Analyzing Long-Term Dynamics of Biological Networks with Answer Set Programming

Emna Ben Abdallah, Maxime Folschette, Morgan Magnin

Chapter 7

Analyzing Long-Term Dynamics of Biological Networks with Answer Set Programming

Emna Ben Abdallah,[1] Maxime Folschette,[2*] and Morgan Magnin[3]

[1] *Independent researcher*
[2] *Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France*
[3] *Centrale Nantes, Université de Nantes, CNRS, LS2N, F-44000 Nantes, France*

*Corresponding Author: Maxime Folschette, maxime.folschette@centralelille.fr

Merits of ASP for biological systems analysis: The formal study of the dynamics of biological systems raises many problems, e.g., identification of attractors, bifurcations, reachability, that are combinatorial by essence. Making these issues tractable requires to design efficient methods that rely on solid and efficient programming frameworks. During the last decades, Answer Set Programming (ASP) [Baral, 2003] has proven to be a strong logic programming paradigm to deal with the inherent complexity of the biological models, allowing to quickly investigate a wide range of configurations. ASP can efficiently enumerate a large number of answer sets, as well as easily filter the results thanks to constraints based on certain properties. This chapter begins by a motivation of the merits of ASP in biological studies based on a

state of the art. The basic concepts about ASP and its use in systems biology are then introduced. After having given an overview of the different issues that can be tackled using ASP, the focus is turned onto one problem that is of critical importance: model-checking with ASP, and more specifically the identification of attractors. The merits of such an approach are then exhibited using case studies. This chapter ends on a summary of the promising perspectives that ASP holds in future works about biological dynamics.

Keywords: Answer Set Programming (ASP), logic programming, attractors, stable states, dynamical analysis

7.1. Introduction

Recent advances in molecular biology have made it possible to produce comprehensive genome maps of many living organisms. However, these static maps — made popular in case of humans in particular through the "Human Genome Project" [Collins et al., 2003] — are not sufficient to account for the intrinsic complexity of living things. It is indeed necessary to consider genetic phenomena with regard to the full complexity of the dynamics of their interactions, either internal or with their environment (for example, the stress represented by an external component such as light or the ingestion of a protein).

Meanwhile, the development of biotechnologies such as DNA chips and the rise of high-throughput sequencing techniques (such as RNAseq) facilitate the production of time series data corresponding to the expression of several thousand genes. The question then arises of the meaning to be given to this profusion of information. One of the main current challenges in systems biology is thus to integrate these high-throughput data and to infer the associated genetic regulatory networks. As the interactions occur at different scales (genes, proteins, biochemical components, cells, etc.), there is a

need to develop integrative methods that can formally and automatically learn biological data to facilitate understanding, at a systemic level, of the phenomena involved.

In fact, the modeling of biological regulatory mechanisms can be divided into two main trends. The first, quantitative, is based on ordinary differential equations involving the quantitative expression of interacting components. However, these equations are generally non-linear, which prevents their analytical resolution. In addition, the biological data obtained experimentally are generally relatively noisy, which means that they must be filtered efficiently. Conversely, the second tendency consists in addressing the problem using a discrete methodology. In other words, the expression of each component is assessed according to several (usually two, sometimes three or four) qualitative levels, and no longer quantitatively. Even though such discrete modeling might be considered a less faithful abstraction of reality, it has been shown to be effective in addressing many qualitative biological questions (such as understanding the interactions between several components of a biological system, or determining the reachability of a state).

In this chapter, the emphasis is placed on regulatory processes at the genetic level. We choose here to abstract certain mechanisms, e.g., the ones at the molecular level, which would require to specify in detail the behavior of each biological element like proteins, plasmids, etc.

Difficulties encountered in analysis and prediction can be considered to fall into one of the following four broad categories:

- Parameter identification: discrete models, such as synchronous Boolean networks introduced by Stuart Kauffman (1969) or asynchronous multi-valued networks introduced by René Thomas (1973), do not only require information about the network topology and the type of influences — activation or inhibition — between genes, but also on the respective strength of each interaction.

This type of information is needed to determine, for example, the predominant influence when a gene is the single target of two influences with opposite effects. As we mentioned previously, such information is represented either at the core of a network node update function or in a dedicated parameterization of the model. In either cases, this information can be deduced from experimental biological observations by automatic methods such as model-checking or constraint programming.

- Model inference: broader than the identification of parameters, the problem of inference also encompasses the determination of the overall structure of the network. Considering time series data that can be converted (once discretized) into a state transition system, it may be important — to facilitate data analysis — to build a discrete model compatible with the available data.

- The identification of stable states and attractors, key properties of most biological systems, which are tackled in this chapter.

- Model control, which is the ability to control a model so that it exhibits a desired behavior or, on the contrary, guarantees that undesirable behaviors are avoided. This issue is closely linked to the analysis of the aforementioned key properties. This is the most recent, but also the most difficult, research topic in the field of systems biology, closely linked to the problem of gene therapy.

Answer Set Programming (ASP) is a logic programming paradigm that has proven to be useful in all these four challenges. We will here focus on the ones that seem, to us at least, the most representative of ASP merits, especially the full identification of attractors.

7.2. State of the Art

7.2.1. Qualitative modeling of biological systems

In the last decades, the emergence of a wide range of new technologies has made it possible to produce a massive amount of biological data (genomics, proteomics...). This leads to considerable developments in systems biology which takes profit from this data. In order to understand the nature of a cellular function or more broadly a living biological system (healthy or diseased), it is indeed essential to study not only the individual properties of cellular components, but also their interactions. The behavior and functionalities of the cells emerge from such networks of interactions.

Considering this paradigm, the long-term behavior of a regulatory network's dynamics is of specific interest [Wuensche, 1998]. Indeed, at any moment, a system may fall into a *trap domain*, which is a part of its dynamics that cannot be escaped. While evolving, the system may eventually fall into new and smaller trap domains, thus reducing its possible future behaviors by making previous states no longer reachable. This phenomenon depends on biological disruptions or other complex phenomena. Such outline has been interpreted as distinct responses of the organism, such as differentiating into distinct cell types in multicellular organisms [Huang et al., 2005].

Moreover, when refining a model of a living system, one way to remove inconsistencies or to predict missing information in biological models consists in comparing the attractors of the model with the experimentally observed long-term behavior. For instance, the model of the cellular development of *Drosophila melanogaster* was described using Boolean networks and their attractors [González et al., 2008, Albert and Othmer, 2003].

As mentioned in the previous sections, the modeling of biological regulatory mechanisms can be divided into two main categories: based on either ordinary differential equations or discrete values. Even if discrete modeling can be roughly seen as a less faithful abstraction, it has proven to be effective in addressing many biological questions, such as understanding how biological systems evolve or determining whether certain states are reachable. In fact, since data in biology are often more qualitative than quantitative, it is natural that an alternative view to differential equations started to emerge in the late 1960s. This qualitative modeling is based on the following principle: the expression of a component can be encoded with a Boolean variable. This corresponds to the fact that a component (e.g., a gene) is "on" (i.e., the regulations it controls are expressed) or "off". The relations between Boolean variables are governed by activation or inhibition relations, respectively represented by positive or negative signed arcs. Various kinds of mathematical models implementing these precepts have been proposed for the modeling of Biological Regulatory Networks (BRNs). The two main families of such models have been Stuart Kauffman's synchronous Boolean networks [Kauffman, 1969] on the one hand, and René Thomas' asynchronous networks [Thomas, 1991] on the other.

In this chapter, the focus is on a subclass of automata networks, called Asynchronous Automata Networks (AAN) [Folschette et al., 2015, Paulevé, 2016a], which is an extension of a previous framework called Process Hitting [Paulevé et al., 2014], and is convenient to model BRNs. This qualitative approach is based on three key concepts:

- Biological components (e.g., genes) are abstracted in the form of *automata*. The different local states (that are not restricted to Boolean values) of each automaton correspond to the different discrete qualitative levels of the components represented by the automaton.

- Interaction between biological components is modeled in an atomic way by local transitions on the automata, where each one is conditioned by a set of required local states in different automata and can modify the local state of a unique automaton.

- In the modeling task, such a representation makes it possible to build the largest possible dynamics, and then to proceed by successive refinements to restrict the possible behaviors [Paulevé et al., 2011].

To sum up, AANs allow to have multiple requirements for a local transition to occur, but *not* to synchronize several local transitions in different automata. In this sense, they are considered *Asynchronous* Automata Networks, although in the following, we define and apply both the asynchronous and synchronous semantics to such networks. Here, the synchronous semantics allows to fire all local transitions at once, but not to add information about specific transitions being synchronized. Depending on the chosen semantics (asynchronous or synchronous), AANs especially encompass the Boolean frameworks of René Thomas and Stuart Kauffman.

Qualitative frameworks have received substantial attention, because of their capacity to capture the switching behavior of genetic or biological processes, and therefore, the study of their long-term behavior. This adds up to the choice of a qualitative representation for the identification of trap domains. In such a qualitative framework, a minimal trap domain can take two different forms: it can be either a *stable state*, also called *fixed point* or *steady-state*, which is one state in which the system does not evolve anymore; or an *attractor*, which is a minimal set of states that cannot be escaped, and thus loops indefinitely.

7.2.2. Identifying Attractors: A Major Challenge

The computational problem of finding all attractors in a BRN is difficult. Even the simpler problem of deciding whether the system has a stable state, which can be seen as the smallest kind of attractor, is NP-hard [Zhang et al., 2007]. Based on this, many studies have proven that computing attractors in BRNs is also a NP-hard problem [Klemm and Bornholdt, 2005, Akutsu et al., 2012]. Although some methods exist with a lesser complexity, consisting for instance in randomly selecting an initial state and following a long enough trajectory, hoping to eventually finding an attractor, they are not exhaustive and may miss some (hard to reach) attractors.

Therefore, in the absence of more efficient exhaustive methods, it is still relevant to develop an approach to resolve the original NP-hard problem of attractors identification. Such an approach consists in exhaustively examine all possible states of a network, along with all possible paths from each of these states. Obviously, this brute force method is very time and memory consuming: 2^n initial states have to be considered for a Boolean model with n nodes; and multi-valued networks raise this value even more. Furthermore, a sufficient number of computations have to be performed to ensure that all trajectories have been explored and all attractors are found. This high complexity justifies the use of a tool able to tackle such hard problems.

The simplest way to detect attractors is to enumerate all the possible states and to run a simulation from each one until an attractor is reached [Somogyi and Greller, 2001]. This method ensures that all attractors are detected but it has an exponential time complexity, therefore its applicability is highly restricted by the network size.

Regarding Boolean networks only, algorithms for detecting attractors have been extensively studied in the literature. Irons [2006] proposes to analyze partial states in order to discard potential attractors more efficiently. This method improves the efficiency from exponential time to polynomial time for a subset of biological Boolean

models that is highly dependent on the topology of the underlying network (in terms of indegree, outdegree and update functions). Another method, called GenYsis [Garg et al., 2007], starts from one (randomly selected) initial state and detects attractors by computing the successor and predecessor states of this initial state. It works well for small Boolean networks, but becomes inefficient for large Boolean networks.

More generally, the efficiency and scalability of attractor detection techniques are further improved with the integration of two techniques. This first is based on Binary Decision Diagrams (BDD), a compact data structure for representing Boolean functions. In [Zhao et al., 2014], algorithms have been based on the reduced-order BDD (ROBDD) data structure, which further speeds up the computation time of attractor detection. These BDD-based solutions only work for BRNs of a hundred of nodes and also suffer from the infamous state explosion problem, as the size of the BDD depends both on the regulatory functions and the number of nodes in the BRN. The other technique consists in representing the attractor enumeration problem as a satisfiability (SAT) problem such as in [Dubrova and Teslenko, 2011]. The main idea is inspired by SAT-based bounded model-checking: the transition relation of the BRN is unfolded into a bounded number of steps in order to construct a propositional formula which encodes attractors and which is then solved by a SAT solver. In every step, a new variable is required to represent a state of a node in the BRN. It is clear that the efficiency of these algorithms largely depends on the number of unfolding steps and the number of nodes in the BRN. The method presented below is also inspired from this bounded model-checking approach.

In [Mushthofa et al., 2014], the authors separated the rules that describe the network (the nodes and their interactions: activation or inhibition) from the rules that define its dynamics (for instance: a gene will be activated in the next state if all its activators are active or when at least one of its activators is active at the current state).

This allows to obtain more flexible simulations, and the authors also chose to use the declarative paradigm Answer Set Programming [Baral, 2003] in order to have more liberty in the expression of evolution rules. They illustrated that specifying large networks with rather complicated behaviors becomes cumbersome and error prone in paradigms like SAT, whereas this is much less the case in a declarative approach such as theirs.

7.2.3. Answer Set Programming for Systems Biology

More recently, Answer Set Programming (ASP) has been recently used successfully in systems biology on two different challenges. The first one consists in the synthesis of discrete models from different background knowledge. In [Chevalier et al., 2019], the authors focus on Boolean networks and exhibit an ASP-based method to determine the Boolean function from information about the attractors (either stable states or trap spaces). They show the scalability of their approach, that is able to tackle networks up to 50 nodes with an in-degree of 15 per node, and advocate for the associated benefits by addressing the cell differentiation process in the central nervous system. The efficient matching between biological data given as input and a family of admissible models is really one of the cornerstones that motivates for the use of ASP. In [Lemos et al., 2019], the authors introduce a method to revise models, encoded as Boolean logical regulatory graphs, from time series data. Such work stems from the need to be able to revise existing Boolean models that may become inconsistent when new data is made available. The authors define various atomic revision operations, e.g., negation of a regulator (changing a regulator from an inhibitor to an activator, and conversely), operator substitution (turning a Boolean operator from AND to OR,

and conversely), removal of a regulator (removing all occurrences of a given regulator from a logical function). They show how such a method can be applied to biological case studies by applying this approach to 5 models with associated time series containing a dozen of nodes, with a number of time steps varying from 10 to 13.

The second main challenge — which is explored in details in this chapter — consists in the identification of attractors in biological regulator networks. In [Ben Abdallah et al., 2017], the authors proposed a first version of the approach and algorithms that will be explained in more details below. Meanwhile, other authors got interest in such a problem. In [Khaled and Benhamou, 2020], the authors search and enumerate attractors in asynchronous Boolean networks that have the form of a circuit using ASP by introducing a new resolution semantics that does not use the usual negation by failure. Instead, they rely on a weak version of the negation by failure that allows them to provide an enumeration approach preventing the simulation of the underlying networks. This approach targets networks with a given structure (all nodes must have an incoming edge and some results have been obtained only for cyclic interaction graphs). The methodology detailed in this chapter is different in the sense that it is iterative and without any assumption on the topology of the interaction graph.

7.2.4. Enumerating Attractors of a Biological Model Using Answer Set Programming

Our goal in this chapter is to present two methods to enumerate minimal trap spaces of a BRN modeled in AAN: (1) finding all stable states, and (2) enumerating all attractors up to a given length n. A *stable state*, also called *fixed point*, is a state with no more possible dynamical evolution. An *attractor* is a minimal trap domain which is not a fixed point; in other words, it is a minimal set of at least two states that cannot be

dynamically escaped. Here, the *length* refers to the number of dynamical transitions required to cover all states of the attractor, which can be equal or higher than the number of states if there are complex dynamical branchings. This notion of length is required since it gives a higher bound to the dynamical exploration performed, and makes this method fall under bounded model checking.

We focus on two widespread non-deterministic semantics: asynchronous and synchronous. It has to noted that the stable states are the same in both dynamics [Klarner et al., 2015]. Although we do not prove it, we claim that our method is also correct for the enumeration of attractors in the asynchronous dynamics and returns all attractors up to the given length. Regarding the synchronous dynamics, we also claim that out method is correct for *simple attractors*, that is, attractors made of exactly one loop; however, it might miss some *complex attractors*, that are compositions of several loops. We use ASP to perform these aforementioned enumerations.

The originality of our approach is to consider AAN models with ASP. Within the AAN formalism, interactions are modeled as automata transitions instead of generic influences. This allows to define finer influences between components than in formalisms based on parameters or evolution functions. For instance, an AAN model can contain non-monotonic influences, or be built as a union of models to check several dynamics simultaneously [Paulevé et al., 2011]. As mentioned in the introduction of this chapter, AANs are more expressive than the modelings of René Thomas [Thomas, 1973] or Kauffman [Kauffman, 1969], which also means that models in these formalisms can be expressed and analyzed as AANs, including their multi-valued iterations. Moreover, automata-like transitions happen to be easily represented in ASP. Finally, since ASP is a programming paradigm, this would theoretically allow a much wider expression power in the description of update semantics (generalized, block-parallel, with memory, with priorities, etc.) although these are not tackled here.

Although some ideas of this approach were previously introduced in [Ben Abdallah et al., 2015, 2017], the method presented in this chapter has been improved. Thanks to the possibility now offered by the ASP solver Clingo of interfacing with scripts, we propose an approach using Python to perform post-filtering or pre-filtering to enumerate all attractors up to a given length without yielding many spurious solutions as in the previous iterations. This conjunction of ASP and Python illustrates the benefits of such a combination to tackle efficient enumeration of answer sets. Given these changes, we have extended our previous benchmarks to reflect the changes made on the method and its implementation. As such, we summarize our results in updated case studies that illustrate the benefits of our approach.

This chapter is organized as follows. After discussing the possible uses of ASP in systems biology in the current section, Section 7.3 briefly presents the ASP framework that is used later. Then, Section 7.4 presents the main definitions related to the AAN formalism and the specific properties on which we will focus later on, that is, stable states and attractors. Section 7.5 details our approach to define an AAN model using ASP rules and enumerate all stable states or attractors in such a model. The merits of such an approach are then emphasized in Section 7.6, where we give benchmarks of our methods on several models of different sizes (up to 100 components). Finally, Section 7.7 concludes and gives some perspectives to this work.

7.3. Basic Notions of Answer Set Programming

Answer Set Programming (ASP) is a purely declarative language based on the stable model semantics of logic programming [Baral, 2003]. As illustrated by the literature

study proposed in the previous section, ASP has proved to be efficient to address highly computational problems.

In this section, we will introduce the basic elements of ASP syntax and way of modeling problems. We will focus here on the parts that will be necessary to us to model biological systems in the following parts of this chapter.

7.3.1. Syntax and Rules

An answer set *program* is a finite set of *rules* of the form:

$$a_0 \leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n. \tag{7.1}$$

where $n \geq m \geq 0$, a_0 is an *atom* or $\bot$, all $a_1, \ldots, a_n$ are atoms, and the symbol "not" denotes *negation as failure*. The intuitive reading of such a rule is that whenever $a_1, \ldots, a_m$ are known to be true and there is no evidence for any of the negated atoms $a_{m+1}, \ldots, a_n$ to be true, then a_0 has to be true as well. An atom or a negated atom is also called a *literal*. The atom on the left of the arrow is called the *head* of the rule while the set of literals on the right is called its *body*. The order of the rules in the program, and of literals in a body, does not change its meaning.

Some special rules are noteworthy. A rule where $m = n = 0$ is called a *fact* and is useful to represent data because the head atom a_0 is always true. It is often written without the central arrow, as in rule (7.2). On the other hand, a rule where $n > 0$ and $a_0 = \bot$ is called a *constraint*. Because $\bot$ can never become true, if the body of a constraint is true, this invalidates the whole solution. Constraints are thus useful to filter out unwanted solutions when there are several candidate solutions. The symbol

$\perp$ is usually omitted in a constraint, as in rule (7.3).

$$a_0. \tag{7.2}$$

$$\leftarrow\ a_1,\ \ldots,\ a_m,\ not\ a_{m+1},\ \ldots,\ not\ a_n. \tag{7.3}$$

In the ASP paradigm, the search of solutions consists in computing the answer sets of a given program. Intuitively, the answer sets of a program are the minimal sets of atoms (in terms of set inclusion) that respect all rules of this program. Note that for a given program, there might be no, one or several answer sets.

Formally, an answer set for a given program is defined by Gelfond and Lifschitz [1988] as follows. An *interpretation* I is a finite set of propositional atoms. A rule r as given in (7.1) is *true under* I if and only if:

$$\{a_1,\ \ldots,\ a_m\} \subseteq I \wedge \{a_{m+1},\ \ldots,\ a_n\} \cap I = \emptyset \Rightarrow a_0 \in I\ .$$

An interpretation I is a *model* of a program P if each rule $r \in P$ is true under I. Finally, let P^I be the program obtained from P by deleting all rules that contain a negated atom appearing in I, and deleting all negated atoms from the remaining rules. I is an *answer set* (also called *solution* in the following) of P if I is a minimal model of P^I.

Example 7.1: *Consider the following program:*

$$a.$$

$$b\ \leftarrow\ a, not\ c.$$

$$c\ \leftarrow\ b, not\ a.$$

The atom a is given in a fact, and is thus always true. The atoms b and c depend on the presence of the other atoms. The unique solution (answer set) of this program is $\{a, b\}$. Indeed, it is the only minimal set of atoms that respects all rules.

Example 7.2: *Programs can yield no answer set, one answer set, or several answer sets. For example, the following program:*

$$b \leftarrow not\ c.$$

$$c \leftarrow not\ b.$$

produces two answer sets: $\{b\}$ and $\{c\}$. Indeed, the absence of c makes b true, and conversely absence of b makes c true.

Cardinality constructs, depicted by a set of atoms in curly braces, are another way to obtain multiple answer sets. The most usual way of using a cardinality is in the head:

$$l\ \{q_1,\ \ldots,\ q_k\}\ u\ \leftarrow\ a_1,\ \ldots,\ a_m,\ not\ a_{m+1},\ \ldots,\ not\ a_n.$$

where $k \geq 0$, l is an integer and u is an integer or the infinity (∞). Such a cardinality means that, under the condition that the body is satisfied, the answer set X must contain at least l and at most u atoms from the set $\{q_1, ..., q_m\}$, or, in other words: $l \leq |\{q_1, ..., q_m\} \cap X| \leq u$ where $\cap$ is the symbol of set intersection and $|\cdot|$ denotes the cardinality of a set. We note that several answer sets may match this definition, as there may be numerous solutions X to this equation. If they are not explicitly given, l defaults to 0 and u defaults to ∞.

Example 7.3: *Using a cardinality construct, the program of Example 7.2 can be*

summed up into the following program containing only one cardinality fact:

$$1\ \{b, c\}\ 1.$$

7.3.2. Predicates

Atoms in ASP are expressed as *predicates* with an arity, that is, they can apply to *terms* (also called *arguments*). For instance, let us consider the following program:

$$\mathrm{parentOf}(\mathrm{jenny}, \mathrm{charles}). \tag{7.4}$$

$$\mathrm{parentOf}(\mathrm{mary}, \mathrm{jenny}). \tag{7.5}$$

$$\mathrm{grandparentOf}(X, Z) \leftarrow \mathrm{parentOf}(X, Y), \mathrm{parentOf}(Y, Z). \tag{7.6}$$

This program first contains two facts in lines (7.4) and (7.5), stating that Jenny is a parent of Charles, and Mary is a parent of Jenny. This is expressed by predicate "$\mathrm{parentOf}$" with arity 2 (it has two arguments) and three constants "jenny", "$\mathrm{charles}$" and "mary". Then, line (7.6) expresses that a grandparent of someone is the parent of their parent. This is expressed by predicate "$\mathrm{grandparentOf}$" also of arity 2, but this time using variables (X, Y and Z). A variable can stand for any existing atom in the program. By convention, constant names start with a low letter or are surrounded by double quotes, and variable names start with a capital letter. Moreover, variables have a scope limited to the rule, meaning that if X was used in another rule, it would be considered another variable.

Solving an ASP program as explained above requires that it contains no variable; for this, a *grounding* step is first required, consisting in the removal of all free variables by replacing them by possible constants while preserving the meaning of the program. In the example above, the grounding step produces the following variable-

free program, where X, Y and Z are replaced by all possible constants:

$$\mathrm{parentOf}(\mathrm{jenny}, \mathrm{charles}). \quad \mathrm{parentOf}(\mathrm{mary}, \mathrm{jenny}).$$

$$\mathrm{grandparentOf}(\mathrm{mary}, \mathrm{mary}) \leftarrow \mathrm{parentOf}(\mathrm{mary}, \mathrm{mary}), \mathrm{parentOf}(\mathrm{mary}, \mathrm{mary}).$$

$$\mathrm{grandparentOf}(\mathrm{mary}, \mathrm{charles}) \leftarrow \mathrm{parentOf}(\mathrm{mary}, \mathrm{mary}), \mathrm{parentOf}(\mathrm{mary}, \mathrm{charles}).$$

$$\mathrm{grandparentOf}(\mathrm{mary}, \mathrm{charles}) \leftarrow \mathrm{parentOf}(\mathrm{mary}, \mathrm{jenny}), \mathrm{parentOf}(\mathrm{jenny}, \mathrm{charles}).$$

$$\mathrm{grandparentOf}(\mathrm{mary}, \mathrm{jenny}) \leftarrow \mathrm{parentOf}(\mathrm{mary}, \mathrm{charles}), \mathrm{parentOf}(\mathrm{charles}, \mathrm{jenny}).$$

$$\ldots \text{(23 more grounded rules)} \ldots$$

The two first rules in the grounded rules above are simply the facts; all the others come from the replacing of all variables in rule (7.6). Solving then consists in iteratively removing the rules whose body is false. The answer set is the set of head predicates of the remaining rules, that is:

$$\{ \quad \mathrm{parentOf}(\mathrm{jenny}, \mathrm{charles}),$$
$$\mathrm{parentOf}(\mathrm{mary}, \mathrm{jenny}),$$
$$\mathrm{grandparentOf}(\mathrm{mary}, \mathrm{charles}) \quad \} \ .$$

Finally, predicates are very useful in cardinality constructs as defined in the previous section: instead of explicitly giving all atoms in the cardinality set (by extension), it is possible to enumerate them based on predicates and variables (by comprehension). For instance:

$$0 \ \{\mathrm{f}(A, B) : \mathrm{g}(A)\} \ 2 \ \leftarrow \ \mathrm{h}(B).$$

means that for each value of B so that $\mathrm{h}(B)$ is true, all values of A so that $\mathrm{g}(A)$ is true are enumerated in order to create predicates $\mathrm{f}(A, B)$, and at most two are kept. Typically, this might create a number of answer sets which depends on the existing

predicates g and h.

The grounding and solving steps are usually tackled by specialized software. For the present chapter, we use Clingo[1] [Gebser et al., 2016] which is a combination of a grounder and a solver. In the rest of this chapter, we use ASP to tackle the problems of enumerating all stable states and attractors of a given AAN model.

7.3.3. Scripting

Since its version 5.0, Clingo offers scripting capabilities, using either Lua or Python. In both cases, this scripting allows to enrich the usual solving of an ASP program with:

- grounding and solving control,
- external predicates (provided by external data or software),
- on-the-fly modification of programs,
- post-processing of solutions.

For instance, in Python, this scripting allows to programmatically start the solving and interrupt it for each solution found to execute some processing on the result with Python. The introduction of a more classical imperative language into the solving has many advantages, such as allowing to easily compute some parts of the program that are trivially done in imperative scripting, but difficult in ASP.

The solution that we present in this work comes in three versions that make a progressive usage of this scripting:

[1] We used Clingo version 5.4: https://potassco.org/

1. The first version uses no scripting and thus outputs duplicate solutions that are detailed later.

2. In the second version, each answer set found by Clingo is provided to a Python script that checks if it is a duplicate or not, and outputs it only if it is a genuine solution; this solution uses scripting as a mere post-processing tool.

3. In the third version, for each answer set found by Clingo, a constraint is added to the ASP program in order to avoid enumerating a duplicate solution; this solution uses on-the-fly modification of the ASP program for the same result.

7.4. Dynamic Modeling Using Asynchronous Automata Networks

This section introduces the formal definitions required to model and analyze Asynchronous Automata Networks (AANs).

7.4.1. Motivation: Using ASP to Analyze the Dynamics

In order to study the dynamics of a system, one of the main approaches is to model it in a formal way. This allows to understand or control the behavior of this system, by checking properties on its model.

The modeling process can be considered as a particular form of operationalization of the knowledge representation. ASP logic programs follow the “generate and test” strategy. This strategy includes four steps:

- enumerate with facts,
- explain and extend with rules,
- generate all possibilities with cardinalities, and finally
- filter with constraints.

In the rest of this chapter, we propose ASP programs whose semantics and rule syntax are based on those presented here. The idea of using ASP to analyze the dynamics of a discrete model has its origin in the combinatorial explosion of the number of states. Since ASP was designed to demonstrate a certain efficiency in enumerating sets satisfying a set of rules, this framework seems appropriate to study some dynamic properties of interest. In the following, we will focus on one type of dynamic model: Asynchronous Automata Networks (AANs), that we introduce formally in this section. The conversion of such an AAN model into ASP is the subject of Section 7.5.

7.4.2. Definition of Asynchronous Automata networks

Asynchronous Automata Networks (AANs) are a recently introduced formalism allowing to model discrete dynamical systems. Although mostly used to model Biological Regulatory Networks (BRNs), AANs are general-purpose and could be applied to other fields. AAN are conveniently encoded into ASP because of their simple form that can be straightforwardly described in terms of a few simple rules. Its two most used update semantics are also easily represented in ASP (as explained later in Section 7.5.3). Moreover, AANs have been chosen for this chapter as they allow to encompass more widespread frameworks such as René Thomas modeling [Thomas,

1973] and Boolean networks [Kauffman, 1969]. The results of this chapter are thus applicable to these frameworks too.

Definition 7.1 introduces the formalism of AANs [Folschette et al., 2015] which allows to model a finite number of discrete levels, called *local states*, into several *automata*. A local state is denoted a_i, where a is the name of the automaton, corresponding usually to a biological component, and i is a level identifier within a. At any time, exactly one local state of each automaton is *active*, modeling the current level of activity of this automaton or, equivalently, its internal state. The set of all active local states at a given time is called the *global state* of the network. Figure 7.1 depicts the structure of a simple AAN, which is more thoroughly detailed later.

The possible local evolutions inside an automaton are defined by *local transitions*. A local transition is a triple noted $a_i \xrightarrow{\ell} a_j$ and is responsible, inside a given automaton a, for the change of the active local state (a_i) to another local state (a_j), conditioned by a set ℓ of local states belonging to other automata. Such a local transition is *playable* if and only if a_i and all local states in the set ℓ are active. Thus, it can be read as "all the local states in ℓ can cooperate to change the active local state of a by making it switch from a_i to a_j". It is required that a_i and a_j are two different local states in automaton a and that ℓ contains no local state of a. We also note that ℓ should contain at most one local state per automaton, otherwise the local transition is unplayable. Conversely, if ℓ is empty, the transition is playable if and only if a_i is active.

Definition 7.1*(Asynchronous Automata Network)*: An *Asynchronous Automata Network* is a triple $(\Sigma, \mathcal{S}, \mathcal{T})$ where:

- $\Sigma = \{a, b, \dots\}$ is the finite set of *automata* identifiers;
- For each $a \in \Sigma$, $\mathrm{b}(a) \in \mathbb{N} \setminus \{0\}$ denotes the upper bound of automaton a

and $\mathcal{S}_a = \{a_0, \ldots, a_{\mathrm{b}(a)}\}$ is the finite set of *local states* of automaton a; $\mathcal{S} = \prod_{a\in\Sigma} \mathcal{S}_a$ is the finite set of *global states*; $\mathcal{LS} = \cup_{a\in\Sigma}\mathcal{S}_a$ denotes the set of all the local states.

- For each $a \in \Sigma$, $\mathcal{T}_a \subset \{a_i \xrightarrow{\ell} a_j \in \mathcal{S}_a \times \wp(\mathcal{LS}\setminus\mathcal{S}_a) \times \mathcal{S}_a \mid a_i \neq a_j\}$ is the set of *local transitions* on automaton a, where $\wp$ denotes the power set; $\mathcal{T} = \bigcup_{a\in\Sigma} \mathcal{T}_a$ is the set of all local transitions in the model.

For a given local transition $\tau = a_i \xrightarrow{\ell} a_j$, a_i is called the *origin* or τ, ℓ the *condition* and a_j the *destination*, and they are respectively noted $\mathsf{ori}(\tau)$, $\mathsf{cond}(\tau)$ and $\mathsf{dest}(\tau)$.

Example 7.4: *Figure 7.1 represents an AAN $(\Sigma, \mathcal{S}, \mathcal{T})$ with 4 automata (among which two contain 2 local states and the two others contain 3 local states) and 12 local transitions:*

- $\Sigma = \{a, b, c, d\}$,
- $\mathrm{b}(a) = 1$, $\mathrm{b}(b) = 2$, $\mathrm{b}(c) = 1$, $\mathrm{b}(d) = 2$,
- $\mathcal{S}_a = \{a_0, a_1\}$, $\mathcal{S}_b = \{b_0, b_1, b_2\}$, $\mathcal{S}_c = \{c_0, c_1\}$, $\mathcal{S}_d = \{d_0, d_1, d_2\}$,
- $\mathcal{T} = \{\, a_0 \xrightarrow{\{c_1\}} a_1, a_1 \xrightarrow{\{b_2\}} a_0, \quad b_0 \xrightarrow{\{d_0\}} b_1, b_0 \xrightarrow{\{a_1,c_1\}} b_2, b_1 \xrightarrow{\{d_1\}} b_2, b_2 \xrightarrow{\{c_0\}} b_0,$
  $c_0 \xrightarrow{\{a_1,b_0\}} c_1, c_1 \xrightarrow{\{d_2\}} c_0, \quad d_0 \xrightarrow{\{b_2\}} d_1, d_0 \xrightarrow{\{a_0,b_1\}} d_2, d_1 \xrightarrow{\{a_1\}} d_0, d_2 \xrightarrow{\{c_0\}} d_0 \,\}.$

The local transitions given in Definition 7.1 define concurrent interactions between automata. They have by nature only local consequences, but they can be combined in different ways to create a global dynamics for the model. When such a way of combining local transitions, also called *semantics*, is decided, it becomes possible to compute *global transitions* between global states. In the following, we will only

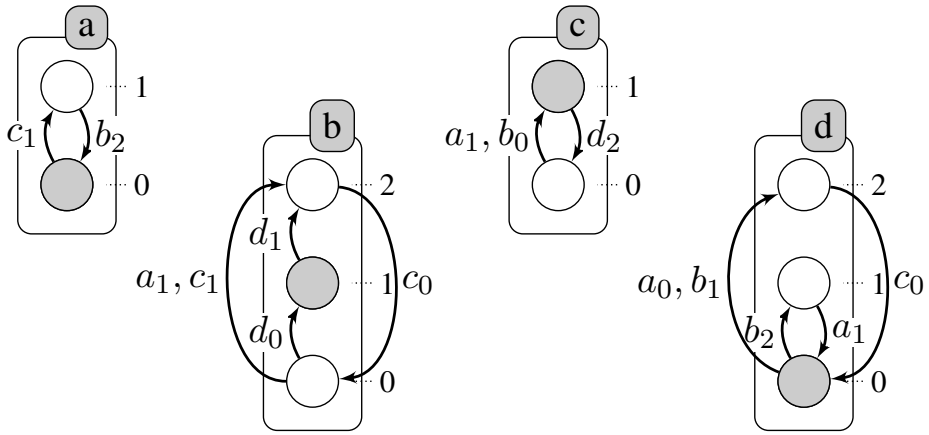

Figure 7.1: An example of an AAN model with 4 automata: a, b, c and d. Each box represents an automaton (modeling a biological component), circles represent their local states (corresponding to their discrete expression levels) and the local transitions are represented by arrows labeled by their necessary conditions (consisting of a set of local states from other automata). The automata a and c are either at level 0 or 1, and b and d have 3 levels (0, 1 and 2). The grayed local states stand for the global state $\langle a_0, b_1, c_1, d_0 \rangle$.

focus on the (purely) asynchronous and (purely) synchronous semantics, which are the most widespread in the literature. The choice of such a semantics mainly depends on the considered biological phenomena modeled and the mathematical abstractions chosen by the modeler.

7.4.3. Semantics and Dynamics of Asynchronous Automata Networks

As explained in the previous section, a global state of an AAN is a set of local states of automata, containing exactly one local state for each automaton. In the following, we give some notations related to global states, then we define the global dynamics of an AAN.

The active local state of a given automaton $a \in \Sigma$ in a global state $\zeta \in \mathcal{S}$ is noted $\zeta[a]$. For any given local state $a_i \in \mathcal{LS}$, we also note: $a_i \in \zeta$ if and only if $\zeta[a] = a_i$. For a given set of local states $X \subseteq \mathcal{LS}$, we extend this notation to $X \subseteq \zeta$ if and only if $\forall a_i \in X, a_i \in \zeta$, meaning that all local states of X are active in ζ.

Furthermore, for any given local state $a_i \in \mathcal{LS}$, $\zeta \Cap a_i$ represents the global state that is identical to ζ, except for the local state of a which is substituted with a_i: $(\zeta \Cap a_i)[a] = a_i \wedge \forall b \in \Sigma \setminus \{a\}, (\zeta \Cap a_i)[b] = \zeta[b]$. We generalize this notation to a set of local states $X \subseteq \mathcal{LS}$ containing at most one local state per automaton, that is, $\forall a \in \Sigma, |X \cap \mathcal{S}_a| \leq 1$ where $|\cdot|$ denotes the cardinality of a set; in this case, $\zeta \Cap X$ is the global state ζ where the local state of each automaton has been replaced by the local state of the same automaton in X, if it exists: $\forall a \in \Sigma, (X \cap \mathcal{S}_a = \{a_i\} \Rightarrow (\zeta \Cap X)[a] = a_i) \wedge (X \cap \mathcal{S}_a = \emptyset \Rightarrow (\zeta \Cap X)[a] = \zeta[a])$.

In Definition 7.2, we formalize the notion of playability of a local transition which was informally presented in the previous section. Playable local transitions are not necessarily used as such, but combined depending on the chosen semantics, which is the subject of the rest of the section.

Definition 7.2*(Playable Local Transitions)*: Let $\mathcal{AAN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous Automata Network and $\zeta \in \mathcal{S}$ a global state. The set of playable local transitions in ζ is called P_ζ and defined by: $P_\zeta = \{a_i \xrightarrow{\ell} a_j \in \mathcal{T} \mid \ell \subseteq \zeta \wedge a_i \in \zeta\}$.

The dynamics of the AAN is a composition of global transitions between global states, that consist in selecting and playing a set of local transitions. Such sets are different depending on the chosen semantics. In the following, we give the definition of the asynchronous and synchronous semantics by characterizing the sets of local transitions that can be "played" as global transitions. The sets of the asynchronous semantics (Definition 7.3) are made of exactly one playable local transition; thus,

a global asynchronous transition changes the local state of exactly one automaton. On the other hand, the sets of the synchronous semantics (Definition 7.4) consist of exactly one playable local transition for each automaton, except for the automata where no local transition is playable; in other words, a global synchronous transition changes the local state of all automata that can evolve at a time. The empty set is not a valid set of local transitions for both semantics, meaning that they both cannot produce a global transition that changes no automaton (also known as self-transition).

Definition 7.3*(Asynchronous semantics)*: Let $\mathcal{AAN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous Automata Network and $\zeta \in \mathcal{S}$ a global state. The set of global transitions playable in ζ for the *asynchronous semantics* is given by:

$$\mathcal{U}^{\mathsf{asyn}}(\zeta) = \{\{a_i \xrightarrow{\ell} a_j\} \mid a_i \xrightarrow{\ell} a_j \in P_\zeta\}.$$

Definition 7.4*(Synchronous semantics)*: Let $\mathcal{AAN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous Automata Network and $\zeta \in \mathcal{S}$ a global state. The set of global transitions playable in ζ for the *synchronous* semantics is given by:

$$\mathcal{U}^{\mathsf{syn}}(\zeta) = \{u \subseteq \mathcal{T} \mid u \neq \emptyset \wedge \forall a \in \Sigma, (P_\zeta \cap \mathcal{T}_a = \emptyset \Rightarrow u \cap \mathcal{T}_a = \emptyset) \wedge (P_\zeta \cap \mathcal{T}_a \neq \emptyset \Rightarrow |u \cap \mathcal{T}_a| = 1)\}.$$

Once a semantics has been chosen, it is possible to compute the corresponding dynamics of a given AAN. Thus, in the following, when it is not ambiguous and when results apply to both of them, we will denote by $\mathcal{U}$ a chosen semantics among $\mathcal{U}^{\mathsf{asyn}}$ and $\mathcal{U}^{\mathsf{syn}}$. Definition 7.5 formalizes the notion of a global transition depending on a chosen semantics $\mathcal{U}$.

Definition 7.5*(Global Transition)*: Let $\mathcal{AAN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous

Automata Network, $\zeta_1, \zeta_2 \in \mathcal{S}$ two states and $\mathcal{U} \in \{\mathcal{U}^{\mathsf{asyn}}, \mathcal{U}^{\mathsf{syn}}\}$ a semantics. The *global transition* relation between two states ζ_1 and ζ_2 for the semantics $\mathcal{U}$, noted $\zeta_1 \rightarrow_{\mathcal{U}} \zeta_2$, is defined by:

$$\zeta_1 \rightarrow_{\mathcal{U}} \zeta_2 \iff \exists u \in \mathcal{U}(\zeta_1) \wedge \zeta_2 = \zeta_1 \Cap \{\mathsf{dest}(\tau) \in \mathcal{LS} \mid \tau \in u\}.$$

The state ζ_2 is called a *successor* of ζ_1.

Example 7.5: *Figures 7.2 and 7.3 illustrate respectively the asynchronous and synchronous semantics on the model of Figure 7.1. Each global transition is depicted by an arrow between two global states. Only an interesting subset of the whole dynamics is depicted in both figures.*

A semantics is called *deterministic* if it makes each global state to have at most one successor. However, in general, semantics are *non-deterministic*: each global state can have several successors. Indeed, the two semantics studied here are non-deterministic in general (some particular models may not show the non-determinism).

In the case of the asynchronous semantics, the non-determinism may come from concurrent local transitions, but it actually mainly comes from the fact that exactly one local transition is taken into account for each global transition (see Definition 7.3). Thus, for a given state $\zeta \in \mathcal{S}$, as soon as $|P_\zeta| > 1$, several successors may exist. In the model of Figure 7.1, for example, the global state $\langle a_1, b_2, c_0, d_1\rangle$ (in green on Figure 7.2) has three successors because: $\langle a_1, b_2, c_0, d_1\rangle \rightarrow_{\mathcal{U}^{\mathsf{asyn}}} \langle a_0, b_2, c_0, d_1\rangle$, $\langle a_1, b_2, c_0, d_1\rangle \rightarrow_{\mathcal{U}^{\mathsf{asyn}}} \langle a_1, b_0, c_0, d_1\rangle$ and $\langle a_1, b_2, c_0, d_1\rangle \rightarrow_{\mathcal{U}^{\mathsf{asyn}}} \langle a_1, b_2, c_0, d_0\rangle$.

In the case of the synchronous semantics (see Definition 7.4), however, the non-determinism on the global scale is only generated by local transitions that create non-determinism inside an automaton, that is, local transitions that have the same

origin, are together playable in at least one global state, but have different destinations. For example, the model of Figure 7.1 features two local transitions $b_0 \xrightarrow{\{d_0\}} b_1$ and $b_0 \xrightarrow{\{a_1,c_1\}} b_2$ that can produce the two following global transitions from the same state (depicted by red arrows on Figure 7.3): $\langle a_1, b_0, c_1, d_0\rangle \rightarrow_{\mathcal{U}^{\mathsf{syn}}} \langle a_1, b_1, c_1, d_0\rangle$ and $\langle a_1, b_0, c_1, d_0\rangle \rightarrow_{\mathcal{U}^{\mathsf{syn}}} \langle a_1, b_2, c_1, d_0\rangle$. Note that for this particular case, these transitions also exist for the asynchronous semantics (also depicted by red arrows on Figure 7.2).

Finally, Definition 7.6 introduces the notions of *path* and *trace* which are used to characterize a set of successive global states with respect to a global transition relation. Paths are useful for the characterization of attractors that are the topic of this work. The trace is the set of all global states traversed by a given path (thus disregarding the order in which they are visited). Thus, a path is a sequence while a trace is a set.

Definition 7.6*(Path and Trace)*: Let $\mathcal{AAN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous Automata Network, $\mathcal{U}$ a semantics and $n \in \mathbb{N} \setminus \{0\}$ a strictly positive integer. A sequence $\mathbf{H} = (\mathbf{H}_i)_{i\in[\![0;n]\!]} \in \mathcal{S}^{n+1}$ of global states is a *path of length* n if and only if: $\forall i \in [\![0; n-1]\!], \mathbf{H}_i \rightarrow_{\mathcal{U}} \mathbf{H}_{i+1}$. $\mathbf{H}$ is also simply called a *path* if its length n is not known or relevant. The *trace* of $\mathbf{H}$ is the set of global states it contains: $\mathsf{trace}(\mathbf{H}) = \{\mathbf{H}_j \in \mathcal{S} \mid j \in [\![0; n]\!]\}$.

In the following, when we define a path $\mathbf{H}$ of length n, we use the notation $\mathbf{H}_i$ to denote the i^{th} element in the sequence $\mathbf{H}$, with $i \in [\![0; n]\!]$. We also use the notation $|\mathbf{H}| = n$ to denote the length of a path $\mathbf{H}$, allowing to write: $\mathbf{H}_{|H|}$ to refer to its last element. $\mathbf{H}$ is said to *start from* a given global state $\zeta \in \mathcal{S}$ if and only if $\mathbf{H}_0 = \zeta$; it is said to *end in* a given global state $\zeta' \in \mathcal{S}$ if and only if $\mathbf{H}_n = \zeta'$. Finally, we note that a path of length n models the succession of n global transitions, and thus features at

most $n+1$ states: $\mathsf{trace}(\mathbf{H}) \leq n+1$. When at least one state is visited more than once in $\mathbf{H}$, the inequality becomes strict.

Example 7.6: *The following sequence is a path of length 6 for the asynchronous semantics:*

$$\mathbf{H} = (\langle a_1, b_2, c_1, d_1\rangle; \langle a_0, b_2, c_1, d_1\rangle; \langle a_1, b_2, c_1, d_1\rangle; \langle a_1, b_2, c_1, d_0\rangle;$$
$$\langle a_0, b_2, c_1, d_0\rangle; \langle a_0, b_2, c_1, d_1\rangle; \langle a_1, b_2, c_1, d_1\rangle)$$

We have: $\mathsf{trace}(\mathbf{H}) = \{\langle a_1, b_2, c_1, d_1\rangle, \langle a_0, b_2, c_1, d_1\rangle, \langle a_1, b_2, c_1, d_0\rangle, \langle a_0, b_2, c_1, d_0\rangle\}$. *Although* $|\mathbf{H}| = 6$*, we note that* $|\mathsf{trace}(\mathbf{H})| = 4$ *because* $\mathbf{H}_0 = \mathbf{H}_2 = \mathbf{H}_6$ *and* $\mathbf{H}_1 = \mathbf{H}_5$.

7.4.4. Stable States and Attractors in Asynchronous Automata Networks

Studying the dynamics of biological networks was the focus of many works, explaining the diversity of existing frameworks dedicated to modeling and the different methods developed in order to identify some patterns, especially attractors [Skodawessely and Klemm, 2011, Zhang et al., 2007, Mushthofa et al., 2014, Akutsu et al., 2012, Berntenis and Ebeling, 2013]. In this chapter we focus on several sub-problems related to this: we seek to identify the *stable states* and the *attractors* of a given network. The stable states and the attractors are the two long-term structures in which any dynamics eventually falls into. Indeed, they consist in terminal (sets of) global states that cannot be escaped, and in which the dynamics always ends. In the following, we formally define these dynamical properties.

A *stable state* (sometimes called *fixed point*) is a global state which has no successor, as given in Definition 7.7. The existence of several of these states is called multistability, and implies bifurcations in the dynamics [Wuensche, 1998]. At this point, it is important to remind that the empty set never belongs to the semantics defined above: $\forall \zeta \in \mathcal{S}, \emptyset \notin \mathcal{U}^{\mathsf{asyn}}(\zeta) \wedge \emptyset \notin \mathcal{U}^{\mathsf{syn}}(\zeta)$. The consequence on the dynamics is that a global state can never be its own successor. In other words, even when no local transition can be played in a given global state (i.e., $P_\zeta = \emptyset$), we do not add a self-transition on this state. Instead, this state has no successor and is thus structurally a sink node in the state-transition graph.

Definition 7.7*(Stable state)*: Let $\mathcal{AN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous Automata Network, and $\mathcal{U} \in \{\mathcal{U}^{\mathsf{asyn}}, \mathcal{U}^{\mathsf{syn}}\}$ be a semantics. A global state $\zeta \in \mathcal{S}$ is called a *stable state* if and only if no global transition can be played in this state:

$$\mathcal{U}(\zeta) = \emptyset.$$

It is interesting to note that the set of stable states of a model is the same in both the asynchronous and the synchronous semantics [Klarner et al., 2015, Moisset de Espanés et al., 2016]: $\forall \zeta \in \mathcal{S}, \mathcal{U}^{\mathsf{asyn}}(\zeta) = \emptyset \iff \mathcal{U}^{\mathsf{syn}}(\zeta) = \emptyset$. This comes from the fact that both semantics rely on the same definition of set of playable transitions.

Example 7.7: *The state-transition graphs of Figures 7.2 and 7.3 depict three stable states colored in red: $\langle a_1, b_1, c_1, d_0\rangle$, $\langle a_1, b_1, c_0, d_0\rangle$ and $\langle a_0, b_0, c_0, d_1\rangle$. Visually, they can be easily recognized because they have no outgoing arrows (meaning that they have no successors). Although these figures do not represent the whole dynamics, they allow to check that in both semantics, the stable states are shared. Actually, the model contains no other stable state than the ones depicted in these figures.*

Another complementary dynamical pattern consists in the notion of *trap domain* (Definition 7.8), which is a non-empty set of states that the dynamics cannot escape, and thus in which the system indefinitely remains. Relying on this, an *attractor* (Definition 7.9) is a minimal trap domain in terms of set inclusion. In this work, we focus more precisely on *non-singleton attractors*, that is, attractors made of at least two states. Formally speaking, stable states are trap domains of size 1, and could thus be considered attractors. However, in the scope of this chapter and for the sake of clarity, in the following, we call "attractors" only non-singleton attractors. This is justified by the very different approaches developed to enumerate stable states and attractors in the next sections.

Definition 7.8*(Trap Domain)*: Let $\mathcal{AAN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous Automata Network and $\mathcal{U}$ a semantics. A set of global states $T \subseteq \mathcal{S}$, with $T \neq \emptyset$, is called a *trap domain* (regarding a semantics $\mathcal{U}$) if and only if the successors of each of its elements are also in T:

$$\forall \zeta_1 \in T \wedge \forall \zeta_2 \in \mathcal{S}, \zeta_1 \rightarrow_{\mathcal{U}} \zeta_2 \Rightarrow \zeta_2 \in T.$$

Definition 7.9*(Attractor)*: Let $\mathcal{AAN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous Automata Network and $\mathcal{U}$ a semantics. A set of global states $A \subseteq \mathcal{S}$, with $|A| \geq 2$, is called an *attractor* (regarding semantics $\mathcal{U}$) if and only if it is a minimal trap domain in terms of set inclusion.

The previous definition is hard to encode into ASP, due to the notion of minimality. In the following, we use the notion of *cycle* (Definition 7.10), which is a looping path, in order to give an alternative definition for an attractor (Lemma 7.3).

Definition 7.10*(Cycle)*: Let $\mathcal{AAN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous Automata

Network, $\mathcal{U}$ a semantics and $\mathbf{C}$ a path for this semantics. $\mathbf{C}$ is called a *cycle* (regarding the semantics $\mathcal{U}$) if and only if it starts from and ends in the same state:

$$\mathbf{C}_0 = \mathbf{C}_{|\mathbf{C}|} \ .$$

Example 7.8: *The path* $\mathbf{H}$ *of length 6 given in Example 7.6 is a cycle because* $\mathbf{H}_0 = \mathbf{H}_6$.

Definition 7.11 recalls the definition of a *strongly connected subgraph* (SCSG) in the scope of the dynamics of an AAN: it is a subgraph of the global state-transition graph in which there exists a path between any pair of states. Note that, because we consider here only the synchronous and asynchronous semantics, a subgraph made of exactly one state ζ cannot be a SCSG, because there never exists a self-transition $\zeta \to \zeta$.

The definition of SCSG is then used in two lemmas that later allow to give an alternative definition of an attractor. Lemma 7.1 states that the set of (traces of) cycles in a model is exactly the set of strongly connected subgraphs. Indeed, a cycle allows to "loop" between all states that it contains, and conversely, such a cycle can be built from the states of any strongly connected subgraph. Lemma 7.2 states that any attractor is also a SCSG. This well-known result comes from the minimality of an attractor and implies that an attractor is always made of one or several loops.

Definition 7.11*(Strongly Connected Subgraph)*: Let $\mathcal{AAN} = (\Sigma, \mathcal{S}, \mathcal{T})$ be an Asynchronous Automata Network, $\mathcal{U}$ a semantics and $G \subseteq \mathcal{S}$ a set of states. G is a *strongly connected subgraph* (regarding the semantics $\mathcal{U}$) or *SCSG* for short, if and only if for all pairs of states $(\zeta_1, \zeta_2) \in G^2$ in this set, there exists a path $\mathbf{H}$ made of

states in G that starts from ζ_1 and ends in ζ_2, that is:

$$\mathbf{H}_0 = \zeta_1 \wedge \mathbf{H}_{|\mathbf{H}|} = \zeta_2 \wedge \forall \zeta \in \mathsf{trace}(\mathbf{H}), \zeta \in G \ .$$

Lemma 7.1*(The Traces of Cycles are the SCSGs)*: The traces of the cycles are exactly the strongly connected subgraphs.

Proof. ($\Rightarrow$) From any state of a cycle, it is possible to reach all the other states (by possibly cycling). Therefore, the trace of this cycle is a strongly connected subgraph. ($\Leftarrow$) Let G be a strongly connected subgraph. Consider, without loss of generality, an arbitrary state $\zeta_0 \in G$. For each other state $\zeta' \in G \setminus \{\zeta_0\}$, by definition of a SCSG, there exists two paths $\mathbf{J}^{\zeta'}$ and $\mathbf{K}^{\zeta'}$ made of states of G so that: $\mathbf{J}^{\zeta'}_0 = \zeta_0$, $\mathbf{J}^{\zeta'}_{|\mathbf{J}^{\zeta'}|} = \zeta'$, $\mathbf{K}^{\zeta'}_0 = \zeta'$ and $\mathbf{K}^{\zeta'}_{|\mathbf{K}^{\zeta'}|} = \zeta_0$. We denote $\mathbf{H}^{\zeta'}$ the concatenation of $\mathbf{J}^{\zeta'}$ and $\mathbf{K}^{\zeta'}$ which is possible because $\mathbf{J}^{\zeta'}_{|\mathbf{J}^{\zeta'}|} = \zeta' = \mathbf{K}^{\zeta'}_0$. $\mathbf{H}^{\zeta'}$ is a cycle because $\mathbf{H}^{\zeta'}_0 = \mathbf{J}^{\zeta'}_0 = \zeta_0 = \mathbf{K}^{\zeta'}_{|\mathbf{K}^{\zeta'}|} = \mathbf{H}^{\zeta'}_{|\mathbf{H}^{\zeta'}|}$. Moreover, $\mathbf{H}^{\zeta'}$ is only made of states of G. By considering, without loss of generality, an arbitrary sequence $A = (\zeta'_i)_{i \in [\![1;|G\setminus\{\zeta_0\}|]\!]}$ of all the states in $G \setminus \{\zeta_0\}$, that is: $\forall \zeta' \in G \setminus \{\zeta_0\}, \exists i \in [\![1; |G \setminus \{\zeta_0\}|]\!], A_i = \zeta_0$, then we can build a path $\mathbf{H}$ by concatenating $\mathbf{H}^{A_0}$, $\mathbf{H}^{A_1}$, ..., $\mathbf{H}^{A_{|A|}}$. $\mathbf{H}$ is a cycle because $\mathbf{H}_0 = \mathbf{H}_{|\mathbf{H}|} = \zeta_0$. Moreover, $\mathsf{trace}(\mathbf{H}) \subseteq G$ because $\mathbf{H}$ is only made of states in G, and $G \subseteq \mathsf{trace}(\mathbf{H})$ because for each state in G, at least one sub-cycle of $\mathbf{H}$ contains it. Therefore, $\mathbf{H}$ is a cycle so that $\mathsf{trace}(\mathbf{H}) = G$. □

Lemma 7.2*(An Attractor is a SCSG)*: An attractor is a strongly connected subgraph.

Proof. Let A be an attractor. Suppose that A is not a SCSG in order to make a proof by contradiction. Then there exists $\zeta_1, \zeta_2 \in A$ so that ζ_2 cannot be reached from ζ_1. Let X be the set containing exactly ζ_1 and all states that can be reached from

ζ_1. Obviously, $X \neq \emptyset$. By definition of an attractor (Definition 7.9), A is also a trap domain; therefore, the successors of each state in A are also in A, which recursively gives $X \subset A$ because $\zeta_1 \in A$. However, $\zeta_2 \notin X$ by definition of ζ_2 and X, thus $X \subsetneq A$. X is a trap domain by construction (if a state can be reached from ζ_1, then it belongs to X). This contradicts the fact that A is a minimal trap domain in terms of set inclusion (Definition 7.9). To conclude, A is a SCSG. □

In Definition 7.9, attractors are characterized in the classical way, that is, as minimal trap domains. However, we use an alternative characterization of attractors in this chapter, due to the specifics of ASP: Lemma 7.3 states that an attractor can alternatively be defined as a trap domain that is also a cycle, and conversely. In other words, the minimality requirement is replaced by a cyclicity requirement.

Lemma 7.3*(The Attractors are the Cyclic Trap Domains)*: The attractors are exactly the traces of cycles that are trap domains.

Proof. ($\Rightarrow$) By Definition 7.9, an attractor is a trap domain. From Lemma 7.2, it is also a strongly connected subgraph, and thus, from Lemma 7.1, it is the trace of a cycle. ($\Leftarrow$) Let $\mathbf{C}$ be a cycle which trace is a trap domain. From Lemma 7.1, $\mathbf{C}$ is also a strongly connected subgraph. Let us prove by contradiction that $\mathbf{C}$ is a minimal trap domain, by assuming that it is not minimal. This means that there exists a smaller trap domain $\mathbf{D} \subsetneq \mathbf{C}$. Let us consider $\zeta_1 \in \mathbf{D}$ and $\zeta_2 \in \mathbf{C} \setminus \mathbf{D}$. Because $\mathbf{D}$ is a trap domain, there exists no path between ζ_1 and ζ_2; this is in contradiction with $\mathbf{C}$ being a strongly connected subgraph (as both ζ_1 and ζ_2 belong to $\mathbf{C}$). Therefore, $\mathbf{C}$ is a minimal trap domain, and thus an attractor. □

As explained before, Lemma 7.3 will be used in Section 7.5.3 to enumerate attractors. Indeed, directly searching for minimal trap domains would be too cumbersome; instead, we enumerate cycles of length n in the dynamics of the model and filter out

those that are not trap domains.

Example 7.9: *The state-transition graphs of Figures 7.2 and 7.3 feature different attractors:*

- $\{\langle a_0, b_1, c_0, d_0\rangle, \langle a_0, b_1, c_0, d_2\rangle\}$ *is depicted in blue and appears in both figures. It is a simple attractor, because it contains a unique cycle.*
- $\{\langle a_0, b_2, c_1, d_0\rangle, \langle a_0, b_2, c_1, d_1\rangle, \langle a_1, b_2, c_1, d_1\rangle, \langle a_1, b_2, c_1, d_0\rangle\}$ *is only present for the asynchronous semantics and is depicted in yellow on Figure 7.2. It is a complex attractor, that is, a composition of several cycles.*
- $\{\langle a_1, b_2, c_1, d_1\rangle, \langle a_0, b_2, c_1, d_0\rangle\}$ *is, on the contrary, only present for the synchronous semantics and is depicted in gray on Figure 7.3. It is also a simple attractor.*

For each of these attractors, the reader can check that they can be characterized as cycles that are trap domains. For instance, the second attractor can be found by considering the following cycle:

$$\mathbf{A} = (\langle a_0, b_2, c_1, d_0\rangle; \langle a_0, b_2, c_1, d_1\rangle; \langle a_1, b_2, c_1, d_1\rangle; \langle a_1, b_2, c_1, d_0\rangle; \langle a_0, b_2, c_1, d_0\rangle)$$

and checking that its trace is a trap domain (which is visually confirmed in Figure 7.2 by the absence of outgoing arrows from any of the yellow states).

On the other hand, the following cycle is not an attractor:

$$\mathbf{C} = (\langle a_1, b_2, c_0, d_1\rangle; \langle a_1, b_2, c_0, d_0\rangle; \langle a_1, b_2, c_0, d_1\rangle).$$

Indeed, although it is a cycle, it features states having outgoing transitions (such as, for instance, transition $\langle a_1, b_2, c_0, d_0\rangle \rightarrow_{\mathcal{U}^{\mathsf{asyn}}} \langle a_0, b_2, c_0, d_0\rangle$*) and thus is not a trap*

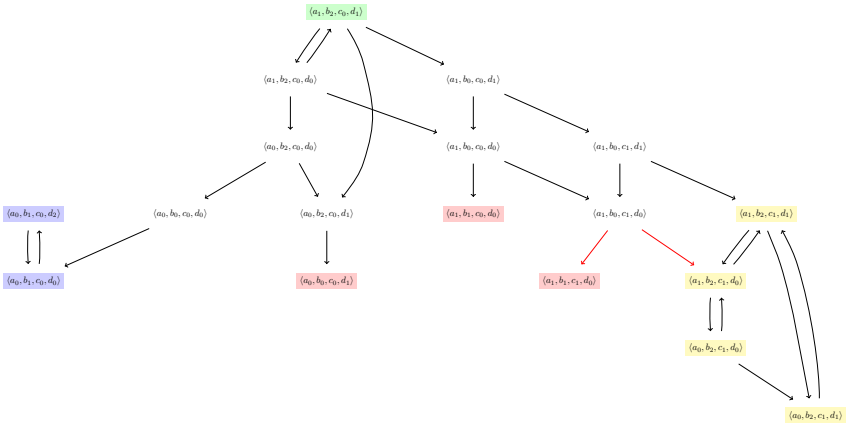

Figure 7.2: A part of the state-transition graph of the AAN given in Figure 7.1 for the **asynchronous** semantics, computed from the initial state: $\langle a_1, b_2, c_0, d_1 \rangle$ until reaching attractors. We can observe three stable states: $\langle a_1, b_1, c_1, d_0 \rangle$, $\langle a_1, b_1, c_0, d_0 \rangle$ and $\langle a_0, b_0, c_0, d_1 \rangle$; an attractor of size 2: $\{\langle a_0, b_1, c_0, d_0 \rangle, \langle a_0, b_1, c_0, d_2 \rangle\}$ (in blue) and an attractor of size 4: $\{\langle a_1, b_2, c_1, d_1 \rangle, \langle a_0, b_2, c_1, d_1 \rangle, \langle a_0, b_2, c_1, d_0 \rangle, \langle a_1, b_2, c_1, d_0 \rangle\}$ (in yellow).

domain.

The aim of the rest of this chapter is to tackle the enumeration of stable states (Section 7.5.2) and attractors (Section 7.5.3) in an AAN using ASP.

7.5. Encoding into Answer Set Programming

This section presents the three facets of this work in terms of ASP:

1. How to encode an AAN model to use with the two other facets (Section 7.5.1),

2. How to encode the stable state enumeration (Section 7.5.2),

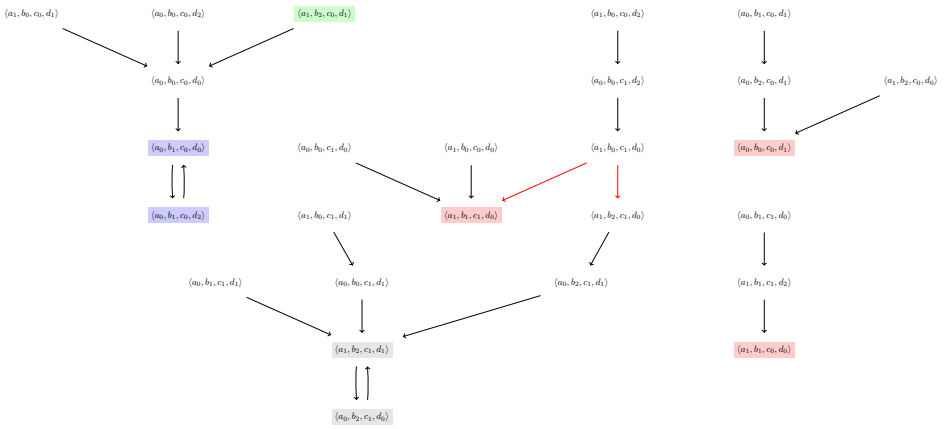

Figure 7.3: A part of the state-transition graph of the AAN given in Figure 7.1 for the **synchronous** semantics, computed from several initial states, such as $\langle a_1, b_2, c_0, d_1 \rangle$, until reaching attractors. It features non-deterministic global transitions, depicted by the two red arrows. We can observe the same three stable states than for the asynchronous semantics of Figure 7.2, but instead two attractors of size 2: $\{\langle a_0, b_1, c_0, d_0 \rangle, \langle a_0, b_1, c_0, d_2 \rangle\}$ (in blue) and $\{\langle a_1, b_2, c_1, d_1 \rangle, \langle a_0, b_2, c_1, d_0 \rangle\}$ (in gray).

3. How to encode the attractor enumeration (Section 7.5.3).

All parts were initially presented in [Ben Abdallah et al., 2015, 2017]. The first two are identical, while the last one has been significantly improved by fixing mistakes and adding filtering methods to avoid redundant answers in attractor enumeration. Indeed, a straightforward approach for this problem, like in [Ben Abdallah et al., 2017], produces a lot of duplicate solutions because of the enumeration method, as explained at the end of Section 7.5.3.2. In this chapter, we propose two additions based on the combination of Python scripting with ASP to solve this issue.

All code presented in this section is available as free software online[2]. The ASP scripts are exactly the lines of code presented below, where the symbol $\leftarrow$ in rules is

[2]All programs and benchmarks are available as additional files and at: https://zenodo.org/record/6534531

replaced by the characters colon and dash "`:-`" for Clingo compatibility. Rules are sometimes written on several lines, as they are ended only by a period "`.`". Lines beginning with a percent sign "`%`" are comments.

7.5.1. Translating Asynchronous Automata Networks into Answer Set Programs

Before any analysis of an AAN model, we first need to express it with ASP rules. We developed a dedicated converter named `an2asp.py` and we detail its principle in the following.

First, the predicate `automaton_level(Automaton, Level)` is used to define each automaton `Automaton` along with its local state `Level`. Each local transition is then represented with two predicates: `condition` which is used several times to define each element of the condition along with the origin, and `target` which is used once to define the target of the local transition. Each local transition is labeled by an identifier that is the same in its `condition` and `target` predicates.

Example 7.10*(Representation of an AAN Model in ASP)*: *Here is the representation of the AAN model of Figure 7.1 in ASP:*

```
% Automata and local states
automaton_level("a", 0..1). automaton_level("b", 0..2).
automaton_level("c", 0..1). automaton_level("d", 0..2).
% Local transitions on a
condition(t1, "a", 0). target(t1, "a", 1). condition(t1, "c", 1).  % a_0 -{c_1}-> a_1
condition(t2, "a", 1). target(t2, "a", 0). condition(t2, "b", 2).  % a_1 -{b_2}-> a_0
```

```
7 % Local transitions on b
8 condition(t3, "b", 0). target(3, "b", 1). condition(t3, "d", 0).
9 condition(t4, "b", 0). target(4, "b", 2). condition(t4, "a", 1). condition(t4, "c", 1).
10 condition(t5, "b", 1). target(5, "b", 2). condition(t5, "d", 1).
11 condition(t6, "b", 2). target(6, "b", 0). condition(t6, "c", 0).
12 % Local transitions on c
13 condition(t7, "c", 0). target(7, "c", 1). condition(t7, "a", 1). condition(t7, "b", 0).
14 condition(t8, "c", 1). target(8, "c", 0). condition(t8, "d", 2).
15 % Local transitions on d
16 condition(t9, "d", 0). target(9, "d", 1). condition(t9, "b", 2).
17 condition(t10, "d", 0). target(10, "d", 2). condition(t10, "a", 0). condition(t10, "b", 1).
18 condition(t11, "d", 1). target(11, "d", 0). condition(t11, "a", 1).
19 condition(t12, "d", 2). target(12, "d", 0). condition(t12, "c", 0).
```

In lines 2–3 we define the model's automata with their local states. For example, the automaton `"a"` *has two levels numbered* `0` *and* `1`*. For this, we use the construct* `m..n` *in rule* `automaton_level("a", 0..1).` *of line 2 which is actually a shortcut for the two following rules:*

```
automaton_level("a", 0). automaton_level("a", 1).
```

All the local transitions of the network are defined in lines 5–19; for instance, all the predicates in line 5 declare the transition $\tau_1 = a_0 \xrightarrow{\{c_1\}} a_1$*, which is internally labeled* `t1`*. We declare as many predicates* `condition` *as necessary in order to fully define a local transition* τ *that has potentially several elements in its condition* $\mathsf{cond}(\tau)$*. For instance, transition* $b_0 \xrightarrow{\{a_1,c_1\}} b_2$ *is defined in line 9 with label* `t4` *and requires three of these predicates for* b_0*,* a_1 *and* c_1*.*

Since the names of the biological components may start with a capital letter or contain non-alphabetic characters, it is preferable to use double quotes (" ") around the automata names in the parameters of predicates to ensure that they are correctly interpreted as constants by the ASP grounder.

Finally, lines lines 21–24 extend the facts presented above and are thus always defined in other scripts to be used in more complex rules. Predicate `automaton` gathers all existing automata names in the model. The underscore symbol "_" in the parameters of a predicate is a placeholder for any value that is not used elsewhere; it could be replaced by a variable name occurring only here in the rule, but this practice reduces readability. Predicate `local_transition` comes in two forms: with arity 1 (i.e., one argument) it gathers all transition labels, and with arity 2 (i.e., two arguments) it gathers transition labels and the automaton they target. Note that two predicates with the same name but different arities are completely distinct; nevertheless, using the same name allows to remind that they have a close meaning.

```
20 % Automata names
21 automaton(Automaton) ← automaton_level(Automaton, Level).
22 % Local transition names
23 local_transition(Transition) ← target(Transition, _, _).
24 local_transition(Transition, Automaton) ← target(Transition, Automaton, _).
```

7.5.2. Stable State Enumeration

The first aspect of this work is the enumeration of a particular type of minimal trap domains: stable states (also called fixed points or steady states) which are composed of only one global state (see Definition 7.7). They can be studied separately from

attractors because their enumeration follows a different pattern which is more specific to this problem. The encoding presented here is equivalent to [Ben Abdallah et al., 2017], which itself is an extension of [Ben Abdallah et al., 2015] that tackled a more restrictive class of automata.

To summarize roughly, the enumeration of stable states requires to encode the definition of a stable state (given in Definition 7.7) as an ASP program through logic rules. The first step of this process is to browse all the possible states of the network; in other words, all possible combinations of local states are generated by choosing exactly one local level for each automaton. This is performed by lines 26–27 which create several *candidate* solutions, more precisely as many as there are sets respecting the cardinality constructs (around the curly braces). More precisely, for each value of `Automaton` so that `automaton(Automaton)` is true, we enumerate all values of `Level` so that `automaton_level(Automaton, Level)` is true, and create as many `local_state(Automaton, Level)` predicates; only one is kept in each answer set due to the lower and upper cardinalities of 1. Therefore, we theoretically obtain as many answer sets as there are possible states in the model. Of course, among these candidate solutions, we want to filter out all those that are not valid answers to our problem of stable state enumeration.

```
25 % Enumerate all possible states (one local state per automaton)
26 1 { local_state(Automaton, Level) : automaton_level(Automaton, Level) } 1 ←
27   automaton(Automaton).
```

For each enumerated state, we want to filter out those featuring at least one *playable* local transition, that is, a local transition for which all conditions are met. It is not possible to express playable transitions directly, because ASP naturally expresses existentiality (∃) rather than universality (∀). Nevertheless, universality in ASP can be expressed as a negation of existentiality. Therefore, we first express *unplayable* local

transitions, that is, transitions such that there exists a condition that is not met, with predicate `unplayable` (lines 29–32). This predicate can then be used with a negation by default (`not`) to find *playable* transitions: indeed, a transition `T` is playable if `unplayable(T)` is not part of the answer set, and thus if `not unplayable(T)` holds. Finally, using this, we can filter out all candidate solutions (i.e., states) that contain at least one playable local transition, with the constraint of line 34.

```
28 % Compute not playable transitions in the current state
29 unplayable(Transition) ←
30   local_state(Automaton, LevelI),
31   condition(Transition, Automaton, LevelJ),
32   LevelI != LevelJ.
33 % Constraint: discard states with a playable transition
34 ← not unplayable(Transition), local_transition(Transition).
```

The following special clause can be added to the Clingo script in order to show only the predicates `local_state` of arity 2 and hide all the others:

```
35 #show local_state/2.
```

Example 7.11*(Stable State Enumeration)*: *The AAN model of Figure 7.1 contains 4 automata: a and c have 2 local states while b and d have 3; therefore, the whole model has $2 * 2 * 3 * 3 = 36$ states (whether they can be reached or not from a given initial state). We can check that this model contains exactly 3 stable states: $\langle a_0, b_0, c_0, d_1 \rangle$, $\langle a_1, b_1, c_1, d_0 \rangle$ and $\langle a_1, b_1, c_0, d_0 \rangle$. All of them are represented in both Figures 7.2 and 7.3. In this model, no other state verifies this property. We recall that the stable states are identical for the synchronous and asynchronous semantics [Klarner et al., 2015].*

To enumerate stable states, one can execute the ASP program detailed above

(lines 26–35) alongside with the AAN model given in Example 7.10 (lines 1–19) and the extended facts (lines 21–24). This can also be done with the following command line on the supplementary material:

```
clingo 0 stabe-states.lp ../models/asp/example.lp
```

The output of Clingo is the following, matching the expected result:

```
Answer: 1
local_state("a",0) local_state("b",0) local_state("c",0) local_state("d",1)
Answer: 2
local_state("a",1) local_state("b",1) local_state("c",1) local_state("d",0)
Answer: 3
local_state("a",1) local_state("b",1) local_state("c",0) local_state("d",0)
```

The solving is performed in about a hundredth of a second.

7.5.3. Attractors

The previous section offered a method to enumerate all stable states of a given model. In a sense, a stable state can be considered as an attractor: it cannot be escaped and its size ($n = 1$) makes it trivially minimal. However, attractors in the general case are made of several states. In the rest of this section, we exclude one-state attractors and focus on attractors that are made of several states (following Definition 7.9). We describe how to obtain all the attractors up to a given *length* in a model for the asynchronous semantics, where the length is the number of steps of the minimal path covering the whole attractor (see Definition 7.6). Obtaining all attractors of any length can be theoretically tackled by providing a length that is high enough, but attractors being often of small size, results are usually obtained even for small lengths. For the

synchronous semantics, the method still holds but only a part of the attractors are returned.

The computational method to enumerate all attractors of length `n` in AAN models consists in three steps:

1. Enumerate all paths of length `n`,
2. Remove all paths that do not contain a cycle,
3. Remove all paths that are not trap domains.

Once all steps are passed, each trace of the remaining paths is an attractor (following Lemma 7.3).

This whole section is largely similar to Ben Abdallah et al. [2017], with several improvements:

- The returned attractors can now be of length inferior or equal to n (and not necessarily exactly n);
- Minor fixes for bugs that occurred in rare cases;
- Python scripting now allows to avoid duplicate answer sets;
- Specific optimizations in the ASP scripts have been removed as they were deprecated.

7.5.3.1. Cycle Enumeration

The approach presented here first consists in enumerating all possible paths of a given length `n` in the AAN model (Definition 7.6), and attempting to find a cycle (Definition 7.10) among the first states of these paths. In an ASP program, it is possible

to instantiate constants, the values of which are defined by the user at each execution: this is the role of the lowercase `n` in `step(0..n)` (line 37), which represents the number of considered steps. For instance, if `n` is initialized at value 2, the predicate above becomes: `step(0..2)`, which in turns means: `step(0). step(1). step(2).`; in other words, 3 successive global states are considered, that is, a path of length 2 because it contains two transitions. Then, predicate `main_cycle_length(N)` states that we make the assumption that the states `0` and `N` are identical and thus characterize a cycle; this will be checked afterwards. This predicate is instantiated in line 39 with a cardinality construct that means that as many candidate solutions are created as there are possible values for `N`. In other words, all values for `N` are tested, up to `n` (the constant). Finally, we differentiate the steps before and equal to `N`, which make up the *main cycle*, with predicate `cycle_step`, and the steps after `N`, with predicate `after_cycle_step` (lines 41–43).

```
36 % Steps in the whole path
37 step(0..n).
38 % Length of the main cycle (i.e., a cycling sub-path)
39 1 {main_cycle_length(N) : step(N), N > 0 } 1.
40 % Steps in the main cycle
41 cycle_step(0..N) ← main_cycle_length(N).
42 % Steps after the main cycle
43 after_cycle_step(N+1..n) ← main_cycle_length(N).
```

In order to enumerate all the possible paths, step `0` should take the value of all the possible (global) initial states, in a similar way to the stable state enumeration. For this, the cardinality construct in lines 45–46 allows to create as many candidate solutions as there are possible initial states by activating exactly one local state for

each automaton, and thus all combinations are tested. Here and in the following, the fact that local state `L` of automaton `A` is active in a given step `S` is denoted by a predicate: `active(level(A, L), S)`.

```
44 % Select randomly one initial state (step 0)
45 1 { active(level(Automaton, Level), 0) : automaton_level(Automaton, Level) } 1 ←
46   automaton(Automaton).
```

Then, identifying the successors of a given global state requires to identify the set of its playable local transitions. We recall that a local transition is playable in a global state when its origin and all its conditions are active in that global state (see Definition 7.2). Similarly to Section 7.5.2, we define an ASP predicate `unplayable(T,S)` in lines 48–52 stating that transition `T` is *not* playable in step `S`, because at least one of its conditions is not satisfied. Obviously, each local transition that is not flagged as unplayable is playable.

```
47 % Compute not playable transitions for each step
48 unplayable(Transition, Step) ←
49   active(level(Automaton, LevelI), Step),
50   condition(Transition, Automaton, LevelJ),
51   LevelI != LevelJ,
52   step(Step).
```

At this point, one of the two semantics, asynchronous or synchronous, can be applied in order to compute the possible paths from each of the initial states. Each semantics comes in a different piece of ASP script, so that the chosen semantics can be executed alongside the main script solving the attractor enumeration. We show in the following how to compute the evolution of the model through the asynchronous or synchronous semantics, as presented in section 7.4.3. The piece of program that

computes the attractors, given afterwards, is common to both semantics.

The possible evolutions of a model from a given state, that is, the different resulting paths after playing a set of global transitions, can be enumerated with cardinality rules given below. Thus, the rules below reproduce all possible paths in the dynamics of the model by representing each possible successor of a considered state as an answer set, and so on. This enumeration encompasses the non-deterministic behavior (in both semantics).

To enforce the strictly asynchronous dynamics, which requires that exactly one automaton changes during a global transition (see Definition 7.3), we use the constraint of lines 54–59 to choose exactly one local transition to play in each step. This local transition plays the role of a global transition by itself, and will be used later to compute the contents of the following global state. From this new state, the same constraint will be applied to compute the following state, and so on, up to `n` steps.

```
53 % Asynchronous semantics: select exactly 1 local transition to play
54 1 {
55    played(Transition, Step) :
56      local_transition(Transition),
57      not unplayable(Transition, Step)
58 } 1 ←
59  step(Step).
```

The second semantics corresponds to the synchronous dynamics in which a maximal set of playable transitions, with at most one local transition per automaton, has to be played (see Definition 7.4). For this, in lines 61–64, we first search the automata that have at least one playable local transition in the current step with predicate `has_playable`. Then, the constraint of lines 66–71 selects exactly one transition for

each automaton in this case. Finally, we forbid "empty" global transitions, even when no transition is playable (line 73) as this would create a self-transition, which we deliberately exclude here. For this, we use a cardinality construct in the body of a rule, which does not create multiple candidate answer sets but only acts as an atom that "verifies" that the cardinalities are verified. Here, if this cardinality construct is true, then the constraint applies and removes the current candidate solution.

```
60 % Automata that have at least one playable local transition
61 has_playable(Automaton, Step) ←
62   not unplayable(Transition, Step),
63   local_transition(Transition, Automaton),
64   step(Step).
65 % Synchronous semantics: select 1 transition to play for each automaton, if possible
66 1 {
67     played(Transition, Step) :
68       not unplayable(Transition, Step),
69       local_transition(Transition, Automaton)
70 } 1 ←
71   has_playable(Automaton, Step).
72 % Constraint: play at least one local transition
73 ← 0 { played(_, Step) } 0, step(Step).
```

In a nutshell, one should choose one of both pieces of program presented above, that is, either lines 54–59 for the asynchronous semantics, or lines 61–73 for the synchronous one. The point of either of these pieces of programs is to produce a collection of answer sets, where each one is a possible path of length `n` (that is, up to

step `n`) and starting from any initial state (at step `0`).

But to actually produce each step, we first need to compute the resulting global step following of each global transition proposed by the semantics. For this, the predicate `change` in lines 75–78 allows to witness the fact that in a given step, a local transition is played (because it has been chosen by the semantics) and thus the local active state of the related automaton must be updated. If a change is witnessed, then line 80 defines the contents of predicate `active` in the next step, based on the target of the local transition. If there is no such change, then the local level of this automaton stays the same, as stated by line 84.

```
74 % In Step, Automaton uses Transition to change from LevelI to LevelJ
75 change(Transition, Automaton, LevelI, LevelJ, Step) ←
76   played(Transition, Step),
77   target(Transition, Automaton, LevelJ),
78   condition(Transition, Automaton, LevelI).
79 % Change for the new active level if there is a change on Automaton
80 active(level(Automaton, LevelK), Step + 1) ←
81   change(_, Automaton, _, LevelK, Step),
82   Step < n.
83 % Keep the same active level if no change on Automaton
84 active(level(Automaton, LevelK), Step + 1) ←
85   not change(_, Automaton, _, _, Step),
86   active(level(Automaton, LevelK), Step),
87   step(Step),
88   Step < n.
```

Now that the path of length `n` (constant) is fully defined, it is time to check if it actually contains a cycle of length `N` (variable), as this is a necessary and sufficient condition to be a SCSG (see Lemma 7.1) and thus a necessary condition to be an attractor. For this, we simply need to check if the states corresponding to steps `0` and `N` are identical. As this cannot be done directly (because it would require expressing universality), we first define a predicate `different_states_on` that checks whether a given automaton has the same local state in two different steps (lines 90–97). We perform this computation for each automaton and each pair of steps, because this result will be used again later. Based on this, predicate `different_states` (line 99) generalizes this to any global pair of steps, thus witnessing globally different states. Obviously, states that are not different are identical, which is expressed by predicate `same_state` (lines 101–105). As a consequence, this means that there exists a cycle between both steps (starting from the earliest to the latest). When it is not the case between steps `0` and `N`, it means that the assumption `main_cycle_length(N)` is not true, and the current answer set must be rejected, which is done by the constraint of line 107.

```
89 % States of Step1 and Step2 are different on Automaton, with Step1 < Step2
90 different_states_on(Step1, Step2, Automaton) ←
91   active(level(Automaton, LevelI), Step1),
92   active(level(Automaton, LevelJ), Step2),
93   LevelI != LevelJ,
94   step(Step1),
95   step(Step2),
96   automaton(Automaton),
97   Step1 != Step2.
```

```
98 % States of Step1 and Step2 are different on at least one automaton
99 different_states(Step1, Step2) ← different_states_on(Step1, Step2, _).
100 % States of Step1 and Step2 are identical (thus there is a cycle)
101 same_state(Step1, Step2) ←
102   not different_states(Step1, Step2),
103   step(Step1),
104   step(Step2),
105   Step1 != Step2.
106 % Constraint: remove answer sets that are not cyclic on the main cycle (steps 0 and N)
107 ← not same_state(0, N), main_cycle_length(N).
```

We now have the certainty that the steps in the main cycle, that is, from `0` to `N`, form a strongly connected subgraph. However, we must still check that the states after this main cycle, that is, steps `N+1` to `n`, are part of the same strongly connected subgraph, otherwise the rest is not applicable. For this, we simply check that each state after the main cycle is equal to at least one state in the main cycle, with predicate `valid_state_after_main_cycle` (lines 109–112). Paths containing at least one state that does not respect this are removed by the constraint of line 114.

```
108 % Check that the states after the main cycle are already visited in the main cycle
109 valid_state_after_main_cycle(Step2) ←
110   same_state(Step1, Step2),
111   cycle_step(Step1),
112   after_cycle_step(Step2).
113 % Constraint: remove answer sets that visit new states after the main cycle
114 ← not valid_state_after_main_cycle(Step1), after_cycle_step(Step1).
```

As stated in Lemma 7.1, all remaining paths are now SCSGs. We finally need to verify that they are trap domains (Lemma 7.3) in order to discriminate attractors.

7.5.3.2. Attractor Enumeration

Due to the non-deterministic nature of the dynamics handled in this work, each state in the state-transition graph of a given AAN might have several successors. Therefore, a cyclic path is not necessarily an attractor. Theoretically, the only exception is the case of Boolean models under the synchronous semantics, which is always deterministic; in this case, the computation could be stopped at this point because a cycle is necessarily an attractor [Dubrova and Teslenko, 2009, Qu et al., 2015, Hayashida et al., 2008]. In the following, we consider the general and non-deterministic case.

At this point, it is thus necessary to express the fact of *not* being a trap domain (see Lemma 7.3) in order to filter out these cases. For instance, in the partial state-transition graph of Figure 7.2, we can spot many cycles of various lengths but not all of them are attractors. In particular, the initial global state is part of a cycle of length 2 which is not an attractor, and which trace is: $\{\langle a_1, b_2, c_0, d_1\rangle, \langle a_1, b_2, c_0, d_0\rangle\}$. In the following, we will only consider the main cycle (steps `0` to `N`), because we have constrained above that the part of the path after the main cycle is only made of duplicate states of the main cycle (see lines 109–114).

The filtering that we present in the following applies fully to the asynchronous semantics only. In the case of the synchronous semantics, however, it filters out some legitimate attractors. More precisely:

- In the Boolean case (if all automata have only two possible levels) all attractors are correctly returned;
- More generally, *simple attractors*, which are made of a simple loop and not of

a composition of several loops, are correctly returned;

- A *complex attractor*, which is made of a composition of loops, is correctly returned only if there exists a covering path so that all the other global transitions in this attractor (that are not part of this path) are composed of only one local transition (similarly to the asynchronous semantics);
- All other complex attractors are erroneously filtered out.

This filtering is performed with a constraint, which, once again, is the most suitable solution. In order to define such a constraint, we need to describe the behavior that we must not observe, namely: escaping the considered main cycle. For this, it is necessary to distinguish, in a given step, the local transitions that were actually chosen to build the main cycle (predicate `played`) and the local transitions that are also playable but have not been chosen for this candidate solution. Such local transitions are gathered with the predicate `also_playable` given in lines 116–120. Then, predicate `evolves_in_main_cycle` of lines 122–128 checks when such a transition makes the dynamics evolve from a state of the main cycle to another state still in the main cycle. If this predicate does not exist for any local transition in any step, it means that this transition makes the dynamics evolve outside of the main cycle, which is thus not a trap domain. Such a case is filtered out by the constraint of line 131.

```
115 % Enumerate transitions also playable in a state (but not chosen to build the path)
116 also_playable(Transition, Step) ←
117   not unplayable(Transition, Step),
118   not played(Transition, Step),
119   local_transition(Transition),
120   step(Step).
```

```
121 % Transition allows to go from Step1 to Step2 which is also in the path
122 evolves_in_main_cycle(Transition, Step1, Step2) ←
123   also_playable(Transition, Step1),
124   target(Transition, Automaton, LevelK),
125   active(level(Automaton, LevelK), Step2),
126   cycle_step(Step2),
127   Step2 != Step1 + 1,
128   1 = { different_states_on(Step1, Step2, _) }.
129 % Constraint: remove answer sets where a local transition is playable and
130 % allows to escape the main cycle (not an attractor)
131 ← also_playable(Transition, Step), not evolves_in_main_cycle(Transition, Step, _).
```

We stress out the limitation of the piece of code above: although it is correct for the asynchronous case, line 128 is not completely correct for the synchronous case. Indeed, this line makes the assumption that all global transitions are made of exactly one local transition, which is not the case in general for the synchronous semantics. However, as explained above, it works in several cases, some of which are predictable, and, if it can miss legitimate solutions, it never returns erroneous ones.

Finally, the following line tells Clingo to only show the instances of the `active` predicate that actually encompasses the states of the final attractors:

```
132 #show active/2.
```

To conclude, we note that at this point, an attractor will always be output in a duplicated manner. Indeed, the pieces of code above enumerate all initial states, all possible main cycle lengths, and all possible dynamical branchings. Any path in this enumeration that covers an attractor is returned by the solver, and therefore a given

attractor will be returned under different forms as it can be covered in different manners. This is due to the fact that each answer set is oblivious of the other ones, and there is no possibility to stop the computation *at the ASP scripting level* when an attractor has been found under at least one form. In the next section, we will provide two ways to filter out these spurious solutions in order to output each attractor only once.

Example 7.12: *In the dynamics of the networks presented in Figure 7.1 with the asynchronous semantics, let us consider the following cycle of length* 2*, which can be seen in Figure 7.2:* $\langle a_0, b_1, c_0, d_0\rangle \rightarrow_{\mathcal{U}^{\text{asyn}}} \langle a_0, b_1, c_0, d_2\rangle \rightarrow_{\mathcal{U}^{\text{asyn}}} \langle a_0, b_1, c_0, d_0\rangle$*. Since this cycle is an attractor, following the pieces of program given above, we can use the following command line and try to enumerate it:*

```
clingo 0 --const n=2 asynch.lp attractors.lp ../models/asp/example.lp
```

The two answer sets returned are given below, the predicates have been reordered for readability:

```
Answer: 1
active(level("a",0),0) active(level("b",1),0)
active(level("c",0),0) active(level("d",0),0)
active(level("a",0),1) active(level("b",1),1)
active(level("c",0),1) active(level("d",2),1)
active(level("a",0),2) active(level("b",1),2)
active(level("c",0),2) active(level("d",0),2)
main_cycle_length(2) played(t10,0) played(t12,1) played(t10,2)

Answer: 2
```

```
active(level("a",0),0) active(level("b",1),0)

active(level("c",0),0) active(level("d",2),0)

active(level("a",0),1) active(level("b",1),1)

active(level("c",0),1) active(level("d",0),1)

active(level("a",0),2) active(level("b",1),2)

active(level("c",0),2) active(level("d",2),2)

main_cycle_length(2) played(t12,0) played(t10,1) played(t12,2)
```

Consider the first answer set (`Answer: 1`). The three states in the cycle are labeled `0`, `1` and `2`, and the active local states they contain are described by the predicate `active` (see lines 45–46, lines 80–82 and line 84). We note that states `0` and `2` are actually identical. Two additional predicates are shown to give supplementary information: predicate `played` shows the transitions (labeled `t10` and `t12`, see lines 17 and 19) allowing to run through all the states of the cycle, while predicate `main_cycle_length` (see line 39) gives the size of the cycle analyzed (here equal to `n`, but it could be lesser).

Consider now the second answer set (`Answer: 2`): it actually describes exactly the same attractor, but covered starting from the initial state $\langle a_0, b_1, c_0, d_2 \rangle$ instead of $\langle a_0, b_1, c_0, d_0 \rangle$. This second answer set, although correct, can be considered spurious. This kind of duplicated solutions is of course more present when considering answer sets of bigger size (having more possible initial states) or with internal dynamical branchings (i.e., complex attractors). Indeed, increasing the value of `n` to 4 allows to obtain 11 answer sets that in fact represent only the two attractors of the model: four answer sets for the attractor of size 2 (two similar to above, and two using a cycle of size `N = 4`) and seven for the attractor of size 4.

7.5.3.3. Python Scripting

The ASP code presented up to this point consists in the complete ASP program of our solution. However, as showed in Example 7.12, this code produces a lot of duplicated solutions: one for each possible initial state and possible traversal. Filtering out these duplicates is very troublesome in "pure" ASP. Given the implementation choices, it is even impossible, as the different answer sets do not "communicate" one with each other. Python scripting here comes in handy. In the following, we propose a Python script that, used conjointly with the previous ASP code, offers two different approaches to filter out unwanted solutions. In both cases, the scripting simply takes the form of a Python `for` loop that awaits the production of the next answer set. Once such an answer set is raised by Clingo, it is stored in a variable, ready to be processed by Python's classical imperative scripting.

Post-Filtering. With the first approach, each answer set obtained by the Python script is processed in order to represent the contained attractor under a normalized form, using a dictionary. This attractor is then compared to the current set of attractors that have already been found. If it is new, then it is output to the terminal and added to the set of found attractors; otherwise, it is simply discarded. This is a mere post-filtering because all answer sets are still enumerated by Clingo, but only a part of them are considered "legitimate" solutions and are actually output.

Pre-Filtering. With the second approach, every answer set obtained by the Python script is processed: each state contained in the attractor is translated into an ASP constraint on step `0` and added to the ASP program being solved. For instance, if an attractor $\{\langle a_1, b_1\rangle\langle a_1, b_2\rangle\}$ is found, the following constraints are added:

```
← active(level("a", 1)), 0), active(level("b", 1)), 0).

← active(level("a", 1)), 0), active(level("b", 2)), 0).
```

Each constraint allows to never visit the related states of this attractor again, thus ensuring that no duplicate of this attractor will be returned by Clingo. These constraints are arbitrarily applied to step `0` of the traversals, but any other step would lead to the same result. This can be considered pre-filtering as the next answer sets enumerated by Clingo will never be duplicates of already found attractors, because all their states are now "forbidden". We note that this approach is possible because attractors are all disjoint (which can be easily proven knowing that an attractor is both a trap domain and a SCSG). Moreover, with this pre-filtering, no more post-filtering is needed.

To call one of these scripting enhancements, simply add to the Clingo command line the `filtering-attractors.lp` script, along with a special constant to define which filtering to apply:

```
clingo 0 --const n=<length> --const filtering={pre|post} <model.lp>
{asynch|synch}.lp attractors.lp filtering-attractors.lp
```

7.6. Case Studies

In this section, we exhibit several experiments conducted on biological networks. We first detail the results of our programs on the AAN toy model of Figure 7.1, and on another 4 components model of a real system, the bacteriophage lambda. Finally, we sum up the results of benchmarks performed on other models up to 101 components. In general, the time performances are good and the overall results confirm the applicability of ASP for the verification of formal properties or the enumeration of some dynamical patterns in biological systems, although the biggest model of 101 components highlights the limits of this approach.

All experiments were performed on a desktop PC running Ubuntu 18.04 with an

Intel Core i7-8565U processor (8 cores at 1.80GHz) and 8 GB memory. The default settings of Clingo were used, including multi-threading options.

7.6.1. Toy example

We first conducted detailed experiments on the 4 components model of Figure 7.1 for the asynchronous semantics only. As detailed in Section 7.4, this network contains 4 automata and 12 local transitions. Its asynchronous state-transition graph comprises 36 different global states and some of them are detailed in the partial state-transition graph of Figure 7.2.

The analytic study of the minimal trap domains on this small network allows to find the following attractors and stable states for the asynchronous semantics:

- stable states: $\langle a_1, b_1, c_1, d_0\rangle$,$\langle a_1, b_1, c_0, d_0\rangle$ and $\langle a_0, b_0, c_0, d_1\rangle$;
- attractor of length 2: $\{\langle a_0, b_1, c_0, d_0\rangle, \langle a_0, b_1, c_0, d_2\rangle\}$;
- attractor of length 4: $\{\langle a_1, b_2, c_1, d_1\rangle, \langle a_0, b_2, c_1, d_1\rangle, \langle a_0, b_2, c_1, d_0\rangle, \langle a_1, b_2, c_1, d_0\rangle\}$.

When given to Clingo, the ASP programs given in the previous sections output the expected solutions. The output for the stable state enumeration (following the scripts of Section 7.5.2) was given in Example 7.11.

An attractor enumeration (following the scripts of Section 7.5.3) with additional information (printing of the solutions as Python structures and computation of the number of attractors of each size) can be performed by executing the following command line:

```
clingo 0 --const n=4 --const print_solutions=1 --const print_solution_sizes=1
asynch.lp attractors.lp filtering-attractors.lp ../models/asp/example.lp
```

Pre-filtering is used by default. The computation ends in about 100 milliseconds and

the output provided by the Python filtering script is given below, slightly refactored for readability:

```
*** Solutions found: 2; skipped: 0

Automata names in order:

['a', 'b', 'c', 'd']

Unique attractors:

{frozenset({(0, 1, 0, 2), (0, 1, 0, 0)}),

 frozenset({(1, 2, 1, 1), (1, 2, 1, 0), (0, 2, 1, 0), (0, 2, 1, 1)})}

Frequencies of attractor sizes:

{2: 1, 4: 1}
```

Each information is given under a Python-compatible format for easy re-use. The informations given are the following:

- the name of each automaton in the order used after;
- the set of attractors under the form of frozen sets (immuable Python sets) where each tuple gives the local state of each automaton in order;
- a dictionary giving the sizes of the attractors and the number of attractors of this size (here: one attractor of size 2 and one of size 4).

Clingo also prints these solutions under the form of predicates (not reproduced here). Note that when the filtering script is not used, the same results are found but each attractor is duplicated several times.

The computation above requires the knowledge of the length of the biggest attractor (here, 4) in order to give an adequate value to `n`. When this value is unknown and the model is small enough, it is possible to use bigger values of `n` to search for bigger attractors. For instance, when using $\texttt{n} = 40$, the computation ends in about 6 seconds

and yields the same results. The total number of global states in the model can be considered a possible higher value. However, although it is very unlikely, attractors of bigger length might still exist. Moreover, for big models, this number of global states is too big to give good performances.

The same analysis can be performed for the synchronous semantics by replacing `asynch.lp` by `synch.lp` in the command line above, and the results are also compatible with Figure 7.3, thus no attractors are missed in this case.

7.6.2. Bacteriophage Lambda

This section focuses on applying our methods in detail to a real model of small size that has been studied in existing works, namely: the *bacteriophage lambda*, also known as *lambda phage*.

The bacteriophage lambda is a virus of particular biological interest as it can show two very different responses when infecting a host bacteria: either a usual "lytic" response in which the host cell is destroyed to help the virus reproduction, or a "lysogenic" response, where the virus DNA is simply merged into the host's without fatal outcome. Four genes, named cI, cII, cro and N, have been known to be of particular importance in this process. Among them the switch of gene cI is decisive in the choice between the lytic and lysogenic responses.

To model this behavior, a small model containing exactly these four genes of interest was proposed by Thieffry and Thomas [1995], where the complete dynamics is given as a René Thomas model with discrete parameters. In this model, one of these genes is supposed to have 4 discrete expression levels, another to have 3, and the remaining two are Boolean (2 expression levels). Therefore, this model contains 48 different states. This state graph is explored in [Thieffry and Thomas, 1995] using

the classical asynchronous semantics, and the authors show the presence of a stable state and an attractor of size 2, corresponding respectively to the lysogenic and lytic responses.

To test this model with our implementation, we fetch it from the GINsim repository[3] and use the following tools to translate it to a logic program representation:

- The existing GINsim[4] tool allows to export GINML models into the *SBML-qual*[5] formalism;
- The existing bioLQM library[6] [Paulevé, 2016b, Naldi et al., 2015] can convert *SBML-qual* models into AAN models;
- Finally, our script `an2asp.py`, provided with the supplementary material, converts AAN models into ASP programs, following the principles detailed in Section 7.5.1.

It is noteworthy that each step fully preserves the dynamics between models regarding the asynchronous semantics [Chatain et al., 2014]; thus, the final ASP program is bisimilar, under the asynchronous semantics, to the original GINML model.

At this point, we can apply our stable state and attractor enumerations to the final file. The stable state enumeration outputs a unique stable state, which is valid for both the asynchronous and synchronous semantics. This stable state is coherent with the literature and is known to characterize the lysogenic response:

$$\langle \mathrm{cI} = 2; \mathrm{cII} = 0; \mathrm{cro} = 0; \mathrm{N} = 0 \rangle \ .$$

[3] http://ginsim.org/node/47
[4] http://ginsim.org/
[5] http://colomoto.org/formats/sbml-qual.html
[6] https://github.com/colomoto/bioLQM

The attractor enumeration with the asynchronous semantics has been performed for $\mathtt{n} = 48$ (the size of the state space) as it can be considered to enumerate all attractors almost certainly. This is of course only possible because the models is small. The computation finishes in 21 seconds and is also coherent with the literature as it outputs the unique following attractor, corresponding to the lytic response:

$$\{\langle \mathrm{cI} = 0; \mathrm{cII} = 0; \mathrm{cro} = 2; \mathrm{N} = 0\rangle, \langle \mathrm{cI} = 0; \mathrm{cII} = 0; \mathrm{cro} = 3; \mathrm{N} = 0\rangle\} \ .$$

Finally, the attractor enumeration with the synchronous semantics can also be applied, although the original model was not originally created to be used with this semantics. This enumeration, once again performed for $\mathtt{n} = 48$, produces the two following attractors:

$$\{\langle \mathrm{cI} = 0; \mathrm{cII} = 0; \mathrm{cro} = 2; \mathrm{N} = 0\rangle, \langle \mathrm{cI} = 0; \mathrm{cII} = 0; \mathrm{cro} = 3; \mathrm{N} = 0\rangle\} \ ,$$

$$\{\langle \mathrm{cI} = 2; \mathrm{cII} = 0; \mathrm{cro} = 1; \mathrm{N} = 0\rangle, \langle \mathrm{cI} = 1; \mathrm{cII} = 0; \mathrm{cro} = 0; \mathrm{N} = 0\rangle\} \ .$$

7.6.3. Benchmarks on Models Coming from the Literature

The problem of finding attractors in a discrete network is NP-hard, therefore the implementation that we give in this work also faces such a complexity. However, ASP solvers (namely, Clingo in our case) are specialized in tackling such complex problems. This section is dedicated to the results of several computational experiments that we performed on biological networks. We show that our ASP implementation can notably return results in only a few seconds for attractors of small size even on models with 100 components, which is considered large.

We do not give the details of the results of these experiments but rather focus on the computation times and the number of attractors found. We used several preexisting Boolean and multi-valued networks inspired from real organisms and found in the literature:

- **Lambda phage:** as detailed in Section 7.6.2 [Thieffry and Thomas, 1995];
- **Trp-reg:** a qualitative model of regulated metabolic pathways of the tryptophan biosynthesis in E. Coli [Simão et al., 2005];
- **Fission yeast:** a cell cycle model of Schizosaccharomyces Pombe [Davidich and Bornholdt, 2008];
- **Mamm.:** a mammalian cell cycle model [Fauré et al., 2006];
- **Tcrsig:** a signaling and regulatory network of the TCR signaling pathway in the mammalian differentiation [Klamt et al., 2006];
- **T-helper:** a model of the T-helper cells differentiation and plasticity, which accounts for novel cellular subtypes [Abou-Jaoudé et al., 2014].

Two approaches were used to obtain the models studied in this section. The first one consisted in downloading them from the GINsim model repository[7] [Chaouiya et al., 2012], in GINML format, and applying the automated translations detailed in Section 7.6.2 in order to obtain a model in ASP format. The second method consisted to simply manually translate the model from the literature into AAN, and finally only use the final step of this process. The characteristics of each model once translated in AAN are given in Table 7.1. The results of our benchmarks[8] are given in Table 7.2

[7] http://ginsim.org/models_repository

[8] All programs and benchmarks are available as additional files and at: https://zenodo.org/record/6534531

Table 7.1: Brief description of the models used in our benchmarks: number of automata ($|\Sigma|$), maximal local level in the automata ($\max(\mathrm{b}(\Sigma))$), number of local transitions ($|\mathcal{T}|$) and number of states in the state-transition graph ($|\mathcal{S}|$).

Models	**Model description**			
	$\|\Sigma\|$	$\max(\mathrm{b}(\Sigma))$	$\|\mathcal{T}\|$	$\|\mathcal{S}\|$
Example	4	2	12	36
Lambda phage[1]	4	3	46	48
Trp-reg[2]	4	2	14	36
Fission yeast[3]	9	2	43	$3 \times 2^9 = 1{,}536$
Mamm.[4]	10	1	34	$2^{10} = 1{,}024$
Tcrsig[5]	40	1	85	$2^{40} \simeq 10^{12}$
T-helper[6]	101	2	316	$2^{102} \simeq 5.7 \times 10^{31}$

References of the models:

[1] [Thieffry and Thomas, 1995] – http://ginsim.org/node/47

[2] [Simão et al., 2005] – manually translated

[3] [Davidich and Bornholdt, 2008] – http://ginsim.org/node/37

[4] [Fauré et al., 2006] – manually translated

[5] [Klamt et al., 2006] – http://ginsim.org/node/78

[6] [Abou-Jaoudé et al., 2014] – manually translated

for the stable state enumeration, and Tables 7.3 and 7.4 for the attractor enumeration. For stable state enumeration (Table 7.2), the following command line has been used:

```
clingo --quiet=1 0 stable-states.lp <model file>
```

in order to search for all solutions while avoiding their output on the terminal, which considerably slows down the execution when a lot of them are found. Regarding attractor enumeration, the script `benchmarks.sh` has been used, which itself calls command lines of the form:

```
clingo 0 --const n=<length> --const filtering={no|pre|post}
--const write_nbr_solutions=<output file> --quiet=2 --time-limit=100
{synch|asynch}.lp attractors.lp filtering-attractors.lp <model
file>
```

Table 7.2: Results of our stable states enumeration implementation. The successive lines sum up the information regarding models detailed in Table 7.1. For each model, the table shows the computation time for the enumeration of all results and the total number of returned answer sets.

Models	**Stable states enumeration** for both semantics	
	Time (ms)	Number
Example	< 5	3
Lambda phage	< 5	1
Trp-reg	< 5	2
Fission yeast	< 5	1
Mamm.	< 5	1
Tcrsig	6	7
T-helper	6,774	5,875,504

Finally, it has to be noted that the scripts can also be used to search for the mere *presence* of an attractor of length at most `n`, by changing the first argument of Clingo to `1` (i.e., search for one solution) instead of `0` (i.e., search for all solutions). The computation is then much faster since the first solution found triggers the end of the execution.

7.7. Conclusion

In this chapter, we emphasized the merits of ASP, a powerful declarative programming paradigm, for the analysis of dynamical biological systems. Being able to verify properties on the dynamics of biological systems is crucial in many ways. First, it helps validating the models that are designed thanks to biological expertise or raw data. Indeed, the confrontation between the knowledge on the dynamics of a system and the actual behavior of the constructed model allows to discriminate the valid models.

Table 7.3: Results of the attractor enumeration implementation. The first column gives the model name (cf. Table 7.1), the second column (n) gives the maximum size of the sought attractors, and the column 3 to 6 (resp. 7 to end) give the results of the computation of the attractors with respect to the asynchronous (resp. synchronous) semantics. For each semantics: • The three first columns (**Computation time**) give the total computation time in milliseconds for each filtering mode: no filtering, post-filtering and pre-filtering. **T.O.** means that the timeout of 100s has been reached, thus cutting the computation and outputting the number of attractors already found. • The last column ($|\mathcal{A}|$ or **inf** $|\mathcal{A}|$) gives the total number of attractors computed, according to the pre-filtering. This value is the same for post-filtering, except when there is a timeout. A trailing "+" means that more attractors could possibly be found with more computation time.

Model	n	**Asynchronous**				**Synchronous**			
		Computation time (in ms) by filtering type			$\lvert\mathcal{A}\rvert$	**Computation time (in ms) by filtering type**			**inf** $\lvert\mathcal{A}\rvert$[1]
		None	**Post**	**Pre**	**Pre**	**None**	**Post**	**Pre**	**Pre**
Example	2	20	14	16	1	21	16	16	2
	5	25	26	22	2	24	21	19	2
	10	311	307	88	2	45	49	35	2
	15	7,587	7,608	224	2	67	59	53	2
Lambda phage	2	17	16	17	1	17	17	17	2
	5	65	65	63	1	35	35	36	2
	10	576	567	585	1	115	262	105	2
	15	773	787	788	1	206	204	171	2
Trp-reg	2	13	13	13	0	13	14	13	0
	5	21	22	21	1	18	19	18	1
	10	38	40	38	1	71	71	67	1
	15	62	63	67	1	55	57	52	1
Fisson yeast	2	16	17	17	0	18	17	18	1
	5	57	55	55	0	34	34	34	1
	10	345	340	336	0	192	105	178	1
	15	1,003	1,004	972	0	185	179	178	1
Mamm.	2	16	16	16	0	16	16	16	0
	5	36	36	37	0	31	31	31	0
	10	237	246	243	0	123	125	114	1
	15	2,945	2,903	3,024	0	221	228	187	1
Tcrsig	2	25	25	25	0	26	26	26	0
	5	96	101	93	0	98	98	99	0
	10	1,043	1,024	1,030	0	501	526	436	1
	15	13,109	12,983	13,076	0	1,276	1,223	1,136	1
T-helper	2	154	149	150	0	**T.O.**	**T.O.**	**T.O.**	70,544+[2]
	5	2,656	2,617	2,569	0	**T.O.**	**T.O.**	**T.O.**	73,428+[2]
	10	92,384	92,315	92,548	0	**T.O.**	**T.O.**	**T.O.**	41,038+[2]
	15	**T.O.**	**T.O.**	**T.O.**	0+	**T.O.**	**T.O.**	**T.O.**	21,792+[2]

[1] As the computation for the synchronous semantics might miss some solutions, this value is only an inferior bound of the actual value.

[2] The decrease in the number of solutions is an effect of the timeout only, since increasing the value of n can only lead to find more solutions.

Table 7.4: Sizes of enumerated attractors. The enumeration has been performed with pre-filtering only and a maximum size of $n = 15$. The first column gives the model name (cf. Table 7.1) and the second (resp. third) column details the sizes of attractors found for the asynchronous (resp. synchronous) semantics. **T.O.** means that the timeout of 100s has been reached, thus cutting the computation and outputting the number of attractors already found. The "+" means that more attractors could theoretically be found with more computation time.

Model	**Asynchronous**	**Synchronous**[1]
Example	1 attractor of size 2 and 1 attractor of size 4	2 attractors of size 2
Lambda phage	1 attractor of size 2	2 attractors of size 2
Trp-reg	1 attractor of size 4	1 attractor of size 4
Fission yeast	no attractor	1 attractor of size 2
Mamm.	no attractor	1 attractor of size 7
Tcrsig	no attractor	1 attractor of size 7
T-helper	no attractor (**T.O.**)	20512+ attractors of size 3 and 1280+ attractors of size 9 (**T.O.**)[2,3]

[1] As the computation for the synchronous semantics might miss some solutions, these values are only inferior bounds of the actual values.

[2] Due to the timeout, the sum of these values does not correspond to the total in the corresponding line $n = 15$ of Table 7.3. A possible explanation is that this benchmark included an additional attractor size counting step which lead to less solutions found before timeout.

[3] Surprisingly, no attractor of size 2 is found here, despite being found when searching with $n = 2$ (see Table 7.3). This is probably due to Clingo's internal optimization heuristics that were left as defaults for these benchmarks.

Second, it gives new knowledge about the system. It is thus possible to discover previously unknown dynamical properties or to obtain useful information for conducting new biological experiments. In this chapter, we decided to focus on a challenging, yet rewarding, scientific issue consisting in the study of attractors.

We presented a logical approach to efficiently compute the list of all stable states and attractors in biological regulatory networks. We formalized our approach using the AAN framework, which is bisimilar to many logical networks [Chatain et al., 2014]. All results given here can thus be applied to the widespread Thomas' modeling [Thomas, 1973] in the asynchronous semantics and to the Kauffman modeling in the synchronous semantics [Kauffman, 1969]. In addition, this framework could encompass any update rules, such as the ones represented in [Gershenson, 2004, Noual and Sené, 2017].

In biological models, the identification of attractors is critical, as it gives an insight on the long-term behavior of biological systems. By combining ASP and Python scripting (an interaction recently introduced in the ASP solver we use, Clingo), we exhibited an efficient method to enumerate stable states, cycles and attractors of large models. The computational framework is based on the AAN formalism assuming non-deterministic dynamics. The major benefit of such a method is to get an exhaustive enumeration of all potential states while still being tractable for models with a hundred of interacting components. This method covers fully the asynchronous semantics, but only partially the synchronous semantics. Yet, even when synchronous semantics is considered, it identifies correctly simple attractors and a subset of complex attractors.

This work could be extended by considering adaptations and optimizations of the approach to address larger models. On the one hand, the method can be improved, for instance by generalizing it to comprehensively tackle the synchronous semantics, along with others, or by developing optimizations to tackle even larger models. On

the other hand, close problems and similar dynamical patterns can also be considered for enumeration, such as basins of attraction, gardens of Eden or bifurcations [Fippo-Fittime et al., 2016].

Bibliography

Wassim Abou-Jaoudé, Pedro T Monteiro, Aurélien Naldi, Maximilien Grandclaudon, Vassili Soumelis, Claudine Chaouiya, and Denis Thieffry. Model checking to assess t-helper cell plasticity. *Frontiers in bioengineering and biotechnology*, 2, 2014.

Tatsuya Akutsu, Sven Kosub, Avraham A Melkman, and Takeyuki Tamura. Finding a periodic attractor of a boolean network. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 9(5):1410–1421, 2012.

Réka Albert and Hans G Othmer. The topology of the regulatory interactions predicts the expression pattern of the segment polarity genes in drosophila melanogaster. *Journal of theoretical biology*, 223(1):1–18, 2003.

Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003. ISBN 0521818028.

Emna Ben Abdallah, Maxime Folschette, Olivier Roux, and Morgan Magnin. Exhaustive analysis of dynamical properties of biological regulatory networks with answer set programming. In *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, pages 281–285. IEEE, 2015.

Emna Ben Abdallah, Maxime Folschette, Olivier Roux, and Morgan Magnin. Asp-based method for the enumeration of attractors in non-deterministic synchronous

and asynchronous multi-valued networks. *Algorithms for Molecular Biology*, 12 (1):1–23, 2017.

Nikolaos Berntenis and Martin Ebeling. Detection of attractors of large boolean networks via exhaustive enumeration of appropriate subspaces of the state space. *BMC bioinformatics*, 14(1):1, 2013.

Claudine Chaouiya, Aurelien Naldi, and Denis Thieffry. Logical modelling of gene regulatory networks with GINsim. *Bacterial Molecular Networks: Methods and Protocols*, pages 463–479, 2012.

Thomas Chatain, Stefan Haar, Loïg Jezequel, Loïc Paulevé, and Stefan Schwoon. Characterization of reachable attractors using petri net unfoldings. In *International Conference on Computational Methods in Systems Biology*, pages 129–142. Springer, 2014.

Stéphanie Chevalier, Christine Froidevaux, Loïc Paulevé, and Andrei Zinovyev. Synthesis of boolean networks from biological dynamical constraints using answer-set programming. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 34–41. IEEE, 2019.

Francis S Collins, Michael Morgan, and Aristides Patrinos. The human genome project: lessons from large-scale biology. *Science*, 300(5617):286–290, 2003.

Maria I Davidich and Stefan Bornholdt. Boolean network model predicts cell cycle sequence of fission yeast. *PloS one*, 3(2):e1672, 2008.

Elena Dubrova and Maxim Teslenko. A SAT-based algorithm for computing attractors in synchronous boolean networks. *arXiv preprint arXiv:0901.4448*, 2009.

Elena Dubrova and Maxim Teslenko. A SAT-based algorithm for finding attractors in

synchronous boolean networks. *IEEE/ACM transactions on computational biology and bioinformatics*, 8(5):1393–1399, 2011.

Adrien Fauré, Aurélien Naldi, Claudine Chaouiya, and Denis Thieffry. Dynamical analysis of a generic boolean model for the control of the mammalian cell cycle. *Bioinformatics*, 22(14):e124–e131, 2006.

Louis Fippo-Fittime, Olivier Roux, Carito Guziolowski, and Loïc Paulevé. Identification of bifurcations in biological regulatory networks using answer-set programming. In *Constraint-Based Methods for Bioinformatics Workshop*, 2016.

Maxime Folschette, Loïc Paulevé, Morgan Magnin, and Olivier Roux. Sufficient conditions for reachability in automata networks with priorities. *Theoretical Computer Science*, 608:66–83, 2015.

Abhishek Garg, Luis Mendoza, Ioannis Xenarios, and Giovanni DeMicheli. Modeling of multiple valued gene regulatory networks. In *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1398–1404. IEEE, 2007.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. 52, 2016.

Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

Carlos Gershenson. Updating schemes in random boolean networks: Do they really matter. In *Artificial Life IX Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems*, pages 238–243. MIT Press, 2004.

Aitor González, Claudine Chaouiya, and Denis Thieffry. Logical modelling of the role

of the hh pathway in the patterning of the drosophila wing disc. *Bioinformatics*, 24 (16):i234–i240, 2008.

Morihiro Hayashida, Takeyuki Tamura, Tatsuya Akutsu, Shu-Qin Zhang, and Wai-Ki Ching. Algorithms and complexity analyses for control of singleton attractors in boolean networks. *EURASIP Journal on Bioinformatics and Systems Biology*, 2008(1):1, 2008.

Sui Huang, Gabriel Eichler, Yaneer Bar-Yam, and Donald E Ingber. Cell fates as high-dimensional attractor states of a complex gene regulatory network. *Physical review letters*, 94(12):128701, 2005.

David James Irons. Improving the efficiency of attractor cycle identification in boolean networks. *Physica D: Nonlinear Phenomena*, 217(1):7–21, 2006.

Stuart A. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467, 1969.

Tarek Khaled and Belaid Benhamou. An asp-based approach for boolean networks representation and attractor detection. In *LPAR*, pages 317–333, 2020.

Steffen Klamt, Julio Saez-Rodriguez, Jonathan A Lindquist, Luca Simeoni, and Ernst D Gilles. A methodology for the structural and functional analysis of signaling and regulatory networks. *BMC bioinformatics*, 7(1):1, 2006.

Hannes Klarner, Alexander Bockmayr, and Heike Siebert. Computing maximal and minimal trap spaces of boolean networks. *Natural Computing*, 14(4):535–544, 2015.

Konstantin Klemm and Stefan Bornholdt. Stable and unstable attractors in boolean networks. *Physical Review E*, 72(5):055101, 2005.

Alexandre Lemos, Inês Lynce, and Pedro T Monteiro. Repairing boolean logical models from time-series data using answer set programming. *Algorithms for Molecular Biology*, 14(1):1–16, 2019.

Pablo Moisset de Espanés, Axel Osses, and Ivan Rapaport. Fixed-points in random Boolean networks: The impact of parallelism in the Barabási–Albert scale-free topology case. *Biosystems*, 150:167–176, 2016.

Mushthofa Mushthofa, Gustavo Torres, Yves Van de Peer, Kathleen Marchal, and Martine De Cock. ASP-G: an ASP-based method for finding attractors in genetic regulatory networks. *Bioinformatics*, page btu481, 2014.

Aurélien Naldi, Pedro T Monteiro, Christoph Müssel, Hans A Kestler, Denis Thieffry, Ioannis Xenarios, Julio Saez-Rodriguez, Tomas Helikar, Claudine Chaouiya, et al. Cooperative development of logical modelling standards and tools with colomoto. *Bioinformatics*, page btv013, 2015.

Mathilde Noual and Sylvain Sené. Synchronism versus asynchronism in monotonic boolean automata networks. *Natural Computing*, pages 1–10, 2017. ISSN 1572-9796. doi: 10.1007/s11047-016-9608-8. URL http://dx.doi.org/10.1007/s11047-016-9608-8.

Loïc Paulevé. Goal-oriented reduction of automata networks. In *International Conference on Computational Methods in Systems Biology*, volume 9859 of *Lecture Notes in Bioinformatics*, pages 252–272. Springer, 2016a.

Loïc Paulevé. Pint, a static analyzer for dynamics of automata networks. In *14th International Conference on Computational Methods in Systems Biology (CMSB 2016)*, 2016b.

Loïc Paulevé, Morgan Magnin, and Olivier Roux. Refining dynamics of gene regulatory networks in a stochastic π-calculus framework. In *Transactions on Computational Systems Biology XIII*, pages 171–191. Springer, 2011.

Loïc Paulevé, Courtney Chancellor, Maxime Folschette, Morgan Magnin, and Olivier Roux. Analyzing large network dynamics with process hitting. *Logical Modeling of Biological Systems*, pages 125 – 166, 2014.

Hongyang Qu, Qixia Yuan, Jun Pang, and Andrzej Mizera. Improving bdd-based attractor detection for synchronous boolean networks. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*. ACM, 2015.

E Simão, Elisabeth Remy, Denis Thieffry, and Claudine Chaouiya. Qualitative modelling of regulated metabolic pathways: application to the tryptophan biosynthesis in e. coli. *Bioinformatics*, 21(suppl 2):ii190–ii196, 2005.

Thomas Skodawessely and Konstantin Klemm. Finding attractors in asynchronous boolean dynamics. *Advances in Complex Systems*, 14(03):439–449, 2011.

Roland Somogyi and Larry D Greller. The dynamics of molecular networks: applications to therapeutic discovery. *Drug discovery today*, 6(24):1267–1277, 2001.

Denis Thieffry and René Thomas. Dynamical behaviour of biological regulatory networks—ii. immunity control in bacteriophage lambda. *Bulletin of Mathematical Biology*, 57(2):277–297, 1995.

René Thomas. Boolean formalization of genetic control circuits. *Journal of Theoretical Biology*, 42(3):563 – 585, 1973. ISSN 0022-5193.

René Thomas. Regulatory networks seen as asynchronous automata: a logical description. *Journal of theoretical biology*, 153(1):1–23, 1991.

Andrew Wuensche. Genomic regulation modeled as a network with basins of attraction. In *Pacific Symposium on Biocomputing*, volume 3, pages 89–102, 1998.

Shu-Qin Zhang, Morihiro Hayashida, Tatsuya Akutsu, Wai-Ki Ching, and Michael K Ng. Algorithms for finding small attractors in boolean networks. *EURASIP Journal on Bioinformatics and Systems Biology*, 2007(1):1–13, 2007.

Zheng Zhao, Chian-Wei Liu, Chun-Yao Wang, and Weikang Qian. Bdd-based synthesis of reconfigurable single-electron transistor arrays. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pages 47–54. IEEE Press, 2014.