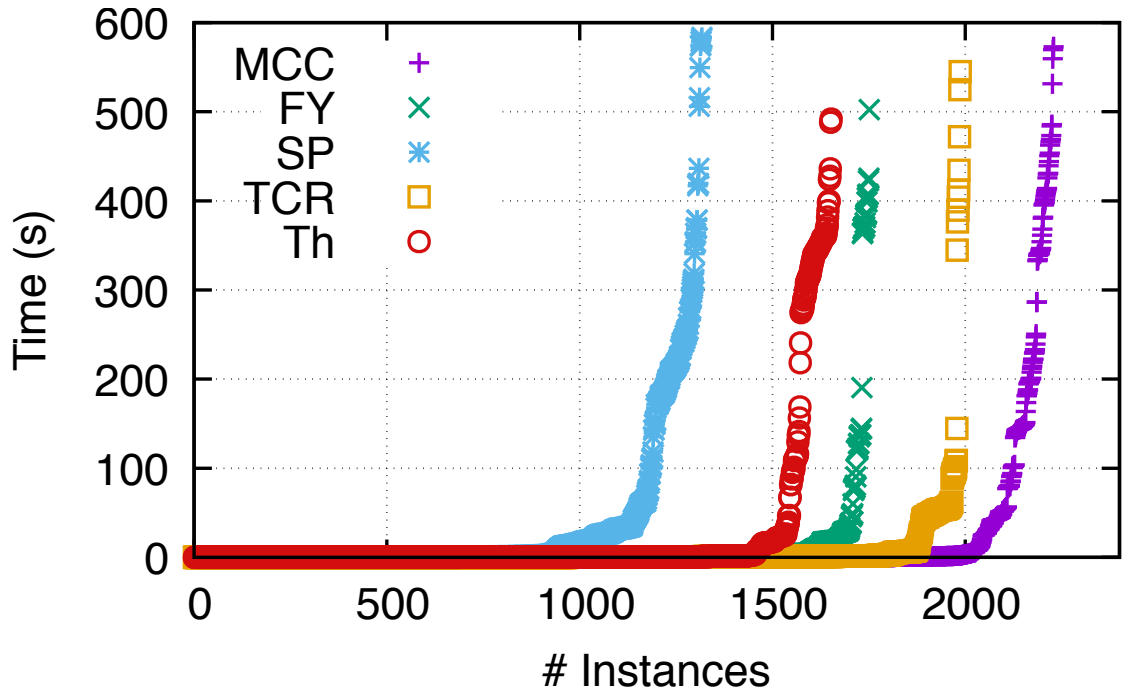


Figure 6.1: Time in seconds of solved instances for each model, under stable state observations, using ASP for function search.

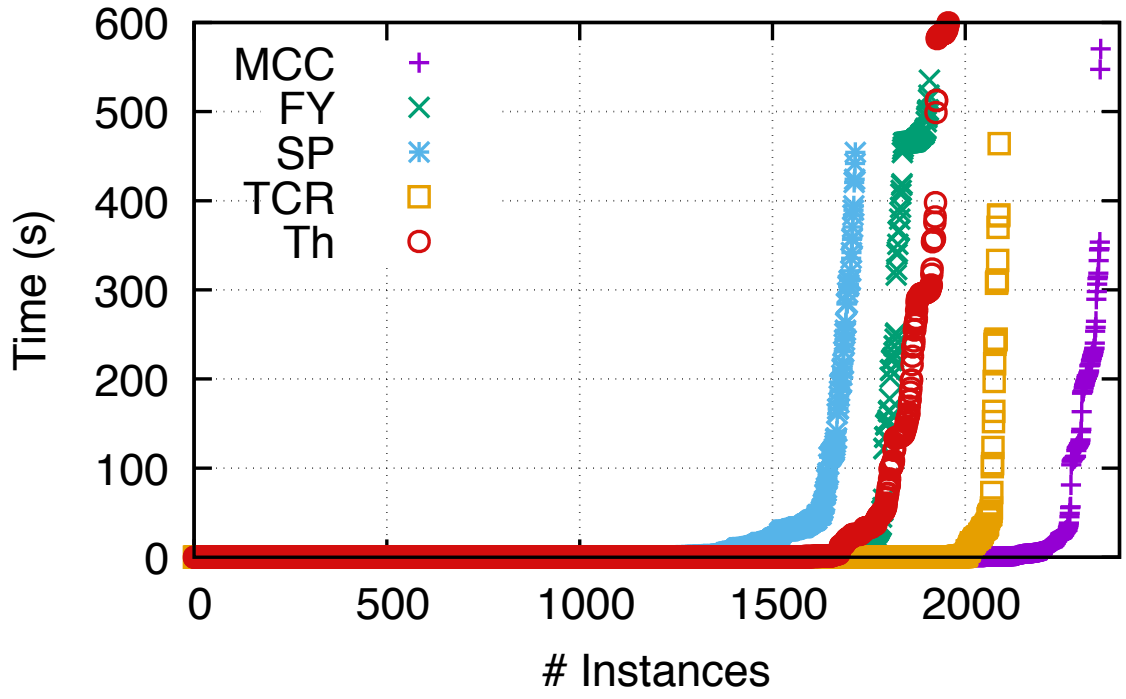


Figure 6.2: Time in seconds of solved instances for each model, under stable state observations, using C++ for function search.

Table 6.5: Number of solved and unsolved instances under stable state observations, considering different approaches for function repair operations search (Function search) and different time limits in seconds. A total of 12 000 instances are considered.

Function Search	Time Limit (s)	Solved Instances	Unsolved Instances
ASP	600	8 936	3 064
C++	600	10 017	1 983
C++	3 600	10 718	1 282

6.2.2 Time-series Observations

The proposed model revision approach is also evaluated by confronting corrupted models with time-series observations. We consider confronting each model with only one observation and with five observations simultaneously. Different sizes of time-series observations are also considered, having three and twenty time steps. Additionally, when confronting a model with the corresponding set of time-series observations, both synchronous and asynchronous update schemes are considered. Note that the time-series observations are consistent with the corresponding original models. A time limit of 3 600 seconds is considered in these experiments.

Table 6.6 shows the results obtained for each model considering different update schemes, one and five observations of three and twenty time steps. The average and median solving times are also shown, as well as the percentage of solved instances, considering 2 400 instances per entry. More detailed results considering time-series observations are presented in Appendix A, in which results are separated by model corruption operations (*e.g.* see Tables A.4 and A.5).

Results in Table 6.6 show that increasing the number of observations or the number of time steps, the solving time increases and the number solved instances decreases. Table 6.6 also shows that the number of solved instances is higher considering the asynchronous update scheme than the synchronous update scheme. Considering the asynchronous update scheme, $91,82\%$ of the instances were solved, and considering the synchronous update scheme, $76,84\%$ of the instances were solved, under the time limit defined.

Figure 6.3 shows the solving times for each model when confronted with five time-series observations with twenty time steps under a synchronous update scheme. Results clearly show that more connected networks, *i.e.* networks with a higher ratio of edges per node, tend to take longer to solve and have a higher number of unsolved instances. In fact, ordering the models by the ratio of edges per node in non increasing order, we obtain: MCC, SP, FY, Th, and TCR. This order can be seen in the results shown in Figure 6.3

Table 6.6: Results for model revision of FY, SP, TCR, MCC, and Th, under time-series observations. Median (Med.) and average (Avg.) solving times in seconds for each model considering: synchronous (S) and asynchronous (A) dynamics; 1 and 5 observations (Obs.); and 3 and 20 time steps (T.S.). The percentage of solved instances (%) consider a total of 2 400 instances for each table entry.

Update	Obs.	T.S.	FY			SP			TCR			MCC			Th		
			Time (s)		Solved	Time (s)		Solved	Time (s)		Solved	Time (s)		Solved	Time (s)		Solved
			Avg.	Med.	(%)	Avg..	Med.	(%)	Avg.	Med.	(%)	Avg.	Med.	(%)	Avg.	Med.	(%)
S	1	3	224,193	0,017	95,63	59,863	0,023	67,08	26,180	0,028	87,54	15,867	0,019	94,67	95,654	0,021	84,04
		20	325,370	0,044	90,75	63,700	0,068	66,79	29,726	0,087	86,46	40,412	0,052	59,71	62,861	0,055	82,88
	5	3	413,305	0,132	85,08	63,765	0,070	58,38	185,533	0,104	79,79	466,870	0,242	80,29	34,372	0,112	76,54
		20	382,021	0,506	82,54	72,647	0,772	53,38	104,566	0,656	78,96	277,921	0,640	53,04	28,779	0,661	73,33
A	1	3	16,724	0,023	99,88	0,032	0,031	100,00	4,124	0,040	96,54	0,040	0,024	99,96	1,242	0,029	99,29
		20	230,888	0,538	94,83	38,453	0,748	88,67	19,126	0,880	91,50	5,545	0,569	99,50	66,106	0,509	88,75
	5	3	17,735	0,300	99,50	7,233	0,436	99,04	11,407	0,527	93,42	16,014	0,317	95,13	22,963	0,313	95,17
		20	254,957	16,086	94,54	240,310	28,287	48,42	138,833	28,545	91,33	93,915	17,216	77,88	50,362	16,535	83,04

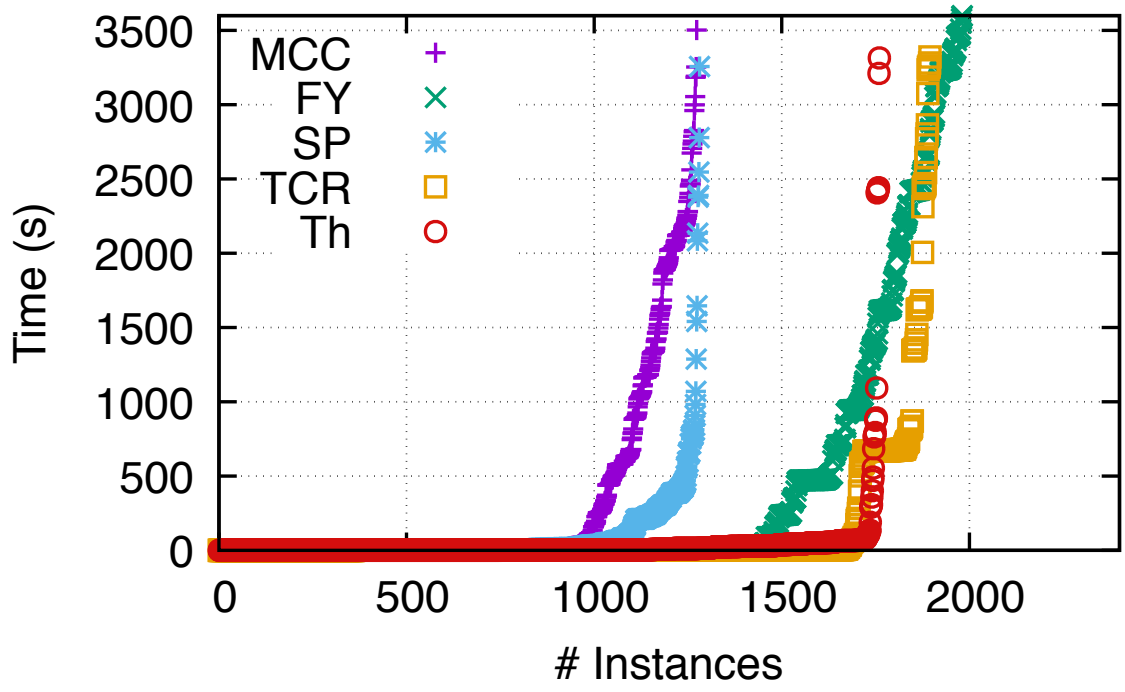


Figure 6.3: Time in seconds of solved instances for each model, confronted with five time-series observation of twenty time steps, under the synchronous update scheme.

Figure 6.4 shows a comparison between the FY and MCC models when confronted with five time-series observations with twenty time steps, under synchronous and asynchronous update schemes. It is shown that less MCC instances are solved (and higher solving times) than the instances of the FY model, which is a less connected model than the MCC model. Moreover, Figure 6.4 also shows that the results are better when considering the asynchronous update scheme than the synchronous update scheme. Considering the asynchronous update scheme, most of the instances of both models can be solved under ten seconds.

Figure 6.5 shows in detail the median solving times for the SP model when confronted with five time-series observations with twenty time steps. These results show the median time per corruption configuration (see Table 6.2) considering the synchronous update scheme. Figure 6.5 shows that the solving time increases when topological changes are performed as corruption operations. The first five corruption configurations correspond to function changes (see Table 6.2), having the lowest median times, below one second.

More detailed results regarding time-series observations are presented in Appendix A. Graphics comparing different number of observations and time steps for each model under both synchronous and asynchronous update scheme are presented in Appendix A (*e.g.* see Figures A.1 and A.2).

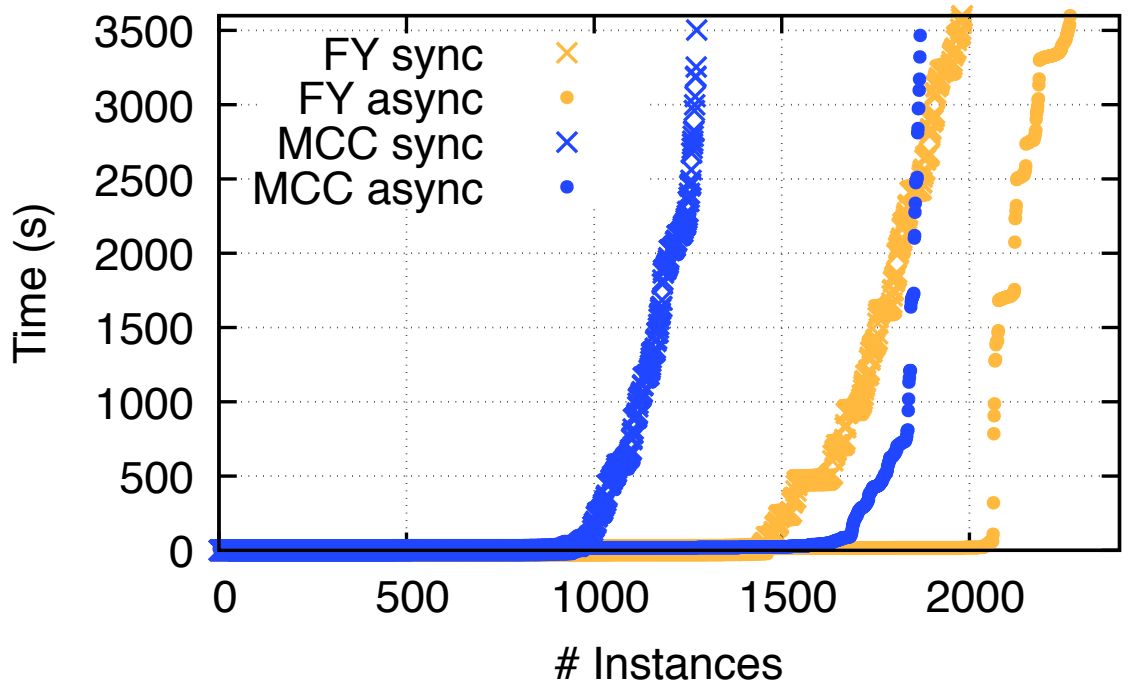


Figure 6.4: Time in seconds of solved instances for FY and MCC models under synchronous and asynchronous update schemes. Models are confronted with five different time-series observations with twenty time steps.

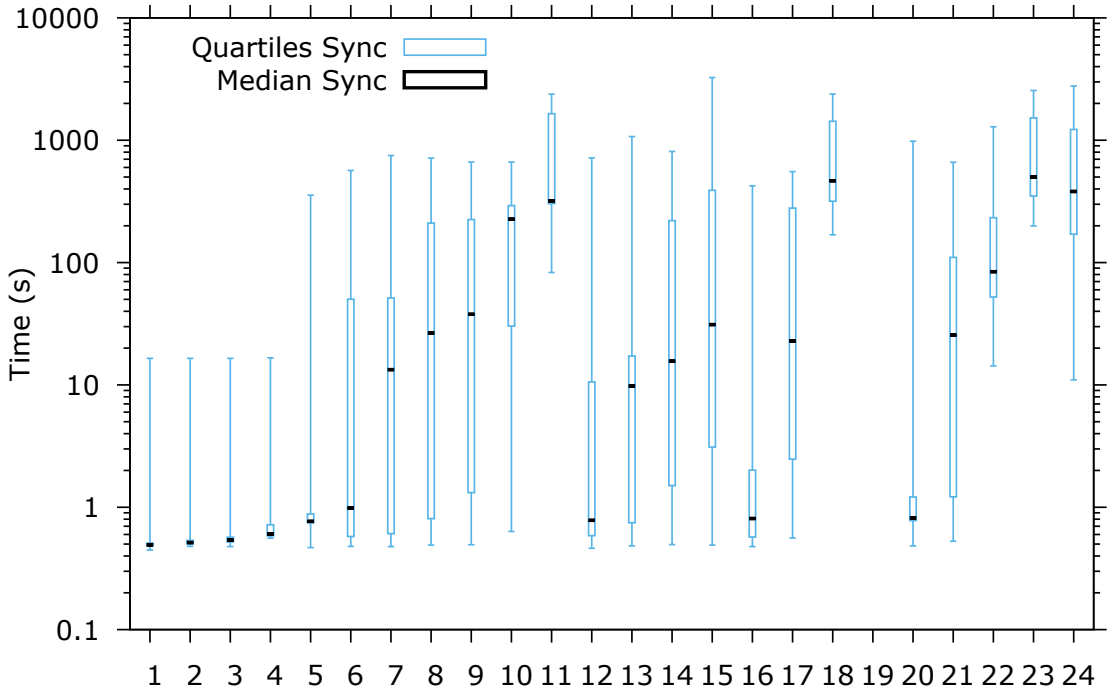


Figure 6.5: Median time in seconds of solved instances of SP model under the synchronous update scheme when confronted with five time-series observations with twenty time steps.

6.3 Discussion

The results presented in Section 6.2 shed a light on the impact of several characteristics in the presented model revision approach.

Impact of the type of corruption

Different types of corruption operations are applied to the original models in order to test the ability of the presented approach to revise the corrupted models. Topological corruptions, *i.e.* changing the sign of the edges, removing edges and adding edges, to the models take more time to solve. In particular, when new regulators are added, the number of unsolved instances is very high, having configurations in which no instance could be solved under the time limit defined. This happens because by increasing the number of regulators, we increase the dimension of the corresponding regulatory function. Given the optimisation criterion defined for the search of repair operations, the function repair search is the first to be performed. As the number of monotone non-degenerate Boolean functions increases with the dimension of the function (see Table 4.1), the search space for function repair increases when we add new regulators as a corruption operation. Therefore, the solving time increases when new edges are added. Moreover, when searching for possible repair operations, we first search for function repair operations, then we search for possible changes in the sign of the edges, and finally consider addition or removal of edges as repair operations. When considering changing the sign of an edge, or adding or removing edges, the search for possible consistent Boolean functions must be performed again, evidencing the impact that the addition of new edges (as a corruption operation) has on the solving time of the proposed approach.

When the original models are corrupted by only changing the signs of edges, we observe an increasing number of unsolved instances and solving times. We observe that when the signs of edges are changed, most of the inconsistencies (when present) are *double inconsistencies*, *i.e.* inconsistent nodes in which the corresponding regulatory function should be more generic and more specific at the same time (see Section 5.1.2 for details). This means that, if a consistent regulatory function exists, is not comparable with the original function, forcing a complete search of the function space. Considering the whole space of Boolean functions translates to higher solving times and consequently, higher number of unsolved instances considering the time limits defined (3 600 seconds). These results show that the impact that the complete search of possible function repairs has on the efficiency of the presented model revision approach, justifies the choice of implementing and using by default the near optimal search for function repairs whenever possible. This choice revealed to be efficient, and the results show that the majority

of the instances can be solved under 60 seconds.

The impact that the type of corruption has on the model revision process can be verified by the results shown in Figure 6.5, in which the solving time increases when considering topological changes in the original model (see Table 6.2).

Solving time analysis

There is a difference of the average and median times of solved instances of most of the experimental results, where the average is higher than the median. This means that most instances have a lower solving time than the average time. In fact, Figures 6.1, 6.2 and 6.3 show precisely that the solving time is mostly lower for the majority of the instances and the solving time quickly grows for the remaining few instances. This can be observed when the time limit is increased from 600 seconds to 3 600 seconds under stable state observations, where the number of solved instances only increases 7%. With a 600 seconds time limit there are 10 017 solved instances, whereas with 3 600 seconds time limit there are 10 718 solved instances (see Table 6.5).

Model characteristics

The connectivity of the model plays a significant role in the model revision process. More connected models, *i.e.* models with higher ratio of edges per node, typically have regulatory functions of higher dimension, leading to higher solving times. This fact can be seen in Figures 6.2, 6.3, and 6.4. A higher function dimension leads to a bigger search space for function repairs, and therefore, less solved instances under the predefined time limit. For example, comparing the solving times of MCC with FY (Figure 6.4), we can verify that the former has higher solving times. In fact, both MCC and FY models have 10 nodes, but FY only has 27 edges whereas MCC has 35 edges, making MCC a more connected model (see Table 6.1). The same phenomenon can be verified by comparing the SP with the TCR model (Figure 6.3). Both models have 57 edges, but SP has 19 nodes whereas TCR has 40 nodes, making SP a more connected model and, therefore, with higher solving times (considering the same number of observations).

Impact of the number of observations

Taking a closer look at Figure 6.2, the solving time decreases with less connected models, with the exception of the MCC model. The MCC model is not the model with the lowest ratio of edges per node, however, is the model with lower solving times under stable state observations. This happens because MCC only has one stable state observation, whereas the other models have multiple stable states. This indicates that the number of observations plays a significant role on

the efficiency of the model revision procedure. The same behaviour can be verified by the results showed in Table 6.6. Each observation represents a constraint on the model behaviour when searching for repair operations in case of inconsistency. The higher the number of observations, the higher the number of constraints over the model. Therefore, with an increasing number of observations there is a higher probability of the necessity of a higher number of repair operations to render an inconsistent model consistent. This leads to increasing solving times. Moreover, considering time-series observations, not only the number of solved instances decreases with an increase in the number of observations, but also with an increase in the number of time-steps, as shown in Table 6.6. The reason behind these results is the same: each time-step represents a constraint over the model behaviour.

Comparing ASP and C++ approach for function search

Two different approaches are presented to compute parents and children of monotone non-degenerate Boolean functions, used in the search of function repair operations: one in ASP and another in C++ (see Section 4.2). Both approaches are considered under stable state observations, as shown in Table 6.3 and Figure 6.1 (ASP), and Table 6.4 and Figure 6.2 (C++). Results show that the C++ library developed for function search produces significantly better times than the equivalent ASP approach. The average and median solving times using C++ (Table 6.4) are lower than using ASP (Table 6.3) in most scenarios. When the solving times are not lower, there is an increase of solving instances (less number of timeouts). Therefore, the C++ approach for function search solves instances faster and solves more instances. This is due to the fact that in ASP all the possible instances of the defined predicates (see Section 4.2) are generated to compute a neighbour of a given function, and then a solution is computed considering the defined ASP rules. Instantiating all the possibilities of variable-free predicates has an exponential growth with the increase of the function dimension. Using a dedicated procedure to compute parents and children of monotone non-degenerate Boolean functions, avoiding the generation of all the possible combinations, results in a more efficient approach.

Comparing Figures 6.1 and 6.2, an interesting result is observed. Using ASP for function search, model Th has higher solving times (and more timeouts) than the FY model, whereas with the C++ approach the reverse happens. The FY model is a more connected model than Th, and, considering stable state observations, FY has more observations, which justifies the results shown in Figure 6.2. However, using ASP for function search, the results are not consistent with the expected behaviour, indicating that there is some underlying property influencing the results. Looking closer to Tables 6.3 and 6.4, comparing models FY and Th, we can see that the difference in solved instances (times and timeouts) is bigger in the Th model under corruption

configurations where only the sign of the edges are changed (E). In these configurations, considering the Th model, we are able to reduce the number unsolved instances by (approximately) half, with the C++ approach. As mentioned before, in corruptions where only the sign of edges are changed we tend to have nodes with *double inconsistencies*, which means that the whole function space is considered in function repair search, having a greater impact on the model revision approach. Although the Th model has on average regulatory functions with lower dimension than the FY model, as shown in Table 6.1, the repair search is slower in the Th model using ASP, contrary to what would be expected. The Th model is bigger than the FY, *i.e.* has more nodes, which leads to more inconsistent nodes upon corruption, which in turn leads to more function repair searches during the model revision procedure. The obtained results indicate that whenever there are more functions searches and the complete function space is considered, the ASP approach suffers a greater impact. Thus the discrepancy in the results shown in Figure 6.1 in which the Th model has bigger solving times than the FY model. Moreover, comparing the FY with the Th model, the number of unsolved instances (considering a 600 seconds timeout) of FY and Th using ASP are 649 and 748, respectively, and considering the C++ approach for function search the number of unsolved instances are 493 and 444, respectively. We have a bigger reduction in the number of unsolved instances in the Th model when considering the C++ approach. Moreover, the median solving time of the 304 extra instances that we are able to solve is $28,060$ seconds, indicating that most of these instances can be solved quickly.

Synchornous vs. asynchronous update scheme

When considering time-series observations, Table 6.6 clearly shows that the model revision approach performs better under the asynchronous update scheme than under the synchronous update scheme. This is due to the fact that the synchronous update scheme has more restrictions, since all the regulatory functions must produce the expected value of the corresponding node at every time step. Considering that we have one inconsistent node, with five observations and twenty time steps, in the synchronous update scheme, $(5 \times 20 =)$ 100 verifications must be performed when repairing that node. On the other hand, in the asynchronous update scheme, for each time step only one node is updated, and therefore only that node must be verified, leading to considerably less verifications when repairing an inconsistent node. In fact, when considering the asynchronous update scheme with one observation with three time steps, which represents the least amount of restrictions considered in the model revision process, results show that many corrupted model instances are consistent with the observation.

Solutions quality

Analysing the repair sets of operations produced by our model revision approach, we verified that in the majority of the instances, the number of repair operations produced in a solution is equal or less than the number of operations applied to corrupt the original model. The instances where the number of repair operations is higher than the number of operations to corrupt the original model are justified by the optimisation criterion defined (see Section 5.1). For example, if, to corrupt a model, one corruption operation is applied by adding one new edge, there might be a solution where three repair operations are produced: changing a function and changing the sign of two edges. In fact, by adding a new edge (as a corruption operation), we are increasing the function dimension, being likely to find a consistent function, even if changing the sign of edges is necessary. The ideal reverse operation of the example to repair the model would be to remove the added edge (one operation). However, following the defined optimisation criterion, a solution with one function change and two signs of the edge flipped (three operations) is better than a solution with one remove operation, and therefore it is the solution produced by the presented model revision approach. With the presented optimisation criterion, changing the structure of the model is preferred to be avoided whenever possible.

The optimisation criterion of repair operations considered takes into account the construction process of the Boolean logical models, where it is assumed that there is a higher level of confidence in the correctness of the topology of the model, than in the correctness of the regulatory functions. Therefore, when searching for repair operations, the process starts with function repair searches. Note that instances where only topological corruptions are applied, are against the above assumption, and therefore may lead to unideal solutions. However, such instances are considered in order to evaluate the impact of the different corruption types in the proposed model revision approach. Nevertheless, optimal solutions are found taking into account the optimisation criterion, compliant with the assumption that the most probable incorrect components of a model are the regulatory functions.

Chapter 7

Conclusions

In this work, a model revision procedure capable of repairing Boolean logical models is presented, considering stable state and time-series observations. It verifies the consistency of a model with a set of observations and, in case of inconsistency, all optimal sets of possible repair operations are produced which render the model consistent. In the model revision procedure, four repair operations are considered:

- Change regulatory function;
- Change the type of an interaction;
- Remove a wrong regulator;
- Add a missing regulator.

To the best of our knowledge, this approach is the only one considering these four types of repair operation simultaneously. Moreover, when searching for function repair operations, the impact that changing a function has in the networks dynamic is considered. To select optimal solutions, a lexicographic optimisation criterion that takes into consideration the model construction process, being an innovative approach, is defined as follows:

1. Minimise the number of add/remove edge operations;
2. Minimise the number of flip sign of an edge operations;
3. Minimise the number of function change operations.

The method is successfully tested using several well-known biological models. Corrupted versions of the original models are generated in order to assess if the proposed procedure is able to determine if a corrupted model is consistent with a given set of observations, and in case of

inconsistency, if the procedure is able to repair it. To evaluate the model revision approach under stable state observations, the stable states of the corresponding original models are considered. Different time-series data, consistent with the original models, are also generated in order to evaluate the presented procedure under time-series observations, considering both synchronous and asynchronous update schemes. The model revision approach is able to repair most of the corrupted models in less than one second.

We conclude that the degree of connectivity of the model plays a big role in the model revision procedure. More connected models, *i.e.* models with a higher ratio of edges per nodes (likely having regulatory functions with higher dimensions), take longer to repair. Moreover, the type of corruption also influences the model revision procedure. For models where the corruption led to repairs considering the addition or removal of regulators, the repair time increased, due to the change of the function space and subsequent search of a compatible function.

With this model revision method, we conclude that the dimension of the regulatory functions has the biggest impact on the performance, since the search space of function repair increases exponentially with the number of regulators.

When comparing different update schemes of the model dynamics, the synchronous update scheme has more restrictions than the asynchronous update leading to higher solving times. This is due to the fact that, in the synchronous case, all the regulatory functions must produce the expected values at every time step. In the asynchronous update, only one regulatory function is applied at each time step, and therefore only the updated function must produce the expected value at each time step.

With this work, we are able to repair Boolean logical models of biological regulatory networks considering stable state and time-series observations, and both synchronous and asynchronous update schemes for the model dynamics.

7.1 Contributions

During the development of this work, relevant contributions for the Computer Science and Bioinformatics communities were created. The modelling of Boolean logical models of biological regulatory networks using ASP is presented in "*Model Revision of Logical Regulatory Networks Using Logic-Based Tools*" [102]. In this paper it is also presented an initial version of the ASP program for the consistency check procedure under stable state observations described in Section 4.1. The paper was presented in the Doctoral Consortium of the 34^{th} International Conference on Logic Programming (ICLP 2018), allowing to validate the work and getting important feedback for the work that followed.

A first approach of the model revision method under stable state observations is described in "*Model Revision of Boolean Regulatory Networks at Stable State*" [103], presented in the 15^{th} International Symposium on Bioinformatics Research and Applications (ISBRA 2019). This work validated the model revision approach, shedding light on the impact that the degree of connectivity of a model has on solving time.

A detailed study of monotone non-degenerate Boolean functions, in particular the search of consistent function repairs, is presented in "*Revision of Boolean Models of Regulatory Networks Using Stable State Observations*" in the Journal of Computational Biology [104]. The presented methods for function repair search improved the previous work, allowing us to compute optimal or near-optimal solutions. Since the search for function repairs has the greatest impact on the model revision process, being able to compute near-optimal solutions is of great importance.

The model revision procedure of Boolean logical models when confronted with time-series observations is described in "*Semi-automatic model revision of Boolean regulatory networks: confronting time-series observations with (a)synchronous dynamics*" [105]. The paper presents the consistency check procedure in ASP considering the dynamic behaviour under synchronous and asynchronous update schemes, and the search method for repair operations under time-series observations in case of inconsistency.

This work culminated in the development of the command line tool ModRev presented in "*ModRev - Model Revision Tool for Boolean Logical Models of Biological Regulatory Networks*" [106] in the International Conference on Computational Methods in Systems Biology (CMSB 2020). The ModRev tool is freely available online[1]. ModRev is capable of confronting Boolean logical models with either stable state or time-series observations and assess whether the model is consistent. In case of inconsistency, the tool searches for possible repair operations, and all the optimal sets of repair operations are produced. When considering time-series observations, both synchronous and asynchronous update schemes are supported. The tool allows the user to control some parameters in the search of function repairs, improving efficiency at the cost of possibly loosing some optimal solutions. Moreover, the user is able to prevent some repair operations to be considered, having some control in the model revision process (see Section 5.2).

[1] https://filipegouveia.github.io/ModelRevisionASP/

7.2 Discussion and Future Work

Repair Sets

Our proposed approach may produce multiple solutions, which can be overwhelming for the user. Heuristics to filter the produced output could be considered, in particular those proposed by Merhej et al. [19] where rules of thumb are discussed. These rules are based on expert knowledge and properties found in literature, such as the number of edges, or the diameter of a network, and can be used to narrow down the number of solutions presented by our model revision approach.

Regulatory Functions

In this work we consider that the regulatory functions are defined as monotone non-degenerate Boolean functions. The domain of Boolean functions considered has a biological meaning. Monotone Boolean functions are functions where each regulator only has one role, either an activator, or an inhibitor, but not both. In non-degenerate functions every regulator affects in some way the output of the regulatory function, otherwise it should not be considered a regulator. Despite the biological meaning, restricting the domain of Boolean functions may lead to undesired results. Considering degenerate Boolean functions means that functions can be defined with unnecessary variables that do not affect the output of the function, and can be simulated by adding or removing regulators which our method supports. However, it would be possible to consider degenerate functions, for which the Hasse diagram is well defined, in our search for function repairs by adapting the rules to compute neighbours of a function (see Section 2.2). This approach may lead to solutions where regulatory functions are changed to degenerate functions, suggesting that some regulators are not necessary and could be removed, without explicitly search for topological repairs. However, with our approach we are not able to consider non-monotone Boolean functions without changing the function repair search method. Moreover, we rely on the properties of the family of monotone Boolean functions to consider the impact on the networks dynamics when changing a regulatory function. Additionally, assuming that each regulator can only have one role, in order to preserve the coherence between the topology of the network and the regulatory functions, monotone functions must be considered.

Results showed that the dimension of the regulatory function has the biggest impact on the performance of the model revision approach, since the search space of function repairs increases exponentially with the number of regulators. Here, we show that using a near optimal search of functions leads to very good results where the number of repair operations produced are less than or equal to the number of operations performed to corrupt the model, and in

a few seconds/minutes. Nonetheless, an exhaustive search method for function repairs is also implemented and available to the user.

Answer Set Programming

Two different implementations of the function search process are developed, one in ASP and another in C++. With the results, we conclude that for the C++ approach performs better than the ASP approach. In fact, in combinatorial problems, like the computation of monotone Boolean functions neighbours, where not all the possible combinations are necessary to be considered and there are well defined rules to generate only the necessary combinations, results indicate that a dedicated procedure performs better than an ASP approach. Logic-based approaches such as ASP consider all possible solution combinations and use defined rules to cut or discard impossible solutions. However, in combinatorial problems where all possible combinations should be considered, such as the consistency check procedure presented, an ASP approach revealed to perform well, allowing the inference of additional useful information.

For the consistency check procedure, the ASP program developed revealed to be suitable, leading to good results. However, it would be interesting, as future work, to consider different logic-based approaches such as Boolean Satisfiability (SAT), or Satisfiability Modulo Theories (SMT), which have already been applied in the context of biological regulatory networks [17, 72, 107].

Stable State and Time-series Observations

The proposed model revision approach is able to confront a Boolean logical model with a set of time-series observations, considering both synchronous and asynchronous update scheme. It is considered that the simulation time of the time-series observations is known. Moreover, it is assumed that two consecutive observations require only one transition (time step). However, this may not be realistic. It is desired to be able to consider that between two consecutive observations there can be any number of intermediate transitions. However, this would require the consideration of exponentially long trajectories. To overcome this, an iterative approach could be considered, with a higher number of intermediate transitions at each passing iteration. Although an upper bound can be defined, being the number of all possible network states, this approach may be inefficient for large networks, since to compute optimal solutions, all the possibilities must be considered. If a lower upper bound is to be considered, it could potentially lead to false conclusions.

In this work we are only able to consider either a set of stable state observations, or a set of time-series observations, but not both simultaneously. As future work, we intend to be able to

address this limitation, considering mixed sets of observations. To do so, the consistency check procedure must be adapted, as well as the verification process of the possible repair operations.

Regarding time-series observations, both synchronous and asynchronous update schemes are supported. For the asynchronous update scheme, we consider that only one regulatory function is updated at each time step. As future work we intend to be able to support fully asynchronous update scheme, in which any number (at least one) of regulatory functions can be updated at each time step. An experimental approach for the fully asynchronous update scheme is already implemented in the presented ModRev tool.

Optimisation

In this work several optimisation criteria are considered in order to revise a Boolean logical model. In the model revision process, a consistency check procedure is considered firstly to determine whether a given model is consistent when confronted with a set of observations. When repairing a model, we desire to affect the least amount of nodes, and therefore, in case of an inconsistent model, the consistency check procedure determines the minimum set(s) of inconsistent nodes. Although this optimisation directive is compliant with the assumption that we want to change the least amount of components in the model, it may lead to non optimal solutions considering the optimisation criterion defined for the search of repair operations. For example, the consistency check procedure may consider two solutions: one with one inconsistent node; and another with two inconsistent nodes. Only the former is considered to the search of repair operations as it has the minimum number of inconsistent nodes. However, the solution with one inconsistent node may lead to a repair solution containing three repair operations, including topological changes, whereas the solution with two inconsistent nodes (if it were considered), could lead to a repair solution comprising only two repairs (function changes). This latter case is better then the former considering the optimisation criterion defined, however is never considered due to the optimisation directive in the consistency check procedure. It would be interesting to consider different optimisation directives in the consistency check procedure, or to be able to provide the user the option to ignore such directives, allowing the possibility of different solutions.

Besides the optimisation directive to minimise the number of inconsistent nodes in the consistency check procedure, there are other optimisations relative to the number of topological errors and possible reasons of inconsistency (see Section 4.1). These directives may also conceal possible repair solutions, and therefore we consider, as future work, allowing to disable such directives, or even defining different directives, in order to overcome this limitation.

Experimental Evaluation

In this work we consider five well-known biological models with different characteristics to evaluate our model revision approach. Although the evaluation process allows to understand the impact of some characteristics of the models and observations on the model revision process, further studies can be made. Considering increasingly bigger networks would allow us to verify the scalability and limitation of our approach. We expect that the size of the network would greatly impact the approach performance (considering a fixed connectivity ratio of the networks). In fact, we expect that with very large networks, it would be either fast to repair it or it would take too much time. If the inconsistency is in a regulatory function of relatively small dimension, it should be fast to find a solution. However, if topological changes are necessary, considering the addition of missing regulators, with increasingly large networks there is a exponential explosion of the repair search space, not only with the combinations of possible additions, but also in the corresponding function space. If this model revision process is not able to repair such large networks, we could consider smaller portions of the network at a time, repairing them separately, considering a compositional approach. However, this may lead to non optimal solutions, or it could even be impossible to find a solution.

In the experimental evaluation presented, we consider complete time-series observations. We desire to conduct further studies considering incomplete observations, to understand the impact of the number of missing values in our model revision approach. We expect that, with an increase of the number of missing values, the time spent on the consistency check procedure will increase. This is due to the exponential growth of the number of possible combinations of assignments of the missing values. However, intuitively we expect that with an increasing number of missing values, the number of inconsistent nodes would decrease, making the search for repair operations faster. In the extreme end, where all the values are missing, any model would be consistent, as any dynamic behaviour is compatible with an empty observation.

We would also want to confront our model revision approach with bigger time-series observations. Here we considered observations with twenty time steps. This is particularly relevant in the asynchronous update scheme, where the current results show that our approach solves most of the instances under the defined time limit (3 600 seconds). Considering an increasing number of time steps will allow us to determine possible limitations of the proposed approach.

ModRev Tool

There are several tools in Systems Biology that operate over logical models. It is of great interest being able to combine the use of several tools. However, ModRev uses an input format based on ASP predicates. Although this is a simple format, it does not facilitate the

interoperability with other tools. To overcome this difficulty, we plan to integrate our format in the Logical Qualitative Modelling toolkit [108].

Moreover, it would be interesting to develop a graphical user interface (GUI) for the ModRev tool, since currently can only be used as a command line tool. This would facilitate the use of our tool by modellers or biological experts, that are not familiar with command line tools. Complementary, we could integrate it in the CoLoMoTo Interactive Notebook [109].

Final Thoughts

Although our model revision approach was designed for biological regulatory networks, it can be used in different contexts. In fact, the procedure was designed to repair inconsistent Boolean logical models, therefore it can be applied to different problems that can be represented as a Boolean logical model. However, in this work we restricted the domain of the Boolean functions to monotone non-degenerate Boolean functions, which may not be suitable in different contexts. Moreover, the repair operations and optimisation criterion defined take into account the construction process of biological regulatory networks. Under different contexts, it might be necessary to adapt the set of repair operations or the optimisation criterion. One could easily change the lexicographic order of the optimisation criterion, or even consider the same weight for every repair operation, to be more suitable to different model revision contexts.

Bibliography

[1] G. Karlebach and R. Shamir. Modelling and analysis of gene regulatory networks. *Nature Reviews Molecular Cell Biology*, 9(10):770–780, Sept. 2008. doi: 10.1038/nrm2503.

[2] H. de Jong. Modeling and simulation of genetic regulatory systems: A literature review. *Journal of Computational Biology*, 9(1):67–103, Jan. 2002. doi: 10.1089/10665270252833208.

[3] R. Thomas. Boolean formalization of genetic control circuits. *Journal of Theoretical Biology*, 42(3):563–585, Dec. 1973. doi: 10.1016/0022-5193(73)90247-6.

[4] A. Siegel, O. Radulescu, M. Le Borgne, P. Veber, J. Ouy, and S. Lagarrigue. Qualitative analysis of the relation between DNA microarray data and behavioral models of regulation networks. *Biosystems*, 84(2):153–174, 2006.

[5] A. Naldi, P. T. Monteiro, C. Mussel, H. A. Kestler, D. Thieffry, I. Xenarios, J. Saez-Rodriguez, T. Helikar, and C. C. and. Cooperative development of logical modelling standards and tools with CoLoMoTo. *Bioinformatics*, 31(7):1154–1159, Jan. 2015. doi: 10.1093/bioinformatics/btv013.

[6] W. Abou-Jaoudé, P. Traynard, P. T. Monteiro, J. Saez-Rodriguez, T. Helikar, D. Thieffry, and C. Chaouiya. Logical modeling and dynamical analysis of cellular networks. *Frontiers in Genetics*, 7, May 2016. doi: 10.3389/fgene.2016.00094.

[7] G. Selvaggio, S. Canato, A. Pawar, P. T. Monteiro, P. S. Guerreiro, M. M. Brás, F. Janody, and C. Chaouiya. Hybrid epithelial–mesenchymal phenotypes are controlled by microenvironmental factors. *Cancer Research*, 80(11):2407–2420, Mar. 2020. doi: 10.1158/0008-5472.can-19-3147.

[8] N. Friedman. Inferring cellular networks using probabilistic graphical models. *Science*, 303(5659):799–805, 2004.

[9] C. Guziolowski, S. Videla, F. Eduati, S. Thiele, T. Cokelaer, A. Siegel, and J. Saez-Rodriguez. Exhaustively characterizing feasible logic models of a signaling network using Answer Set Programming. *Bioinformatics*, 29(18):2320–2326, July 2013. doi: 10.1093/bioinformatics/btt393.

[10] M. Ostrowski, L. Paulevé, T. Schaub, A. Siegel, and C. Guziolowski. Boolean network identification from perturbation time series data combining dynamics abstraction and logic programming. *Biosystems*, 149:139–153, Nov. 2016. doi: 10.1016/j.biosystems.2016.07.009.

[11] S. Barman and Y.-K. Kwon. A Boolean network inference from time-series gene expression data using a genetic algorithm. *Bioinformatics*, 34(17):i927–i933, 09 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty584.

[12] S. Barman and Y.-K. Kwon. A neuro-evolution approach to infer a Boolean network from time-series gene expressions. *Bioinformatics*, 36(Supplement_2):i762–i769, 2020.

[13] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, Aug. 2012. doi: 10.1016/j.artint.2012.04.001.

[14] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Answer set solving in practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(3):1–238, Dec. 2012. doi: 10.2200/s00457ed1v01y201211aim019.

[15] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[16] M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber. Repair and prediction (under inconsistency) in large biological networks with Answer Set Programming. In *KR*, 2010.

[17] J. Guerra and I. Lynce. Reasoning over biological networks using maximum satisfiability. In *Lecture Notes in Computer Science*, pages 941–956. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-33558-7_67.

[18] C. E. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50(2):510–530, June 1985. doi: 10.2307/2274239.

[19] E. Merhej, S. Schockaert, and M. D. Cock. Repairing inconsistent answer set programs using rules of thumb: A gene regulatory networks case study. *International Journal of Approximate Reasoning*, 83:243–264, Apr. 2017. doi: 10.1016/j.ijar.2017.01.012.

[20] N. Mobilia, A. Rocca, S. Chorlton, E. Fanchon, and L. Trilling. Logical modeling and analysis of regulatory genetic networks in a non monotonic framework. In *Bioinformatics and Biomedical Engineering*, pages 599–612. Springer International Publishing, 2015. doi: 10.1007/978-3-319-16483-0_58.

[21] A. Lemos, I. Lynce, and P. T. Monteiro. Repairing Boolean logical models from time-series data using Answer Set Programming. *Algorithms for Molecular Biology*, 14(1), Mar. 2019. doi: 10.1186/s13015-019-0145-8.

[22] N. Friedman, M. Linial, I. Nachman, and D. Pe'er. Using Bayesian networks to analyze expression data. *Journal of Computational Biology*, 7(3-4):601–620, 2000.

[23] D. T. Gillespie. The chemical Langevin equation. *The Journal of Chemical Physics*, 113 (1):297–306, July 2000. doi: 10.1063/1.481811.

[24] S. Kauffman. Homeostasis and differentiation in random genetic control networks. *Nature*, 224(5215):177, 1969.

[25] I. Shmulevich, E. R. Dougherty, S. Kim, and W. Zhang. Probabilistic Boolean networks: a rule-based uncertainty model for gene regulatory networks. *Bioinformatics*, 18(2):261–274, Feb. 2002. doi: 10.1093/bioinformatics/18.2.261.

[26] I. Shmulevich, E. Dougherty, and W. Zhang. From Boolean to probabilistic Boolean networks as models of genetic regulatory networks. *Proceedings of the IEEE*, 90(11):1778–1792, Nov. 2002. doi: 10.1109/jproc.2002.804686.

[27] M. Gebser, T. Schaub, S. Thiele, B. Usadel, and P. Veber. Detecting inconsistencies in large biological networks with Answer Set Programming. In *Logic Programming*, pages 130–144. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-89982-2_19.

[28] L. Paulevé. Goal-oriented reduction of automata networks. In *Computational Methods in Systems Biology*, pages 252–272. Springer International Publishing, 2016. doi: 10.1007/978-3-319-45177-0_16.

[29] L. Pauleve. Reduction of qualitative models of biological networks for transient dynamics analysis. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 15(4): 1167–1179, July 2018. doi: 10.1109/tcbb.2017.2749225.

[30] L. Paulevé, J. Kolčák, T. Chatain, and S. Haar. Reconciling qualitative, abstract, and scalable modeling of biological networks. *Nature Communications*, 11(1), Aug. 2020. doi: 10.1038/s41467-020-18112-5.

[31] H. Klarner, A. Bockmayr, and H. Siebert. Computing maximal and minimal trap spaces of Boolean networks. *Natural Computing*, 14(4):535–544, Oct. 2015. doi: 10.1007/s11047-015-9520-7.

[32] A. Blake. Canonical expressions in Boolean algebra. In *PhD Thesis, Department of Mathematics*. University of Chicago, 1938.

[33] Y. Crama and P. L. Hammer. *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press, 2011.

[34] J. E. Cury, P. T. Monteiro, and C. Chaouiya. Partial order on the set of Boolean regulatory functions. *arXiv preprint arXiv:1901.07623*, 2019.

[35] I. Wegner. The critical complexity of all (monotone) Boolean functions and monotone graph properties. *Information and Control*, 67(1-3):212–222, Oct. 1985. doi: 10.1016/s0019-9958(85)80036-x.

[36] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge university press, 2003.

[37] V. Lifschitz. What is Answer Set Programming? In *Proceedings of the 23rd national conference on Artificial intelligence-Volume 3*, pages 1594–1597, 2008.

[38] V. Lifschitz. *Answer Set Programming*. Springer International Publishing, 2019. doi: 10.1007/978-3-030-24658-7.

[39] S. Chevalier, V. Noël, L. Calzone, A. Zinovyev, and L. Paulevé. Synthesis and simulation of ensembles of Boolean networks for cell fate decision. In *Computational Methods in Systems Biology*, pages 193–209. Springer International Publishing, 2020. doi: 10.1007/978-3-030-60327-4_11.

[40] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.

[41] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Clingo= asp+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*, 2014.

[42] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, July 2018. doi: 10.1017/s1471068418000054.

[43] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in gringo series 3. In *Logic Programming and Nonmonotonic Reasoning*, pages 345–351. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-20895-9_39.

[44] R.-S. Wang, A. Saadatpour, and R. Albert. Boolean modeling in systems biology: an overview of methodology and applications. *Physical Biology*, 9(5):055001, Sept. 2012. doi: 10.1088/1478-3975/9/5/055001.

[45] A. Montagud, P. Traynard, L. Martignetti, E. Bonnet, E. Barillot, A. Zinovyev, and L. Calzone. Conceptual and computational framework for logical modelling of biological networks deregulated in diseases. *Briefings in Bioinformatics*, 20(4):1238–1249, Dec. 2017. doi: 10.1093/bib/bbx163.

[46] C. Chaouiya, A. Naldi, and D. Thieffry. Logical modelling of gene regulatory networks with GINsim. In *Bacterial Molecular Networks*, pages 463–479. Springer New York, Oct. 2011. doi: 10.1007/978-1-61779-361-5_23.

[47] G. Stoll, E. Viara, E. Barillot, and L. Calzone. Continuous time Boolean modeling for biological signaling: application of Gillespie algorithm. *BMC Systems Biology*, 6(1):116, 2012. doi: 10.1186/1752-0509-6-116.

[48] G. Stoll, B. Caron, E. Viara, A. Dugourd, A. Zinovyev, A. Naldi, G. Kroemer, E. Barillot, and L. Calzone. MaBoSS 2.0: an environment for stochastic Boolean modeling. *Bioinformatics*, 33(14):2226–2228, Mar. 2017. doi: 10.1093/bioinformatics/btx123.

[49] A. Zinovyev, E. Viara, L. Calzone, and E. Barillot. BiNoM: a cytoscape plugin for manipulating and analyzing biological networks. *Bioinformatics*, 24(6):876–877, Nov. 2007. doi: 10.1093/bioinformatics/btm553.

[50] E. Bonnet, L. Calzone, D. Rovera, G. Stoll, E. Barillot, and A. Zinovyev. BiNoM 2.0, a cytoscape plugin for accessing and analyzing pathways using standard systems biology formats. *BMC Systems Biology*, 7(1):18, 2013. doi: 10.1186/1752-0509-7-18.

[51] C. Müssel, M. Hopfensitz, and H. A. Kestler. BoolNet – an R package for generation, reconstruction and analysis of Boolean networks. *Bioinformatics*, 26(10):1378–1380, 2010.

[52] C. Terfve, T. Cokelaer, D. Henriques, A. MacNamara, E. Goncalves, M. K. Morris, M. van Iersel, D. A. Lauffenburger, and J. Saez-Rodriguez. CellNOptR: a flexible toolkit to train protein signaling networks to data using multiple logic formalisms. *BMC Systems Biology*, 6(1):133, 2012. doi: 10.1186/1752-0509-6-133.

[53] P. L. Varela, C. V. Ramos, P. T. Monteiro, and C. Chaouiya. EpiLog: A software for the logical modelling of epithelial dynamics. *F1000Research*, 7, 2018.

[54] A. J. Butte and I. S. Kohane. Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements. In *Biocomputing 2000*, pages 418–429. World Scientific, 1999. doi: 10.1142/9789814447331_0040.

[55] A. A. Margolin, I. Nemenman, K. Basso, C. Wiggins, G. Stolovitzky, R. Dalla Favera, and A. Califano. ARACNE: an algorithm for the reconstruction of gene regulatory networks in a mammalian cellular context. In *BMC bioinformatics*, volume 7. Springer, 2006.

[56] A. Lachmann, F. M. Giorgi, G. Lopez, and A. Califano. ARACNe-AP: gene network reverse engineering through adaptive partitioning inference of mutual information. *Bioinformatics*, 32(14):2233–2235, 2016.

[57] P. E. Meyer, K. Kontos, F. Lafitte, and G. Bontempi. Information-theoretic inference of large transcriptional regulatory networks. *EURASIP journal on bioinformatics and systems biology*, 2007:1–9, 2007.

[58] K. Basso, A. A. Margolin, G. Stolovitzky, U. Klein, R. Dalla-Favera, and A. Califano. Reverse engineering of regulatory networks in human B cells. *Nature genetics*, 37(4): 382–390, 2005.

[59] W. Luo, K. D. Hankenson, and P. J. Woolf. Learning transcriptional regulatory networks from high throughput gene expression data using continuous three-way mutual information. *BMC bioinformatics*, 9(1):467, 2008.

[60] G. D. Tourassi, E. D. Frederick, M. K. Markey, and C. E. Floyd Jr. Application of the mutual information criterion for feature selection in computer-aided diagnosis. *Medical physics*, 28(12):2394–2402, 2001.

[61] G. Altay and F. Emmert-Streib. Inferring the conservative causal core of gene regulatory networks. *BMC systems biology*, 4(1):132, 2010.

[62] R. Bonneau, D. J. Reiss, P. Shannon, M. Facciotti, L. Hood, N. S. Baliga, and V. Thorsson. The Inferelator: an algorithm for learning parsimonious regulatory networks from systems-biology data sets de novo. *Genome biology*, 7(5):R36, 2006.

[63] D. Koczan, S. Drynda, M. Hecker, A. Drynda, R. Guthke, J. Kekow, and H.-J. Thiesen. Molecular discrimination of responders and nonresponders to anti-TNFalpha therapy in rheumatoid arthritis by etanercept. *Arthritis research & therapy*, 10(3):R50, 2008.

[64] M. G. Rabbat, M. A. Figueiredo, and R. D. Nowak. Network inference from co-occurrences. *IEEE Transactions on Information Theory*, 54(9):4053–4068, 2008.

[65] I. M. Ong, J. D. Glasner, and D. Page. Modelling regulatory pathways in E. coli from time series expression profiles. *Bioinformatics*, 18(suppl_1):S241–S248, 2002.

[66] J. Yu, V. A. Smith, P. P. Wang, A. J. Hartemink, and E. D. Jarvis. Advances to Bayesian network inference for generating causal networks from observational biological data. *Bioinformatics*, 20(18):3594–3603, 2004.

[67] A. J. Hartemink, D. K. Gifford, T. S. Jaakkola, and R. A. Young. Using graphical models and genomic expression data to statistically validate models of genetic regulatory networks. In *Biocomputing 2001*, pages 422–433. World Scientific, 2000.

[68] M. I. Davidich and S. Bornholdt. Boolean network model predicts cell cycle sequence of fission yeast. *PLoS ONE*, 3(2):e1672, Feb. 2008. doi: 10.1371/journal.pone.0001672.

[69] T. Fayruzov, J. Janssen, D. Vermeir, C. Cornelis, and M. D. Cock. Modelling gene and protein regulatory networks with Answer Set Programming. *International Journal of Data Mining and Bioinformatics*, 5(2):209, 2011. doi: 10.1504/ijdmb.2011.039178.

[70] S. Videla, C. Guziolowski, F. Eduati, S. Thiele, N. Grabe, J. Saez-Rodriguez, and A. Siegel. Revisiting the training of logic models of protein signaling networks with ASP. In *Computational Methods in Systems Biology*, pages 342–361. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-33636-2_20.

[71] T. Schaub, A. Siegel, and S. Videla. Reasoning on the response of logical signaling networks with ASP. In *Logical Modeling of Biological Systems*, pages 49–92. John Wiley & Sons, Inc., Aug. 2014. doi: 10.1002/9781119005223.ch2.

[72] C. Réda and B. Wilczyński. Automated inference of gene regulatory networks using explicit regulatory modules. *Journal of Theoretical Biology*, 486:110091, Feb. 2020. doi: 10.1016/j.jtbi.2019.110091.

[73] A. Mitsos, I. N. Melas, P. Siminelakis, A. D. Chairakaki, J. Saez-Rodriguez, and L. G. Alexopoulos. Identifying drug effects via pathway alterations using an integer linear programming optimization formulation on phosphoproteomic data. *PLoS Computational Biology*, 5(12):e1000591, Dec. 2009. doi: 10.1371/journal.pcbi.1000591.

[74] R. Sharan and R. M. Karp. Reconstructing Boolean models of signaling. *Journal of Computational Biology*, 20(3):249–257, Mar. 2013. doi: 10.1089/cmb.2012.0241.

[75] T. Tamura and T. Akutsu. Detecting a singleton attractor in a Boolean network utilizing SAT algorithms. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E92-A(2):493–501, 2009. doi: 10.1587/transfun.e92.a.493.

[76] A. A. Melkman, T. Tamura, and T. Akutsu. Determining a singleton attractor of an AND/OR Boolean network in time. *Information Processing Letters*, 110(14-15):565–569, July 2010. doi: 10.1016/j.ipl.2010.05.001.

[77] A. Garg, I. Xenarios, L. Mendoza, and G. DeMicheli. An efficient method for dynamic analysis of gene regulatory networks and in silico gene perturbation experiments. In *Research in Computational Molecular Biology*, pages 62–76. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-71681-5_5.

[78] A. Naldi, D. Thieffry, and C. Chaouiya. Decision diagrams for the representation and analysis of logical models of genetic networks. In *Computational Methods in Systems Biology*, pages 233–247. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-75140-3_16.

[79] A. Garg, A. D. Cara, I. Xenarios, L. Mendoza, and G. D. Micheli. Synchronous versus asynchronous modeling of gene regulatory networks. *Bioinformatics*, 24(17):1917–1925, July 2008. doi: 10.1093/bioinformatics/btn336.

[80] A. Tiwari, C. Talcott, M. Knapp, P. Lincoln, and K. Laderoute. Analyzing pathways using SAT-based approaches. In *Algebraic Biology*, pages 155–169. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-73433-8_12.

[81] O. Ray, T. Soh, and K. Inoue. Analyzing pathways using ASP-based approaches. In *Algebraic and Numeric Biology*, pages 167–183. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-28067-2_10.

[82] E. Dubrova and M. Teslenko. A SAT-based algorithm for finding attractors in synchronous

Boolean networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(5):1393–1399, Sept. 2011. doi: 10.1109/tcbb.2010.20.

[83] T. Khaled and B. Benhamou. An ASP-based approach for attractor enumeration in synchronous and asynchronous Boolean networks. *Electronic Proceedings in Theoretical Computer Science*, 306:295–301, Sept. 2019. doi: 10.4204/eptcs.306.34.

[84] T. Khaled and B. Benhamou. An ASP-based approach for Boolean networks representation and attractor detection. In *LPAR*, pages 317–333, 2020.

[85] E. Remy, B. Mosse, C. Chaouiya, and D. Thieffry. A description of dynamical graphs associated to elementary regulatory circuits. *Bioinformatics*, 19(Suppl 2):ii172–ii178, Sept. 2003. doi: 10.1093/bioinformatics/btg1075.

[86] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992. doi: 10.1016/0890-5401(92)90017-a.

[87] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[88] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 2018.

[89] P. Monteiro, W. Abou-Jaoudé, D. Thieffry, and C. Chaouiya. Model checking logical regulatory networks. *IFAC Proceedings Volumes*, 47(2):170–175, 2014. doi: 10.3182/20140514-3-fr-4046.00135.

[90] C. Baral, K. Chancellor, N. Tran, N. Tran, A. Joy, and M. Berens. A knowledge based approach for representing and reasoning about signaling networks. *Bioinformatics*, 20 (Suppl 1):i15–i22, July 2004. doi: 10.1093/bioinformatics/bth918.

[91] L. Fippo Fitime, O. Roux, C. Guziolowski, and L. Paulevé. Identification of Bifurcations in Biological Regulatory Networks using Answer-Set Programming. In *Constraint-Based Methods for Bioinformatics Workshop*, Toulouse, France, Sept. 2016.

[92] L. F. Fitime, O. Roux, C. Guziolowski, and L. Paulevé. Identification of bifurcation transitions in biological regulatory networks using Answer-Set Programming. *Algorithms for Molecular Biology*, 12(1), July 2017. doi: 10.1186/s13015-017-0110-3.

[93] A. Naldi, E. Remy, D. Thieffry, and C. Chaouiya. Dynamically consistent reduction of logical regulatory graphs. *Theoretical Computer Science*, 412(21):2207–2218, May 2011. doi: 10.1016/j.tcs.2010.10.021.

[94] J. G. T. Zañudo and R. Albert. An effective network reduction approach to find the dynamical repertoire of discrete dynamic networks. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 23(2):025111, June 2013. doi: 10.1063/1.4809777.

[95] H. Klarner, A. Bockmayr, and H. Siebert. Computing symbolic steady states of Boolean networks. In *Lecture Notes in Computer Science*, pages 561–570. Springer International Publishing, 2014. doi: 10.1007/978-3-319-11520-7_59.

[96] A. D. Korshunov. Monotone Boolean functions. *Russian Mathematical Surveys*, 58(5): 929–1001, Oct. 2003. doi: 10.1070/rm2003v058n05abeh000667.

[97] T. Stephen and T. Yusun. Counting inequivalent monotone Boolean functions. *Discrete Applied Mathematics*, 167:15–24, Apr. 2014. doi: 10.1016/j.dam.2013.11.015.

[98] L. Sanchez, C. Chaouiya, and D. Thieffry. Segmenting the fly embryo: logical analysis of the role of the segment polarity cross-regulatory module. *The International Journal of Developmental Biology*, 52(8):1059–1075, 2008. doi: 10.1387/ijdb.072439ls.

[99] S. Klamt, J. Saez-Rodriguez, J. A. Lindquist, L. Simeoni, and E. D. Gilles. A methodology for the structural and functional analysis of signaling and regulatory networks. *BMC Bioinformatics*, 7(1):56, 2006. doi: 10.1186/1471-2105-7-56.

[100] A. Faure, A. Naldi, C. Chaouiya, and D. Thieffry. Dynamical analysis of a generic Boolean model for the control of the mammalian cell cycle. *Bioinformatics*, 22(14):e124–e131, July 2006. doi: 10.1093/bioinformatics/btl210.

[101] L. Mendoza and I. Xenarios. A method for the generation of standardized qualitative dynamical systems of regulatory networks. *Theoretical Biology and Medical Modelling*, 3 (1):13, 2006. doi: 10.1186/1742-4682-3-13.

[102] F. Gouveia, I. Lynce, and P. T. Monteiro. Model revision of logical regulatory networks using logic-based tools. In *Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2018. doi: 10.4230/OASICS.ICLP.2018.23.

[103] F. Gouveia, I. Lynce, and P. T. Monteiro. Model revision of Boolean regulatory networks at stable state. In *Bioinformatics Research and Applications*, pages 100–112. Springer International Publishing, 2019. doi: 10.1007/978-3-030-20242-2_9.

[104] F. Gouveia, I. Lynce, and P. T. Monteiro. Revision of Boolean models of regulatory networks using stable state observations. *Journal of Computational Biology*, 27(2):144–155, Feb. 2020. doi: 10.1089/cmb.2019.0289.

[105] F. Gouveia, I. Lynce, and P. T. Monteiro. Semi-automatic model revision of Boolean regulatory networks: confronting time-series observations with (a)synchronous dynamics. *bioRxiv*, May 2020. doi: 10.1101/2020.05.10.086900.

[106] F. Gouveia, I. Lynce, and P. T. Monteiro. ModRev - model revision tool for Boolean logical models of biological regulatory networks. In *Computational Methods in Systems Biology*, pages 339–348. Springer International Publishing, 2020. doi: 10.1007/978-3-030-60327-4_18.

[107] K. Inoue. Logic programming for Boolean networks. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[108] A. Naldi. BioLQM: A java toolkit for the manipulation and conversion of logical qualitative models of biological networks. *Frontiers in Physiology*, 9, Nov. 2018. doi: 10.3389/fphys.2018.01605.

[109] A. Naldi, C. Hernandez, N. Levy, G. Stoll, P. T. Monteiro, C. Chaouiya, T. Helikar, A. Zinovyev, L. Calzone, S. Cohen-Boulakia, D. Thieffry, and L. Paulevé. The CoLoMoTo interactive notebook: Accessible and reproducible computational analyses for qualitative biological networks. *Frontiers in Physiology*, 9, June 2018. doi: 10.3389/fphys.2018.00680.

Appendix A

Results

Here is presented detailed results of the evaluation method described in Chapter 6.

Table A.1 shows the results obtained when the proposed model revision approach is evaluated under stable state observations considering a time limit of 3 600 seconds. For the search of function repair, it is considered the C++ implementation discussed in Section 4.2. These results are very similar to the results shown in Table 6.4 that corresponds to the same evaluation with a lower time limit (600 seconds). In fact, increasing the time limit by a factor of six only produces an increase of 7% of solved instances.

Here, we also present detailed results when corrupted models are confronted with time-series observations under synchronous and asynchronous update scheme. Table A.2 shows the results obtained for each model under different corruption configurations, when confronted with one time-series observation with three time steps under synchronous update scheme, and for the asynchronous update scheme the results are shown in Table A.3. We increased the number of time steps from three to twenty, and the results are shown in Table A.4 for the synchronous update scheme, and in Table A.5 for the asynchronous update scheme.

Experiments are made considering five time-series observations simultaneously. Table A.6 shows the results for five observations with three time steps under the synchronous update scheme (Table A.7 for the asynchronous case). Then, the results for five observations when the number of time steps increases from three to twenty are shown in Tables A.8 and A.8 considering synchronous and asynchronous update scheme, respectively.

Results show that when the original models are corrupted with topological changes, the solving time and number of unsolved instances increases. Moreover, when the number of observations or the size of the observations increases, the solving time also increases.

Considering the same number of observations and time steps, the results for the asynchronous update scheme are better than the results for the synchronous update scheme.

Table A.1: Results for model revision of FY, SP, TCR, MCC, and Th, under stable state observations using C++ for function repair search. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 3600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,021	0,021	0	0,022	0,022	0	0,032	0,033	0	0,010	0,011	0	0,014	0,014	0
25	0	0	0	0,021	0,022	0	0,022	0,023	0	0,031	0,032	0	0,011	0,011	0	0,014	0,014	0
50	0	0	0	0,021	0,022	0	0,023	0,024	0	0,033	0,034	0	0,011	0,010	0	0,014	0,014	0
100	0	0	0	0,021	0,021	0	0,026	0,027	0	0,034	0,035	0	0,010	0,009	0	0,015	0,015	0
0	5	0	0	47,043	0,023	0	2,368	0,030	1	0,032	0,033	0	0,011	0,008	0	58,700	0,016	0
0	10	0	0	139,205	0,027	0	10,703	0,195	6	0,033	0,034	0	0,019	0,012	0	94,927	0,015	0
0	15	0	0	222,487	0,033	0	71,062	0,306	7	0,034	0,034	0	0,256	0,013	0	134,074	0,019	0
0	20	0	0	326,316	4,652	0	93,628	1,072	12	0,034	0,035	0	6,141	0,015	0	171,773	0,026	0
0	25	0	0	407,593	141,571	0	137,763	1,960	15	0,035	0,035	0	6,566	0,019	0	243,159	0,273	0
0	50	0	0	954,266	721,056	0	544,176	224,736	50	0,038	0,038	0	44,685	1,121	0	297,434	32,711	0
0	75	0	0	1 517, 655	1 405, 030	0	1 089, 124	803,081	85	0,040	0,041	0	458,875	208,973	0	134,983	0,992	0
0	0	1	0	0,026	0,023	6	0,179	0,023	0	0,041	0,033	0	0,010	0,009	0	0,019	0,015	3
0	0	5	0	0,243	0,023	13	0,435	0,030	0	0,065	0,034	0	0,010	0,009	0	0,122	0,017	7
0	0	10	0	0,243	0,023	17	0,711	0,165	0	0,067	0,036	0	0,010	0,008	0	0,221	0,020	10
0	0	15	0	0,742	0,024	18	0,759	0,129	0	0,085	0,055	0	0,010	0,009	0	0,266	0,020	11
0	0	0	1	0,043	0,024	0	21,869	0,035	10	0,053	0,036	0	0,010	0,009	0	2,702	0,017	2
0	0	0	5	7,533	0,046	0	115,744	16,053	33	46,981	0,977	28	3,559	0,012	0	27,395	1,375	40
0	0	0	10	36,869	0,148	5	638,044	133,642	80	573,929	197,227	83	3,919	0,016	0	382,495	71,815	65
0	0	0	15	118,670	0,227	7	710,902	710,902	98	-	-	100	6,247	0,020	0	1 269, 273	263,372	95
25	5	0	0	68,623	0,026	0	2,913	0,029	1	0,034	0,034	0	0,011	0,010	0	68,511	0,018	0
50	25	0	0	282,054	0,786	0	157,226	2,582	21	0,037	0,037	0	4,807	0,022	0	105,260	0,024	0
100	50	0	0	893,875	603,361	1	381,256	53,331	43	0,044	0,044	0	59,727	2,689	0	192,706	32,738	0
5	25	5	5	402,601	0,456	19	1 174, 075	454,305	73	563,139	1,004	51	38,369	0,035	4	313,578	2,456	37
10	10	5	5	57,412	0,047	13	619,409	31,409	49	368,948	12,989	25	0,839	0,016	2	209,292	1,561	36

Table A.2: Results for model revision of FY, SP, TCR, MCC, and Th, confronted with one time-series observation with three time steps under synchronous update scheme. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 3 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,018	0,017	0	0,020	0,020	0	0,028	0,027	0	0,017	0,017	0	0,019	0,019	0
25	0	0	0	0,017	0,017	0	0,019	0,019	0	0,027	0,027	0	0,015	0,015	0	0,020	0,020	0
50	0	0	0	0,015	0,015	0	0,020	0,020	0	0,027	0,027	0	0,015	0,015	0	0,019	0,018	0
100	0	0	0	0,014	0,014	0	0,024	0,023	0	0,028	0,029	0	0,015	0,016	0	0,022	0,022	0
0	5	0	0	0,016	0,015	0	3,528	0,020	8	0,025	0,025	0	0,023	0,016	0	52,331	0,019	0
0	10	0	0	32,696	0,016	1	12,514	0,022	17	0,027	0,026	0	0,028	0,016	0	118,095	0,020	0
0	15	0	0	204,054	0,018	4	22,451	0,025	22	0,027	0,027	0	0,033	0,017	0	195,096	0,019	0
0	20	0	0	270,141	0,019	4	49,032	0,034	27	0,028	0,028	0	0,042	0,020	0	232,137	0,020	0
0	25	0	0	413,535	0,020	6	52,541	0,248	34	0,028	0,027	0	0,045	0,021	0	253,376	0,020	0
0	50	0	0	1 030, 686	1,157	22	99,348	30,521	57	0,029	0,029	0	0,069	0,053	0	310,312	51,220	0
0	75	0	0	1 553, 580	38,097	51	142,784	131,089	78	0,030	0,029	0	0,115	0,110	0	193,844	71,303	0
0	0	1	0	121,292	0,016	0	45,325	0,020	5	0,027	0,026	0	0,115	0,015	0	0,025	0,021	4
0	0	5	0	87,129	0,014	0	100,318	0,023	16	0,044	0,032	0	0,130	0,016	0	0,163	0,022	6
0	0	10	0	163,747	0,016	0	138,886	0,040	25	0,105	0,048	0	0,134	0,019	0	0,475	0,033	6
0	0	15	0	198,759	0,015	0	152,480	0,196	22	0,162	0,068	0	0,261	0,030	0	0,849	0,049	8
0	0	0	1	0,015	0,014	0	1,862	0,021	15	0,029	0,027	0	10,838	0,016	5	7,401	0,019	9
0	0	0	5	0,047	0,015	0	21,484	0,208	51	36,112	1,721	43	75,168	0,026	14	43,346	0,166	54
0	0	0	10	4,800	0,018	0	71,403	23,909	86	1347,743	569,305	97	45,886	0,044	29	159,511	21,006	88
0	0	0	15	9,484	0,028	0	388,199	382,488	96			100	128,785	1,348	46	1 097, 470	1 097, 470	99
25	5	0	0	21,222	0,016	0	20,986	0,021	9	0,025	0,025	0	0,026	0,018	0	29,325	0,020	0
50	25	0	0	374,014	0,023	3	62,301	0,086	33	0,029	0,028	0	0,049	0,023	0	87,167	0,021	0
100	50	0	0	1 220, 268	0,883	1	127,912	38,497	61	0,032	0,031	0	0,135	0,097	0	125,006	0,027	0
5	25	5	5	388,430	0,042	11	520,266	42,913	69	460,220	9,251	35	130,099	0,078	16	363,380	20,947	51
10	10	5	5	184,262	0,019	2	311,957	15,437	59	248,863	4,322	24	88,873	0,041	18	229,340	0,822	58

Table A.3: Results for model revision of FY, SP, TCR, MCC, and Th, confronted with one time-series observation with three time steps under asynchronous update scheme. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 3 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,022	0,022	0	0,030	0,031	0	0,040	0,040	0	0,023	0,024	0	0,032	0,032	0
25	0	0	0	0,022	0,023	0	0,032	0,033	0	0,040	0,040	0	0,024	0,024	0	0,029	0,029	0
50	0	0	0	0,022	0,022	0	0,033	0,031	0	0,039	0,038	0	0,024	0,024	0	0,029	0,030	0
100	0	0	0	0,021	0,020	0	0,034	0,033	0	0,040	0,039	0	0,024	0,024	0	0,029	0,030	0
0	5	0	0	0,022	0,022	0	0,031	0,029	0	0,041	0,040	0	0,023	0,230	0	0,029	0,029	0
0	10	0	0	0,043	0,024	0	0,029	0,028	0	0,041	0,040	0	0,024	0,024	0	0,029	0,029	0
0	15	0	0	0,131	0,023	0	0,031	0,030	0	0,044	0,042	0	0,024	0,024	0	0,029	0,029	0
0	20	0	0	0,891	0,025	0	0,031	0,031	0	0,043	0,041	0	0,023	0,024	0	0,029	0,029	0
0	25	0	0	1,316	0,025	0	0,029	0,029	0	0,043	0,040	0	0,024	0,025	0	0,029	0,030	0
0	50	0	0	33,012	1,296	0	0,032	0,032	0	0,043	0,040	0	0,024	0,025	0	0,028	0,028	0
0	75	0	0	238,724	33,771	0	0,033	0,033	0	0,039	0,038	0	0,024	0,023	0	0,029	0,028	0
0	0	1	0	0,022	0,022	0	0,031	0,031	0	0,039	0,040	0	0,022	0,023	0	0,028	0,028	0
0	0	5	0	0,021	0,021	0	0,030	0,030	0	0,039	0,039	0	0,022	0,023	0	0,027	0,027	0
0	0	10	0	0,020	0,020	0	0,029	0,030	0	0,036	0,036	0	0,022	0,022	0	0,028	0,027	0
0	0	15	0	0,020	0,020	0	0,029	0,028	0	0,037	0,036	0	0,021	0,022	0	0,027	0,027	0
0	0	0	1	0,023	0,023	0	0,033	0,033	0	0,044	0,044	0	0,024	0,024	0	0,027	0,026	0
0	0	0	5	0,067	0,024	0	0,036	0,037	0	2,538	0,054	4	0,025	0,024	0	0,343	0,030	0
0	0	0	10	3,113	0,025	1	0,039	0,037	0	25,248	0,098	15	0,057	0,024	0	3,417	0,034	3
0	0	0	15	6,743	0,031	1	0,042	0,041	0	175,381	1,486	61	0,399	0,025	1	28,781	0,040	12
25	5	0	0	0,023	0,023	0	0,031	0,032	0	0,040	0,040	0	0,023	0,024	0	0,028	0,028	0
50	25	0	0	0,963	0,024	0	0,032	0,031	0	0,042	0,041	0	0,023	0,024	0	0,027	0,028	0
100	50	0	0	101,831	0,059	0	0,033	0,033	0	0,042	0,041	0	0,024	0,024	0	0,032	0,032	0
5	25	5	5	11,193	0,038	1	0,033	0,033	0	1,573	0,055	2	0,023	0,024	0	0,034	0,031	1
10	10	5	5	2,813	0,025	0	0,032	0,033	0	0,965	0,052	1	0,023	0,024	0	0,034	0,030	1

Table A.4: Results for model revision of FY, SP, TCR, MCC, and Th, confronted with one time-series observation with twenty time steps under synchronous update scheme. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 3 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,039	0,038	0	0,053	0,053	0	0,077	0,077	0	0,041	0,041	0	0,048	0,048	0
25	0	0	0	0,039	0,038	0	0,058	0,055	0	0,087	0,084	0	0,046	0,045	0	0,052	0,052	0
50	0	0	0	0,040	0,039	0	0,060	0,058	0	0,089	0,086	0	0,046	0,044	0	0,056	0,053	0
100	0	0	0	0,037	0,036	0	0,062	0,061	0	0,088	0,082	0	0,047	0,047	0	0,052	0,052	0
0	5	0	0	96,888	0,043	0	4,004	0,056	8	0,088	0,082	0	0,529	0,048	17	39,995	0,050	0
0	10	0	0	163,402	0,047	1	16,397	0,058	17	0,089	0,081	0	1,605	0,056	22	89,338	0,050	0
0	15	0	0	416,589	0,046	6	31,769	0,072	22	0,078	0,078	0	2,303	0,119	33	146,347	0,052	0
0	20	0	0	615,862	0,050	8	61,152	0,117	27	0,080	0,079	0	3,741	0,256	45	166,272	0,051	0
0	25	0	0	886,890	0,066	7	69,062	0,496	34	0,079	0,079	0	6,952	0,596	55	177,135	0,055	0
0	50	0	0	1 819, 381	2 794, 790	33	148,705	89,330	57	0,085	0,081	0	143,000	5,894	89	174,941	0,244	0
0	75	0	0	2 100, 315	2 895, 305	72	164,466	135,169	78	0,097	0,091	0	-	-	100	77,679	0,406	0
0	0	1	0	75,810	0,040	4	43,729	0,053	5	0,086	0,087	0	23,074	0,045	9	0,062	0,055	4
0	0	5	0	78,203	0,039	7	99,426	0,077	16	0,132	0,101	0	104,979	0,150	20	0,401	0,058	6
0	0	10	0	132,960	0,043	9	141,165	0,121	25	0,403	0,159	0	136,136	0,371	25	1,029	0,076	6
0	0	15	0	81,180	0,047	10	150,323	0,270	22	0,525	0,196	2	182,249	0,429	33	2,426	0,098	8
0	0	0	1	0,042	0,041	0	1,833	0,054	15	0,121	0,114	0	8,397	0,042	23	8,005	0,056	10
0	0	0	5	0,495	0,042	1	29,533	0,254	51	53,774	2,330	46	56,059	0,061	38	48,181	0,147	61
0	0	0	10	2,171	0,050	1	78,696	25,807	86	1 393, 524	687,870	97	75,595	42,328	60	109,368	10,559	91
0	0	0	15	4,547	0,098	5	382,194	264,173	97	-	-	100	77,182	43,474	81	-	-	100
25	5	0	0	70,675	0,042	0	26,203	0,073	9	0,095	0,103	0	1,609	0,049	19	26,555	0,051	0
50	25	0	0	817,436	0,070	5	94,040	0,259	33	0,099	0,105	0	8,858	0,334	58	66,763	0,054	0
100	50	0	0	1 399, 972	1 336, 210	25	154,106	51,303	62	0,088	0,088	0	73,219	17,333	93	63,983	0,073	0
5	25	5	5	842,780	0,263	21	471,031	54,254	71	637,716	18,367	49	361,661	1,888	84	273,179	14,663	63
10	10	5	5	352,805	0,052	7	315,138	16,055	62	316,313	4,493	31	232,008	0,366	63	211,942	2,191	62

Table A.5: Results for model revision of FY, SP, TCR, MCC, and Th, confronted with one time-series observation with twenty time steps under asynchronous update scheme. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 3 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,596	0,566	0	0,704	0,705	0	0,875	0,852	0	0,529	0,528	0	0,489	0,483	0
25	0	0	0	0,590	0,575	0	0,739	0,736	0	0,951	0,995	0	0,516	0,523	0	0,502	0,497	0
50	0	0	0	0,567	0,551	0	0,726	0,717	0	0,979	0,984	0	0,606	0,613	0	0,492	0,490	0
100	0	0	0	0,482	0,471	0	0,777	0,808	0	0,854	0,850	0	0,590	0,594	0	0,504	0,504	0
0	5	0	0	98,951	0,544	0	0,840	0,706	0	0,875	0,873	0	0,615	0,619	0	39,450	0,526	0
0	10	0	0	132,101	0,536	1	5,043	0,708	0	0,876	0,873	0	0,621	0,611	0	71,228	0,497	0
0	15	0	0	286,754	0,546	3	8,562	0,720	0	0,825	0,824	0	0,626	0,644	0	107,518	0,536	0
0	20	0	0	478,539	0,600	4	15,125	0,837	3	0,984	0,986	0	0,548	0,525	0	134,129	0,497	0
0	25	0	0	526,535	0,621	4	21,625	0,843	6	0,949	0,994	0	0,602	0,573	0	138,431	0,501	0
0	50	0	0	895,868	1,852	23	39,714	0,982	13	0,903	0,859	0	0,615	0,581	0	131,784	0,759	0
0	75	0	0	1272,959	40,636	49	66,461	0,954	15	0,823	0,821	0	0,772	0,816	0	100,395	34,452	0
0	0	1	0	0,539	0,549	2	0,693	0,693	0	0,934	0,950	0	0,664	0,498	0	0,490	0,481	0
0	0	5	0	0,475	0,431	5	0,806	0,812	0	0,860	0,846	0	0,760	0,572	0	0,579	0,578	0
0	0	10	0	0,470	0,407	5	0,666	0,644	0	0,840	0,850	0	0,638	0,419	0	0,493	0,439	0
0	0	15	0	0,575	0,387	4	0,719	0,736	0	0,712	0,732	0	0,778	0,402	0	0,466	0,413	0
0	0	0	1	0,592	0,614	0	128,333	0,736	7	1,166	1,174	0	0,632	0,634	0	1,318	0,544	9
0	0	0	5	0,986	0,540	0	250,241	0,922	24	75,605	1,732	11	0,634	0,634	0	166,914	1,770	34
0	0	0	10	1,123	0,531	2	118,114	24,939	55	240,509	35,419	68	0,826	0,628	0	258,479	43,245	69
0	0	0	15	2,162	0,559	3	426,691	25,721	65	1829,561	2213,210	97	1,925	0,583	3	537,781	282,766	91
25	5	0	0	56,990	0,515	0	2,282	0,716	1	0,856	0,852	0	0,581	0,559	0	19,590	0,497	0
50	25	0	0	507,544	0,516	3	24,876	0,779	10	0,831	0,831	0	0,604	0,589	0	52,811	0,499	0
100	50	0	0	1258,254	8,303	1	34,464	0,811	25	1,021	1,024	0	0,594	0,573	0	60,583	0,586	0
5	25	5	5	497,517	0,513	11	21,195	0,803	25	223,903	1,700	17	95,159	0,534	5	208,960	1,662	34
10	10	5	5	204,649	0,478	4	121,493	0,816	23	20,126	1,542	11	27,923	0,534	4	254,194	2,014	33

Table A.6: Results for model revision of FY, SP, TCR, MCC, and Th, confronted with five time-series observations with three time steps under synchronous update scheme. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 3 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,034	0,034	0	0,047	0,043	0	0,079	0,075	0	0,036	0,034	0	0,539	0,049	0
25	0	0	0	0,036	0,036	0	0,060	0,053	0	0,082	0,076	0	0,035	0,034	0	1,124	0,052	0
50	0	0	0	0,038	0,038	0	0,061	0,052	0	0,092	0,087	0	0,039	0,039	0	1,523	0,049	0
100	0	0	0	0,034	0,034	0	0,496	0,062	0	0,095	0,088	0	0,041	0,039	0	3,984	0,051	0
0	5	0	0	149,036	0,052	0	6,982	0,052	8	47,528	0,082	0	277,263	0,047	0	3,961	0,052	0
0	10	0	0	250,899	0,155	2	23,743	0,070	19	68,926	0,089	0	401,233	0,149	0	7,538	0,054	0
0	15	0	0	609,957	233,752	5	34,089	0,294	25	155,237	0,083	0	587,550	22,387	2	11,938	0,122	0
0	20	0	0	784,269	466,990	9	31,369	0,494	36	254,787	0,086	0	659,563	441,555	2	14,479	0,140	0
0	25	0	0	1 004, 311	468,244	9	40,917	0,515	44	273,838	0,104	0	769,427	484,609	5	13,759	7,217	0
0	50	0	0	1 712, 721	1 269, 560	39	84,042	86,718	81	551,012	1,178	0	1 359, 549	1 261, 060	22	31,704	29,349	0
0	75	0	0	2 072, 776	2 505, 265	76	790,819	193,801	90	807,193	667,769	0	2 324, 141	2 616, 250	65	49,879	51,357	0
0	0	1	0	290,335	0,048	12	44,465	0,054	7	0,097	0,085	3	358,055	0,042	6	0,391	0,053	18
0	0	5	0	282,751	0,068	25	60,434	0,127	28	2,515	0,147	6	537,688	0,571	18	0,863	0,076	31
0	0	10	0	337,528	0,242	27	148,221	1,995	35	2,258	0,319	8	574,117	1,548	32	3,584	0,225	44
0	0	15	0	468,393	0,536	25	228,163	8,817	33	14,432	0,600	11	580,600	4,103	30	11,624	1,528	46
0	0	0	1	0,105	0,043	0	62,692	0,113	20	4,616	0,387	0	5,811	0,042	13	26,934	0,144	13
0	0	0	5	5,227	0,075	1	278,937	35,142	68	1 119, 846	608,474	79	17,177	0,107	28	173,428	3,903	61
0	0	0	10	33,375	0,424	8	138,024	95,579	94	-	-	100	68,031	4,704	42	563,541	24,332	95
0	0	0	15	150,653	1,753	16	-	-	100	-	-	100	99,678	53,101	67	-	-	100
25	5	0	0	93,504	0,040	0	4,408	0,055	11	52,813	0,077	0	161,307	0,042	0	5,855	0,048	0
50	25	0	0	861,516	467,008	8	45,736	2,157	51	222,688	0,112	0	838,546	575,054	9	20,738	6,490	0
100	50	0	0	1 563, 853	1 378, 030	21	92,662	98,487	80	660,699	0,868	0	1 492, 257	1 367, 830	22	27,360	9,100	0
5	25	5	5	1 072, 227	468,463	45	862,258	539,042	89	1 709, 468	2 305, 500	93	792,369	49,447	62	975,991	348,559	81
10	10	5	5	508,984	0,338	30	409,681	45,839	80	540,447	105,732	85	841,158	441,759	48	473,617	25,765	74

Table A.7: Results for model revision of FY, SP, TCR, MCC, and Th, confronted with five time-series observations with three time steps under asynchronous update scheme. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 3 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,251	0,286	0	0,406	0,395	0	0,498	0,500	0	0,295	0,300	0	0,306	0,292	0
25	0	0	0	0,259	0,250	0	0,448	0,441	0	0,503	0,504	0	0,338	0,344	0	0,352	0,355	0
50	0	0	0	0,272	0,281	0	0,450	0,433	0	0,578	0,604	0	0,332	0,332	0	0,332	0,320	0
100	0	0	0	0,272	0,268	0	0,399	0,396	0	0,567	0,538	0	0,301	0,318	0	0,343	0,349	0
0	5	0	0	0,302	0,299	0	0,488	0,497	0	0,561	0,524	0	3,080	0,351	0	0,289	0,284	0
0	10	0	0	0,328	0,302	0	0,480	0,500	0	0,560	0,522	0	5,550	0,298	1	0,287	0,283	0
0	15	0	0	0,405	0,306	0	0,392	0,387	0	0,528	0,520	0	11,026	0,384	1	0,354	0,355	0
0	20	0	0	1,170	0,355	0	0,442	0,442	0	0,564	0,527	0	8,848	0,348	4	0,340	0,322	0
0	25	0	0	1,594	0,318	0	0,436	0,431	0	0,531	0,525	0	11,041	0,322	4	0,333	0,305	0
0	50	0	0	33,478	1,582	0	0,494	0,503	0	0,565	0,530	0	29,372	0,706	20	0,415	0,326	0
0	75	0	0	238,936	34,151	0	0,387	0,384	0	0,572	0,531	0	39,783	1,550	53	0,824	0,523	0
0	0	1	0	0,328	0,341	0	0,466	0,485	0	0,518	0,500	0	0,289	0,289	0	0,310	0,292	0
0	0	5	0	0,241	0,239	0	0,368	0,372	0	0,526	0,495	0	0,300	0,296	0	0,294	0,282	0
0	0	10	0	0,263	0,260	0	0,373	0,370	0	0,437	0,449	0	0,257	0,250	0	0,278	0,270	0
0	0	15	0	0,206	0,205	0	0,363	0,354	0	0,459	0,456	0	0,238	0,236	0	0,266	0,258	0
0	0	0	1	0,223	0,189	0	0,416	0,413	0	0,652	0,625	0	0,339	0,311	0	0,324	0,320	0
0	0	0	5	0,593	0,301	0	1,186	0,476	1	18,075	1,035	11	0,372	0,306	0	32,000	0,423	6
0	0	0	10	3,302	0,283	3	48,156	0,657	6	191,895	14,397	43	78,615	0,318	0	293,350	4,282	26
0	0	0	15	7,528	0,315	6	117,468	0,816	16	533,185	63,266	84	148,966	0,342	3	383,454	130,733	67
25	5	0	0	0,308	0,327	0	0,400	0,397	0	0,528	0,504	0	0,352	0,338	0	0,298	0,295	0
50	25	0	0	1,112	0,307	0	0,503	0,508	0	0,559	0,530	0	8,637	0,353	2	0,290	0,287	0
100	50	0	0	101,106	0,419	0	0,483	0,487	0	0,522	0,521	0	23,627	0,614	22	0,430	0,300	0
5	25	5	5	20,932	0,329	2	0,956	0,541	0	7,243	1,011	13	27,194	0,356	6	125,318	0,692	10
10	10	5	5	11,186	0,254	1	17,668	0,467	0	30,571	1,113	7	5,901	0,341	1	33,689	0,471	7

Table A.8: Results for model revision of FY, SP, TCR, MCC, and Th, confronted with five time-series observations with twenty time steps under synchronous update scheme. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 3 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	0,370	0,351	0	0,671	0,493	0	0,590	0,570	0	0,678	0,384	0	0,999	0,493	0
25	0	0	0	0,363	0,344	0	1,002	0,515	0	0,624	0,584	0	0,709	0,398	0	1,531	0,503	0
50	0	0	0	0,362	0,341	0	1,193	0,539	0	0,626	0,591	0	1,884	0,407	0	1,988	0,520	0
100	0	0	0	0,323	0,318	0	1,910	0,605	0	0,643	0,605	0	2,768	0,428	0	4,428	0,518	0
0	5	0	0	151,053	0,424	0	23,263	0,766	8	47,559	0,564	0	228,281	0,648	22	4,097	0,499	0
0	10	0	0	253,263	0,756	2	57,879	0,989	19	68,883	0,562	0	345,673	1,008	29	7,193	0,548	0
0	15	0	0	611,574	237,375	5	84,908	13,335	25	94,456	0,604	0	527,698	439,478	43	11,809	0,758	0
0	20	0	0	783,316	471,110	9	99,780	26,613	36	128,838	0,654	0	573,800	445,633	56	13,864	0,796	0
0	25	0	0	1 006, 330	475,483	9	122,234	37,737	44	130,726	0,660	0	534,667	446,348	67	14,399	11,818	0
0	50	0	0	1 687, 878	1 233, 490	40	193,176	226,468	81	202,630	0,808	0	1 714, 987	1 641, 190	89	30,699	27,519	0
0	75	0	0	2 078, 331	2 537, 190	76	847,240	318,779	90	163,514	1,684	0	2 237, 030	2 237, 030	99	50,794	54,019	0
0	0	1	0	173,969	0,396	18	54,214	0,781	14	0,648	0,603	3	431,983	0,655	16	0,597	0,512	24
0	0	5	0	145,952	0,395	32	83,629	9,822	44	2,917	0,813	9	551,071	3,315	36	6,150	0,553	40
0	0	10	0	153,948	0,487	35	133,729	15,615	53	4,433	1,403	11	511,235	4,946	54	3,220	0,739	61
0	0	15	0	126,155	0,669	39	333,519	31,110	66	23,010	2,304	20	599,740	5,961	60	18,754	0,830	69
0	0	0	1	0,428	0,361	0	14,984	0,810	32	6,370	1,587	0	12,066	0,648	23	15,936	1,632	12
0	0	0	5	22,332	0,432	1	133,954	22,878	77	1 221, 388	533,617	80	65,005	1,156	38	230,699	7,779	69
0	0	0	10	63,957	0,851	9	1008,229	465,962	97	-	-	100	200,018	67,062	60	265,610	31,696	97
0	0	0	15	166,218	2,918	18	-	-	100	-	-	100	194,446	57,615	82	-	-	100
25	5	0	0	94,188	0,355	0	28,089	0,818	11	40,823	0,561	0	141,176	0,645	24	6,133	0,500	0
50	25	0	0	839,938	471,929	9	85,660	25,524	51	168,599	0,597	0	784,962	546,629	72	20,355	10,356	0
100	50	0	0	1 546, 241	1 380, 905	22	211,082	84,279	81	478,367	1,104	0	1 217, 415	1216, 500	91	27,788	11,010	0
5	25	5	5	1 088, 608	471,932	56	1 081, 923	500,249	97	1 783, 888	1 453, 270	95	1 141, 617	1 071, 785	92	924,193	88,985	88
10	10	5	5	436,920	0,699	39	851,979	381,668	93	685,947	115,152	87	1 094, 204	788,874	74	474,074	16,871	80

Table A.9: Results for model revision of FY, SP, TCR, MCC, and Th, confronted with five time-series observations with twenty time steps under asynchronous update scheme. F%, E%, R%, and A% are the probabilistic parameters used to change the original models. Average (Avg.) and median (Med.) times, in seconds, of the solved instances. #TO is the number of timeouts out of 100 considering a time limit of 3 600 seconds.

(%)				FY			SP			TCR			MCC			Th		
F	**E**	**R**	**A**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**	**Avg. (s)**	**Med. (s)**	**#TO**
5	0	0	0	16,562	16,363	0	25,005	24,411	0	28,710	28,652	0	17,398	17,151	0	16,485	16,393	0
25	0	0	0	16,668	16,538	0	25,639	25,128	0	29,343	28,997	0	17,505	17,254	0	16,956	16,582	0
50	0	0	0	15,957	15,911	0	26,373	26,060	0	28,970	28,819	0	17,528	17,348	0	16,488	16,355	0
100	0	0	0	14,807	14,701	0	29,322	28,815	0	30,458	29,851	0	17,552	17,305	0	16,786	16,614	0
0	5	0	0	109,487	16,377	0	116,984	33,587	24	28,609	28,588	0	33,110	16,559	16	23,235	16,477	0
0	10	0	0	143,790	16,477	1	273,981	33,564	44	29,504	28,921	0	18,575	17,042	16	27,603	16,603	0
0	15	0	0	302,315	17,737	3	401,163	33,261	57	69,856	28,764	0	45,327	17,205	23	32,722	16,532	0
0	20	0	0	475,940	16,348	4	697,376	33,984	65	132,233	28,515	0	74,439	18,005	32	37,723	16,438	0
0	25	0	0	523,951	16,533	4	1050,159	140,906	73	165,542	28,289	0	86,615	17,906	43	37,715	16,677	0
0	50	0	0	1046,724	17,752	18	1590,402	1369,147	92	480,665	30,341	0	269,559	28,443	52	45,745	54,474	0
0	75	0	0	1374,024	57,402	46	2823,235	2658,765	96	1114,170	1323,950	0	399,089	88,909	73	48,493	55,138	0
0	0	1	0	15,009	15,420	2	46,902	33,046	15	28,063	27,952	0	16,741	16,531	0	15,927	15,792	4
0	0	5	0	13,909	14,156	5	82,851	25,152	33	27,306	27,647	0	15,714	15,838	0	15,339	15,355	6
0	0	10	0	12,545	12,327	5	129,420	29,898	35	24,531	25,874	0	15,334	15,134	0	14,832	14,958	6
0	0	15	0	11,759	11,523	4	188,572	31,302	50	22,502	21,658	0	15,042	14,997	0	13,887	14,130	8
0	0	0	1	16,322	16,461	0	222,456	34,351	24	33,821	33,504	0	56,258	16,801	0	41,263	18,259	12
0	0	0	5	25,568	16,014	0	744,221	43,526	75	211,442	50,755	36	101,902	19,654	12	297,203	24,737	56
0	0	0	10	16,464	16,156	4	738,437	145,950	96	-	-	100	257,200	86,724	31	1045,175	789,611	91
0	0	0	15	18,688	16,009	12	44,554	44,554	99	39,644	39,644	99	410,092	241,163	42	-	-	100
25	5	0	0	60,738	16,052	0	65,739	33,166	30	28,071	28,019	0	26,128	17,007	16	21,036	17,630	0
50	25	0	0	507,558	15,358	3	943,385	122,151	67	88,800	28,519	0	165,400	19,797	35	29,453	16,198	0
100	50	0	0	1292,601	17,042	3	1777,356	2162,700	85	594,652	30,550	2	315,118	31,741	45	32,723	16,700	0
5	25	5	5	530,478	15,259	13	1389,377	972,294	90	825,188	91,886	59	459,281	18,736	56	450,681	48,568	62
10	10	5	5	231,279	14,960	4	1008,699	161,689	92	422,554	63,104	38	230,970	18,462	39	301,354	25,621	63

The following figures show time results in seconds considering one and five time-series observations, and observations with three and twenty time steps. The results are presented by model: FY in Figures A.1 and A.2; SP in Figures A.3 and A.4; TCR in Figures A.5 and A.6; MCC in Figures A.7 and A.8; and Th in Figures A.9 and A.10. Side by side are shown the results for the synchronous update scheme (on the left) and for the asynchronous update scheme (on the right). Results show that the solving time increases with the increase of the number of observations considered, as well as with the increase of the size of the observations (number of time steps).

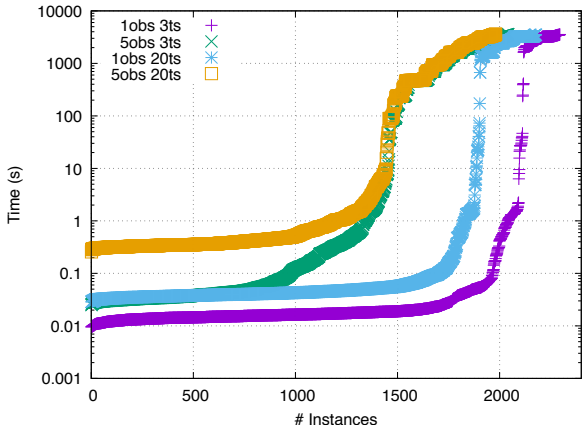


Figure A.1: Solving times for FY model under synchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

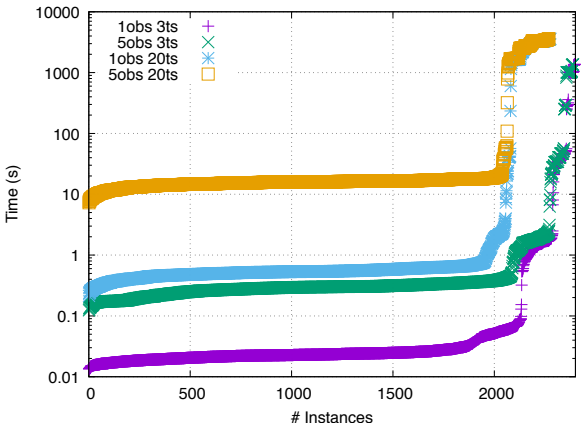


Figure A.2: Solving times for FY model under asynchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

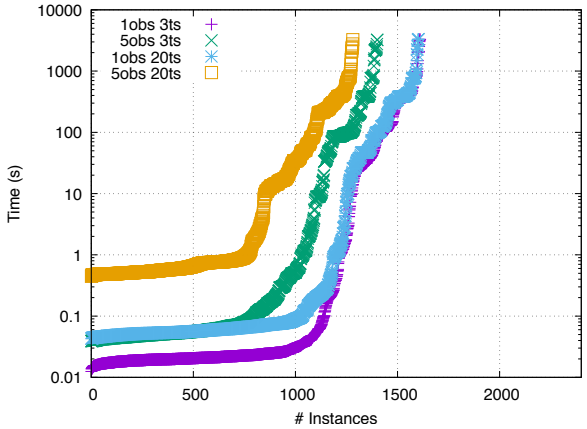


Figure A.3: Solving times for SP model under synchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

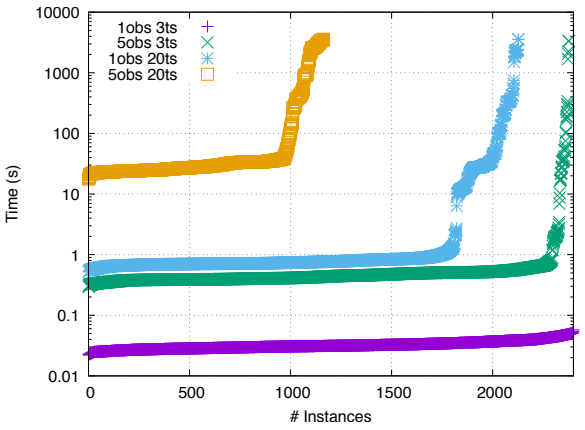


Figure A.4: Solving times for SP model under asynchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

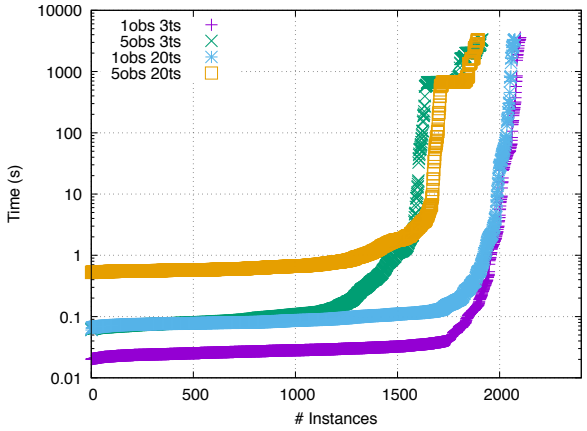


Figure A.5: Solving times for TCR model under synchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

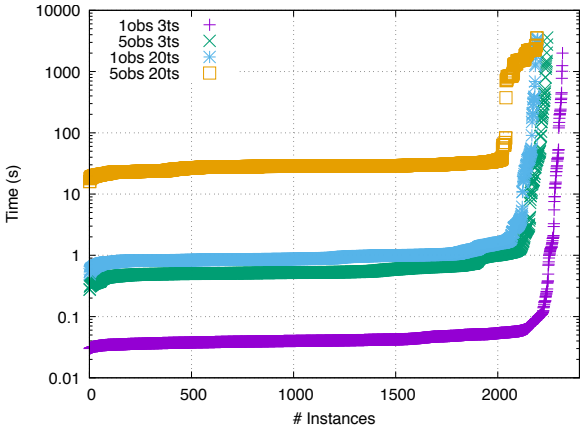


Figure A.6: Solving times for TCR model under asynchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

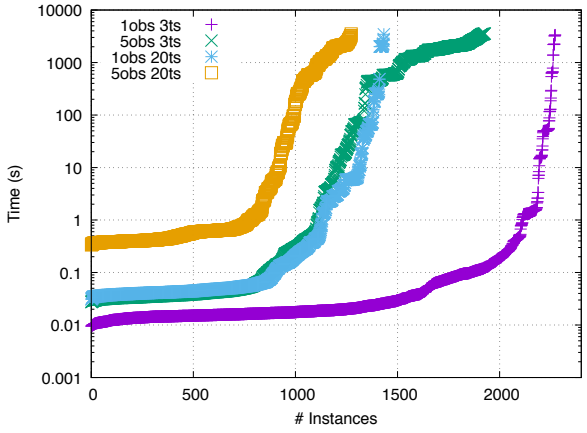


Figure A.7: Solving times for MCC model under synchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

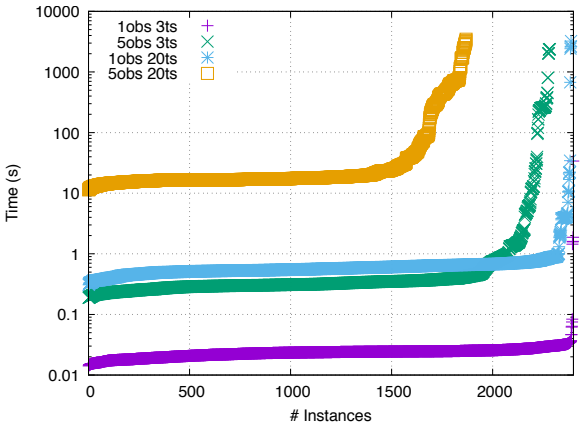


Figure A.8: Solving times for MCC model under asynchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

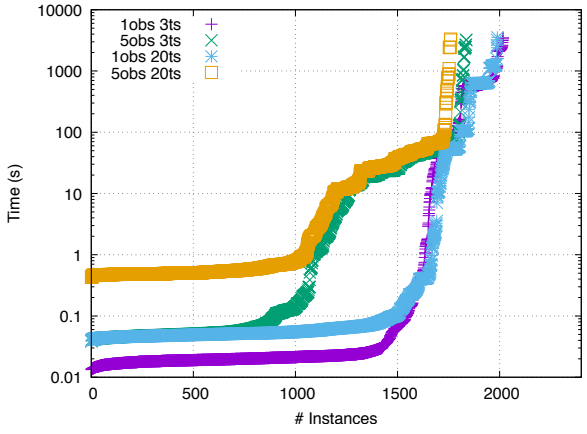


Figure A.9: Solving times for Th model under synchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

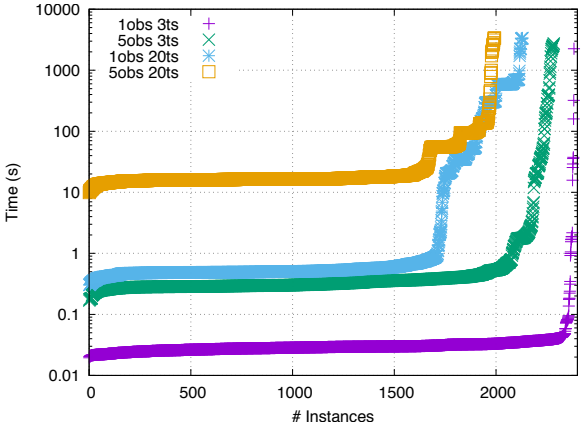
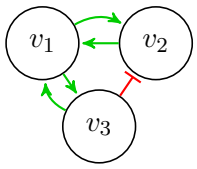


Figure A.10: Solving times for Th model under asynchronous update scheme. Results consider different number of observations (obs) and time steps (ts).

Appendix B

Tutorial

A brief tutorial of the ModRev tool[1] with small examples is here presented.



$$f_{v_1} = v_2 \wedge v_3$$
$$f_{v_2} = v_1 \vee \neg v_3$$
$$f_{v_3} = v_1$$

Figure B.1: Example of a Boolean logical model.

The model shown in Figure B.1 is represented by the following listing:

```
vertex(v1). vertex(v2). vertex(v3).
edge(v1,v2,1). edge(v1,v3,1). edge(v2,v1,1).
edge(v3,v1,1). edge(v3,v2,0).
functionOr(v1,1..1).
functionAnd(v1,1,v2). functionAnd(v1,1,v3).
functionOr(v2,1..2).
functionAnd(v2,1,v1). functionAnd(v2,2,v3).
functionOr(v3,1..1).
functionAnd(v3,1,v1).
```

Now let us consider that we want to define a stable state observation in which v_1 has value 0, v_2 has value 0, and v_3 has value 1, as following:

```
exp(p1).
obs_vlabel(p1,v1,0). obs_vlabel(p1,v2,0). obs_vlabel(p1,v3,1).
```

Using ModRev tool, giving the model defined above (as a file `model.lp`) and the stable state (as a file `obsSS.lp`), execute the following command:

[1] `https://filipegouveia.github.io/ModelRevisionASP`

```
$ ./modrev -m model.lp -obs obsSS.lp -ss
```

```
### Found solution with 1 repair operation.
    Inconsistent node v3.
        Repair #1:
            Flip sign of edge (v1,v3).
```

This output means that the model in Figure B.1 can be repaired by changing the interaction type between v_1 and v_3. If we repair the model and execute the above command again, the result will be:

```
This network is consistent!
```

Now let us assume that the user knows that the interaction between v_1 and v_3 is correct, and wants to prevent repairs over it. The predicate `fixed(v1,v3).` can be used to define that the edge between these nodes can not be changed or removed. Adding this predicate to the model and running the command above, we obtain the following result:

```
### Found solution with 2 repair operations.
    Inconsistent node v3.
        Repair #1:
            Change function of v3 to (v1) || (v3)
            Add edge (v3,v3) with sign 1.
```

A different set of repair operations is obtained that does not change the fixed edge. Now assume that the user wants to prevent any repair over the node v_3. The predicate `fixed(v3).` can be used to prevent that node to be inconsistent. However, in this example, if we prevent any change to node v_3, considering its regulatory function, and that v_1 has value 0 and v_3 has value 1, and we are in the presence of a stable state, it becomes impossible to repair the network. In this case, when the model is over-constrained, using the same command as before, the tool produces the following message:

```
It is not possible to repair this network.
```

Consider now that we have, for the same model in Figure B.1, a time-series data as shown in Table B.1. Consider that this experimental observation with three time-steps (0, 1 and 2) is considering a synchronous update scheme.

Table B.1: Synchronous time-series data

Node	Time 0	1	2
v_1	0	1	0
v_2	0	0	0
v_3	1	0	0

We can represent the time-series data using the following listing:

```
#const t = 2.
exp(p2).
obs_vlabel(p2,0,v1,0). obs_vlabel(p2,0,v2,0).
obs_vlabel(p2,0,v3,1).
obs_vlabel(p2,1,v1,1). obs_vlabel(p2,1,v2,0).
obs_vlabel(p2,1,v3,0).
obs_vlabel(p2,2,v1,0). obs_vlabel(p2,2,v2,0).
obs_vlabel(p2,2,v3,0).
```

Note that we start the file indicating the maximum value of time step with `#const t = 2`.

Using ModRev to verify whether the model is consistent, while considering the above time-series data (as a file `obsTS01.lp`) under a synchronous update scheme, execute the following command:

```
$ ./modrev -m model.lp -obs obsTS01.lp -up s
```

This will produce the following result:

```
### Found solution with 5 repair operations.
    Inconsistent node v1.
        Repair #1:
            Change function of v1 to (v2) || (v3)
    Inconsistent node v2.
        Repair #1:
            Change function of v2 to (v1 && v3)
            Flip sign of edge (v1,v2).
        Repair #2:
            Change function of v2 to (v1 && v3)
```

```
        Flip sign of edge (v3,v2).
Inconsistent node v3.
    Repair #1:
        Change function of v3 to (v1 && v2)
        Add edge (v2,v3) with sign 1.
    Repair #2:
        Change function of v3 to (v1 && v3)
        Add edge (v3,v3) with sign 1.
```

Note that now we have multiple choices to render the model consistent. To repair node v_2, for example, one can apply the operations in `Repair #1` or in `Repair #2`. The same applies to repair node v_3.

If instead of a time-series data under a synchronous update scheme, we are under an asynchronous update scheme, the previous command would change from `-up s` to `-up a`. The option `-up` indicates the update scheme to be considered, with argument `s` for synchronous and `a` for asynchronous.

ModRev also supports incomplete time-series data. Assume that we have the experimental observation shown in Table B.2, where node v_3 was not observed, and a value of v_1 was also not observed.

Table B.2: Incomplete synchronous time-series data

		Time		
		0	1	2
Node	v_1	0		1
	v_2	1	0	0
	v_3			

Consider the following representation of an incomplete time-series data:

```
#const t = 2.
exp(p3).
obs_vlabel(p3,0,v1,0). obs_vlabel(p3,0,v2,1).
obs_vlabel(p3,1,v2,0).
obs_vlabel(p3,2,v1,1). obs_vlabel(p3,2,v2,0).
```

Executing the following command, while considering the above experimental observation (as a file `obsTS02.lp`) under synchronous update scheme, produces the result below.

```
$ ./modrev -m model.lp -obs obsTS02.lp -up s
```

```
### Found solution with 3 repair operations.
    Inconsistent node v1.
        Repair #1:
            Change function of v1 to (v2) || (v3)
            Flip sign of edge (v2,v1).
    Inconsistent node v2.
        Repair #1:
            Change function of v2 to (v1 && v3)
```

ModRev tool also supports confronting a model with multiple experimental observations at the same time. For example, we could confront the model of Figure B.1 with the two time-series data above, using the command:

```
$ ./modrev -m model.lp -obs obsTS01.lp obsTS02.lp -up s
```

Note that the directive `#const t = 2` must only be defined once.