Chapter 1
Fuzzy Answer Set Programming: from Theory to Practice

Mushthofa Mushthofa[1,3], Steven Schockaert[2], and Martine De Cock[1,4]

1.1 Introduction

Fuzzy Answer Set Programming (FASP) is a declarative programming framework aimed at solving combinatorial search/optimization problems in continuous domains [32, 6]. It extends Answer Set Programming (ASP [4, 26]), a well known declarative language that allows users to specify combinatorial search and optimization problems in an intuitive way. ASP stands out among other logic-based programming frameworks by being more purely declarative (compared to, e.g., Prolog), allowing for a more concise and intuitive encoding, while at the same time being highly expressive. The availability of efficient solvers, which are able to solve hard real-world problems, has also significantly contributed to the popularity of this modeling language. In the wake of ASP's extensive development over the last decades [1], FASP has recently been gaining more attention as well, including the development of FASP solvers that enable the use of FASP for real-world problem solving beyond toy examples. In [2], the authors developed a prototype FASP solver using the method of fuzzy set approximations. They improved the solver further by using a translation to Satisfiability Modulo Theory (SMT) [3], which increased the performance of their solver significantly on many test instances. We have also developed our own FASP solver, based on the idea of a translation to ASP, and making use of currently available ASP solvers [27, 28].

In general, the workflows used for problem solving with FASP or ASP are quite similar, and can be summarized as follows (Figure 1.1): (1) first we encode the problem as a (fuzzy) logic program, containing all the required facts, rules and/or con-

[1]Dept. of Applied Math., Comp. Sc. and Statistics, Ghent University, Belgium,
email: {Mushthofa.Mushthofa, Martine.DeCock}@UGent.be ·
[2]School of Computer Science & Informatics, Cardiff University, UK,
email: SchockaertS1@cardiff.ac.uk ·
[3]Department of Computer Science, Bogor Agricultural University, Indonesia,
email: mush@ipb.ac.id ·
[4]Institute of Technology, University of Washington Tacoma, USA, email: mdecock@uw.edu

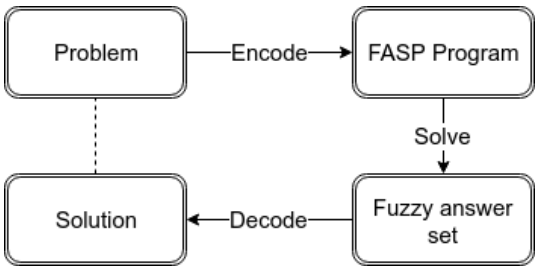

Fig. 1.1 Work flow for solving problems using (F)ASP

straints required to define the conditions of the problem; (2) we then call a (F)ASP solver, which will generate (any/all) answer sets from the specified program; and (3) finally we decode the answer sets to obtain the solutions.

In this chapter, we give an introduction to FASP, as well as a description of a state-of-the-art FASP solver and its use in practice. The remainder of this chapter is structured as follows. In the next section, we provide a high-level explanation of how ASP is typically used for solving problems, and the role that an extension to FASP can play in applications. In Section 1.3, we present the syntax and semantics of FASP, and explain how FASP programs can be used to encode problems. Section 1.4 subsequently explains how our solver finds the answer sets of a FASP program. Finally, in Section 1.5, we illustrate the whole workflow using an application of FASP to model gene regulatory networks.

1.2 Modeling problems as logic programs

Declarative programming allows one to solve problems by encoding/expressing them, usually in a rule-based logical language, allowing the programmer to specify the problem in an intuitive manner, without having to explicitly say "how" to solve the problem. It is then the task of the "solver" (which is an algorithm, or its implementation) to find the solutions in accordance with the problem specification. For example, the well-known Graph 3-coloring problem can be tackled in ASP using the following encoding:

$$col(X, red) \vee col(X, green) \vee col(X, blue) \leftarrow node(X) \tag{1.1}$$

$$\leftarrow col(X, c), col(Y, c), edge(X, Y) \tag{1.2}$$

As is common in logic programming, rules are written in the form $\alpha \leftarrow \beta$ with β the antecedent (body) of the rule and α the consequent (head). When the consequent of a rule is *false*, it is left empty, as in the second rule above, enforcing that the antecedent of the rule must be *false* for the overall rule to be satisfied. Rule (1.1) intuitively expresses that we wish to find all possible 3-colorings of the

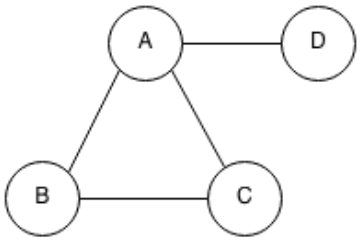

Fig. 1.2 Example graph

nodes in a graph; the predicates $node/1$ and $col/2$ are used to encode the nodes available in the graph and the colors assigned to them, respectively. The second rule, which is a "logical constraint", eliminates all the coloring schemes in which two nodes X and Y sharing an edge receive the same color c. Given the above program and a set of inputs representing all the nodes and edges of the graph, an ASP solver, such as clasp [18] or DLV [25] then searches the *answer sets* of the program, representing the possible solutions to the problem. For example, given a graph in Figure 1.2, we can encode the input to the program using the set of facts $\{node(a), \ldots, node(d), edge(a, b), edge(a, c), edge(a, d), edge(b, c)\}$. An ASP solver will then compute the answer sets, each of which corresponds to a solution. For example, one answer set will contain the atoms $col(a, red), col(b, blue)$, $col(c, green)$ and $col(d, blue)$, which indeed corresponds to a valid coloring of the given graph.

The declarative nature of the syntax of ASP and the efficiency of its solvers makes ASP a popular approach for solving combinatorial search problems. ASP has found applications in a wide range of areas, including cryptography [11], hardware design [15], data mining[20], the space shuttle decision system [20], bioinformatics [19, 29, 13] and many others [26, 14].

FASP extends the expressiveness of ASP by allowing the use of *fuzzy logic* in place of Boolean logic. The use of fuzzy logic in FASP means that predicates can have a continuum of possible truth degrees (usually taken from $[0, 1]$), rather than the discrete choices *false* and *true*. This enables the use of an ASP-like declarative specification of problems involving continuous variables. As a simple toy example, consider the problem of deciding whether to give a generous tip in a restaurant, depending on the quality of the food and service, as follows:

$$good\_food \leftarrow \textbf{not}\ bland \tag{1.3}$$

$$generous \leftarrow good\_food \otimes good\_service \tag{1.4}$$

Here, it can be more natural to express criteria such as *bland* and *good_service* as gradual properties. The truth degree of e.g. *bland* then expresses to what extent the property is satisfied. These truth degrees can them be combined using fuzzy logic operators.

1.3 Syntax and Semantics of FASP

Several different variants of FASP have been considered by different authors. Here we will focus on the variant studied in [8], whose semantics is based on Łukasiewicz logic. Similar to ASP, FASP assumes the availability of a set of propositional atom symbols, $\mathcal{B}_\mathcal{P}$. Alternatively, we can also consider a first-order syntax with predicate symbols, in which case, $\mathcal{B}_\mathcal{P}$ is the set of *ground* atoms obtained from the available predicate and constant symbols. *Grounding* is essentially the process of replacing the variables occurring in any predicate symbols with the available constant symbols. A (classical) literal is either a constant symbol $\overline{c}$ where $c \in [0,1] \cap \mathbb{Q}$, an atom a or a classical negation literal $\neg a$. An *extended* literal is a classical literal l or a *negation-as-failure* (NAF) literal **not** l. Intuitively, classical negation differs from NAF in that the former expresses our **knowledge** about something being **not** true, whereas the latter expresses our inability to **prove** that something is true.

A *head/body expression* is a formula defined recursively as follows:

- any classical literal is a head expression;
- any extended literal is a body expression;
- if α and β are head (resp. body) expressions, then $\alpha \otimes \beta$, $\alpha \oplus \beta$, $\alpha \veebar \beta$ and $\alpha \barwedge \beta$ are also head (resp. body) expressions.

A FASP program is a finite set of rules of the form:

$$\alpha \leftarrow \beta$$

where α is a head expression (called the *head* of the rule) and β is a body expression (called the *body* of the rule). As in classical ASP, we write $H(r)$ and $B(r)$ to denote the head and body of a rule r, respectively. A FASP rule of the form $a \leftarrow \overline{c}$ for a classical literal a and a constant c is called a *fact*.

A FASP rule of the form $\overline{c} \leftarrow \beta$, with $c \in [0,1] \cap \mathbb{Q}$ is called a *constraint*. A rule which does not contain any application of the operator **not** is called a *positive* rule. A rule which has at most one literal in the head is called a *normal* rule. A FASP program is called [*positive*, *normal*] iff it only contains [positive, normal] rules, respectively. Conversely, a [rule/program] which is not normal is called a disjunctive [rule/program]. A positive normal program which has no constraints is called a *simple* program.

The semantics of FASP is usually defined in relation to a chosen truth lattice $\mathcal{L} = \langle L, \leq_L \rangle$ [7]. We consider two types of truth lattices: the infinitely valued lattice $\mathcal{L}_\infty = \langle [0,1], \leq \rangle$ and the finitely valued lattices $\mathcal{L}_k = \langle \mathbb{Q}_k, \leq \rangle$, for an integer $k \geq 1$, where $\mathbb{Q}_k = \{\frac{0}{k}, \frac{1}{k}, \ldots, \frac{k}{k}\}$. Such a choice is usually determined by the nature or the goal of the application. If each proposition can only take a finite number of different truth levels, then using the $\mathcal{L}_k$ would be more appropriate. In this case, FASP is used for modeling discrete problems, and thus remains very close to classical ASP. For modeling continuous problems, or if we do not want to fix the number of truth degrees in advance, we need to use the semantics based on $\mathcal{L}_\infty$.

For any choice of lattice $\mathcal{L}$ (among the considered possibilities $\mathcal{L}_\infty$ or $\mathcal{L}_k$), an interpretation of a FASP program $\mathcal{P}$ is a function $I : \mathcal{B}_\mathcal{P} \mapsto \mathcal{L}$ which can be extended to expressions and rules as follows:

- $I(\bar{c}) = c$, for a constant $c \in \mathcal{L}$
- $I(\alpha \otimes \beta) = \max(I(\alpha) + I(\beta) - 1, 0)$
- $I(\alpha \oplus \beta) = \min(I(\alpha) + I(\beta), 1)$
- $I(\alpha \veebar \beta) = \max(I(\alpha), I(\beta))$
- $I(\alpha \barwedge \beta) = \min(I(\alpha), I(\beta))$
- $I(\textbf{not } \alpha) = 1 - I(\alpha)$
- $I(\alpha \leftarrow \beta) = \min(1 - I(\beta) + I(\alpha), 1)$

for appropriate expressions α and β. Here, the operators **not**, $\otimes, \oplus, \veebar, \barwedge$ and $\leftarrow$ denote the Łukasiewicz negation, t-norm, t-conorm, maximum, minimum and implication, respectively.

An interpretation I is consistent iff $I(l) + I(\neg l) \leq 1$ for each $l \in \mathcal{B}_\mathcal{P}$. We say that a consistent interpretation I of $\mathcal{P}$ satisfies a FASP rule r iff $I(r) = 1$. This condition is equivalent to $I(H(r)) \geq I(B(r))$. An interpretation is a model of a program $\mathcal{P}$ iff it satisfies every rule of $\mathcal{P}$. For interpretations I_1, I_2, we write $I_1 \leq I_2$ iff $I_1(l) \leq I_2(l)$ for each $l \in \mathcal{B}_\mathcal{P}$, and $I_1 < I_2$ iff $I_1 \leq I_2$ and $I_1 \neq I_2$. We call a model I of $\mathcal{P}$ a *minimal* model if there is no other model J of $\mathcal{P}$ such that $J < I$.

For a positive FASP program $\mathcal{P}$, a model I of $\mathcal{P}$ is called a *fuzzy answer set* of $\mathcal{P}$ iff it is a minimal model of $\mathcal{P}$. For non-positive programs, a common way to define the answer set semantics, in the case of classical ASP is to the so-called Gelfond-Lifschitz (GL) reduct to transform the program into a positive one, given a guess of which atoms are true. For a non-positive FASP program $\mathcal{P}$, a generalization of the GL reduct was defined in [10, 23] as follows: the reduct of a rule r w.r.t. an interpretation I is the positive rule r^I obtained by replacing each occurrence of **not** a by the constant $\overline{I(\textbf{not } a)}$. The reduct of a FASP program $\mathcal{P}$ w.r.t. an interpretation I is then defined as the positive program $\mathcal{P}^I = \{r^I \mid r \in \mathcal{P}\}$. A model I of $\mathcal{P}$ is called a fuzzy answer set of $\mathcal{P}$ iff I is a fuzzy answer set of $\mathcal{P}^I$. The set of all the fuzzy answer sets of a FASP program $\mathcal{P}$ is denoted by $\mathcal{ANS}(\mathcal{P})$. A simple FASP program has exactly one fuzzy answer set. A positive FASP program may have no, one or several fuzzy answer sets. In particular, disjunctive rules can generate many fuzzy answer sets, in general. A FASP program is called *consistent* iff it has at least one fuzzy answer set, and *inconsistent* otherwise.

Example 1 *Consider the FASP program $\mathcal{P}_1$ which has the following rules:*

$$\{a \leftarrow \textbf{not } c, b \leftarrow \textbf{not } c, c \leftarrow a \oplus b\}$$

It can be seen that under both the truth-lattice $\mathcal{L}_3$ and $\mathcal{L}_\infty$, the interpretation $I_1 = \{(a, \frac{1}{3}), (b, \frac{1}{3}), (c, \frac{2}{3})\}$ is a minimal model of $\mathcal{P}_1^{I_1}$, and hence it is an answer set of $\mathcal{P}_1$. However, the program admits no answer sets under any $\mathcal{L}_k$, where k is a positive integer not divisible by 3.

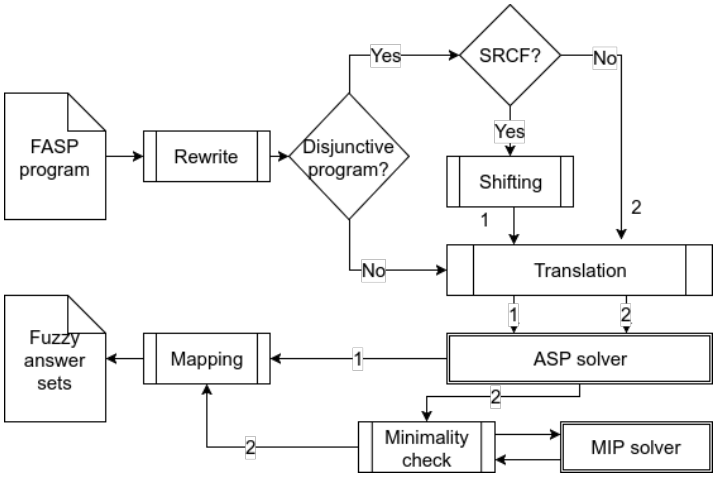

Fig. 1.3 Overall structure of the proposed solver.

Once a problem has been specified, or encoded, as a FASP program, the next main steps are (1) to automatically determine the answer sets of the FASP program, and (2) to map them back to solutions of the original problem. These steps are explained in the following sections.

1.4 Solving FASP programs

In this section, we describe the method we proposed in [27, 28] for finding the answers sets of a FASP program. Given that there are already several quite mature ASP solvers, such as clasp [18] and DLV [25], a natural strategy is to reduce the problem of evaluating a FASP program (i.e., finding its answer sets) to the problem of evaluating one or more classical ASP programs. The overall structure of our method is summarized in Figure 1.3.

The first step is to rewrite the FASP program into a more standardized form, which simplifies the subsequent steps. The strategy then depends on whether the program is disjunctive or not. Non-disjunctive programs are generally easier and more efficient to evaluate. In this case, we simply perform the translation to ASP and then utilize an ASP solver to find answer sets of the translated program, which in turn correspond to the fuzzy answer sets of the original program. In the case of disjunctive programs, two different cases are considered, as we explain below. Interested readers are invited to read [27, 28] for more details about the proposed FASP solver. The proposed approach has been implemented and is available at `https://github.com/mushthofa/ffasp`.

1.4.1 Non-disjunctive programs

We start by simplifying the syntax of a FASP program into a simpler form where there is at most one application of a connective $*$ from $\{\otimes, \oplus, \underline{\vee}, \overline{\wedge}\}$ in any rule. Intuitively, this can be done by substituting a compound expression in the rule with a fresh atom symbol, and then defining a rule for the new atom symbol accordingly. For example, the rule

$$a \leftarrow b \oplus (c \otimes \textbf{not}\ d)$$

can be substituted by the following set of rules

$$a \leftarrow b \oplus p$$
$$p \leftarrow c \otimes r$$
$$r \leftarrow \textbf{not}\ d$$

It can be shown [27] that we can always transform a FASP program $\mathcal{P}$ into the required form, and that the size of the rewritten program is $O(n \cdot m)$, where n is the number of rules in $\mathcal{P}$ and m is the maximum number of atom occurrences per rule in $\mathcal{P}$.

As mentioned previously, the main idea we proposed for evaluating FASP programs is to reduce the problem of finding the answer set of of FASP programs into that of finding the answer sets of ASP programs, which in turn can be performed efficiently using currently available solvers. For finding the answer sets of FASP programs in a truth lattice $\mathcal{L}_k$, for a given k, we have proposed a method to translate any non-disjunctive FASP program (that has been rewritten into the simple form described above) into an ASP program such that the answer sets of the ASP program correspond to the answer sets of the FASP program in $\mathcal{L}_k$. For the finding answer sets of FASP programs in $\mathcal{L}_\infty$, we employ the following strategy: perform the translation and evaluation using ASP by considering different values of k until an answer set is found, or until a certain stopping criteria is met. In this case, we need only to consider the values of k which are compatible with the constants found in the program (e.g., if a constant $\frac{1}{3}$ appears in the program, then it would be reasonable to choose only the values of k which are divisible by 3).

The translation procedure from FASP to ASP for a given k can be explained as follows (interested readers can obtain more information in [27]). First, for every atom a in the FASP program, we create up to k atom symbols a_i, $1 \leq i \leq k$ in the ASP program to denote the fact that atom a has a truth value of *at least* $\frac{i}{k}$. Then, given a FASP rule, we create a (set of) ASP rule(s) that "simulate" the rule at different truth values $1, ..., k$. For example, a simple FASP rule of the form

$$a \leftarrow \overline{c}$$

with $c \in (0, 1]$ can be translated into a single rule

$$a_j \leftarrow$$

where $j = k * c$. A FASP rule of the form

$$a \leftarrow b \oplus c$$

can be translated into the set of ASP rules

$$\{a_i \leftarrow b_j \wedge c_{k-j+i} \mid 1 \leq i \leq k, i \leq j \leq k\}$$

which can be intuitively understood as enforcing that the truth value of a should be at least as large as the sum of the truth values of b and c. The full translation scheme is given in [27]. To complete the translation, we must add the set of rules

$$\{a_i \leftarrow a_{i+1} \mid 1 \leq i \leq k-1\}$$

for every atom a in the original FASP program to ensure that the atoms a_i are consistent with the interpretation that the truth value of a is at least $\frac{i}{k}$. We can show that the resulting ASP program produces answer sets which correspond to the answer sets of the original non-disjunctive FASP program [27].

1.4.2 Disjunctive programs

The approach proposed above to evaluate FASP works well for non-disjunctive programs, i.e., programs without any applications of the operator $\oplus$ in the head of the rules. For disjunctive FASP programs, the approach can still be applied to find all answer sets with rational truth values (i.e., it is complete). However, this method may result in more than just the answer sets (i.e., it is not sound), as shown in the example below.

Example 2 *Consider the following program,* $\{a \oplus b \leftarrow \overline{1}, a \leftarrow b, b \leftarrow a\}$. *The finite-valued answer set obtained by applying the translation method to this program using* $k = 1$ *is* $A_1 = \{(a, 1), (b, 1)\}$. *In this case, it is true that* A_1 *is an answer set of the program under* $\mathcal{L}_1$. *However,* A_1 *is not an answer set of this program under* $\mathcal{L}_\infty$. *In fact, the only answer set of the program in* $\mathcal{L}_\infty$ *is* $A_2 = \{(a, 0.5), (b, 0.5)\}$, *which can be obtained using* $k = 2$.

The example shows that, given a FASP program $\mathcal{P}$, the translation method can potentially produce an answer set of $\mathcal{P}$ in $\mathcal{L}_k$, for a certain k, which is not necessarily an answer set of $\mathcal{P}$ n $\mathcal{L}_\infty$. The problem, as illustrated by the example, is that the translation method may return "answer sets" which are not minimal under $\mathcal{L}_\infty$. We can see that $A_2 < A_1$ in Example 2, and at the same time, A_2 is a model of the reduct of the program w.r.t. A_1, and thus A_1 is **not** minimal. In [28], it was shown that it is sufficient to add an extra step to check for the minimality of any answer sets obtained from the translation method.

Since this extra check of minimality may be costly, we try to avoid it as best as possible by identifying cases where we can transform a disjunctive program into a non-disjunctive one. In ASP there is an operation called "shifting", which can be used to transform certain disjunctive programs into equivalent non-disjunctive programs [5, 12]. In [28], we describe a similar transformation for FASP programs, which works for a class of programs called Self-Reinforcing Cyclic-Free (SRCF) programs. Essentially, SRCF means that we can stratify the "support" in each derived atoms, and that there is no cycle of support for these atoms. For such programs, we can then perform the "shifting" operations, as illustrated in the following example.

Example 3 *Consider program* $\mathcal{P}_2 = \{a \oplus b \leftarrow, a \leftarrow b, b \leftarrow a\}$. *It can be checked that* $\mathcal{P}_2$ *is equivalent with the program* $shift(\mathcal{P}_2)$ *as follows:*

$$a \leftarrow \textbf{\textit{not}}\ b$$
$$b \leftarrow \textbf{\textit{not}}\ a$$
$$a \leftarrow b$$
$$b \leftarrow a$$

and both have the answer set $\{(a, 0.5), (b, 0.5)\}$. *However, the program* $\mathcal{P}_2 \cup \{a \leftarrow a \oplus a\}$ *is not equivalent to* $shift(\mathcal{P}_2) \cup \{a \leftarrow a \oplus a\}$, *because the former has the answer set* $\{(a, 1), (b, 1)\}$, *while the latter does not have any answer sets.*

1.5 An Application of FASP in Biological Network modeling

In this section, we illustrate an application of FASP in the domain of computational biology, specifically in modeling the behavior of Boolean and multi-valued gene regulatory networks and computing their attractors. In biological systems, many phenotypic traits are coordinated by a set of genes interacting with each other. For simplicity, we can regard each gene as a switch that can be either turned on or turned off, representing the condition that the genes can be either expressed or not expressed. Furthermore, each gene may regulate the states of some other genes, forming a so-called Gene Regulatory Network (GRN). To understand the underlying mechanism of a certain phenomenon of a biological system, one often needs to model the GRN(s) that may contribute to it.

One of the formal tools used to model the behavior of such GRNs is called Boolean network. Informally, a Boolean network is a set of nodes, representing the genes, and a set of edges between the nodes, representing the interactions between the genes. Each node is a Boolean variable, while the interactions between the genes are usually described as a Boolean function over a set of nodes, usually called the *activation function*. The activation function of a node determines what value a node should take, given the values of all the nodes that regulate it.

Table 1.1 Regulatory relationship in the *P. aeruginosa* mucus development network

No.	x(t)	y(t)	x(t+1)	y(t+1)
1	0	0	$\frac{1}{2}$	0
2	0	1	0	0
3	$\frac{1}{2}$	0	$\frac{1}{2}$	1
4	$\frac{1}{2}$	1	0	1
5	1	0	1	1
6	1	1	1	1

The state of the network is the set of values that each node takes. Given a Boolean network with n nodes, obviously there are exactly 2^n possible states. If the Boolean network is currently on state S_i, then by applying all the activation functions in the network, we get a new state for the network, say S_j. Such a process is called a *state transition* of the network. We usually consider two modes of transition: a *synchronous* transition, where all nodes are updated simultaneously, and an *asynchronous* transition, where nodes are update sequentially. A graph that shows all the possible states of a network and all the transitions between these states is called a State Transition Graph (STG). Since the number of states are finite, after a finite number of transitions (e.g., k), the network returns to the initial state, i.e., the trajectory of the network is always of the following form: $S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_k$ where $S_k = S_1$. An attractor is a set of states $\langle S_1, S_2, \ldots, S_k \rangle$ such that there is a series of transitions of the form $S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_k$ where $S_k = S_1$ and such that $S_k = S_1$. The number of $k - 1$ is called the **size** of the attractor. An attractor of size 1 is also called a *steady state*. We refer to [24, 30, 22, 31] for a more thorough discussion regarding Boolean networks.

In some cases, representing the state of a gene with Boolean values is not enough to fully capture the important behavior of a biological system [21, 16]. A multi-valued network is a natural extension of Boolean networks where we allow the nodes to have a range of possible values, intuitively capturing the levels of expression of each gene. Consider the following example.

Example 4 *We describe the multi-valued network regulating the production of mucus in Pseudomonas aeruginosa as mentioned in [21]. The network has two nodes, namely x and y, with x having three possible values: 0, 1 or 2, and y having only two values: 0 or 1. The node x is negatively-regulated by node y and positively by itself, while y is positively-regulated by x. The input-output relationships between the two nodes, as given in [21], are shown in Table 1.1. Based on the regulatory relationships between the nodes, the state transition graph of this network is as shown in Figure 1.4. From the state transition graph, we see that the network has one steady state , namely $\langle 1, 1 \rangle$, and one cyclic attractor of size 4.*

Similar as for Boolean networks, for multi-valued networks, we are mainly interested in the steady states and the attractors of the network. In [29], ASP was used to encode the problem of finding the steady states and attractors of Boolean networks. Naturally, FASP can be used to encode the problem of finding the steady states and attractors of multi-valued networks. We first tackle the easy case of find-

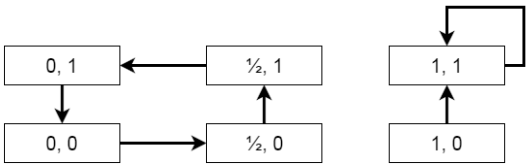

Fig. 1.4 State transition graph for the network of *P. aeruginosa* using the synchronous update

ing the steady states of a multi-valued network. As it turns out, the steady states of a Boolean/multi-valued network under synchronous and asynchronous update are exactly the same [17, 9], and hence, the following approach works for both cases. First, for every node in the network, we consider two fuzzy propositional atoms p_x and p'_x. The former represents a possible "guess" on a the activation level of node x, while the latter represents the inferred activation level after taking into account the regulatory interaction between the nodes. Intuitively, if both values are equal for all the nodes, then the guessed state is a steady state. First, we write the following rules:

$$\begin{aligned} p_x \oplus n_x \leftarrow \\ \overline{0} \leftarrow p_x \otimes n_x \end{aligned}$$

Intuitively, these rules generate the guess for all the possible states in the network, by generating all possible guesses of the truth values of p_x. The proposition n_x is just used to generate all the possible values as the complement of p_x.

We then encode the interaction between nodes by creating a rule for every node x, where the head of the rule is the propositional atom p'_x associated with the node, while the body corresponds to the direct translation of the fuzzy logic function for the update rule of x, replacing the occurrences of the negation symbol $\neg$ with FASP's default negation **not** . The following example illustrates the method.

Example 5 *Consider the network of P. aeruginosa given in Example 4. Since the network consists of two nodes, x and y, the initial guessing rules for the nodes' values can be written as*

$$\begin{aligned} x \oplus nx \leftarrow \overline{1} \\ \overline{0} \leftarrow x \otimes nx \\ x \oplus ny \leftarrow \overline{1} \\ \overline{0} \leftarrow y \otimes ny \end{aligned}$$

Since we need y to be Boolean, we add the following rule:

$$y \leftarrow y \oplus y$$

The regulatory relationships between the nodes x and y in the network (as given by Table 1.1) can be captured by the following update functions expressed in

Łukasiewicz formulas:

$$\begin{aligned} f_1(x,y) &= (\max(x, \tfrac{1}{2}) \otimes \neg y) \oplus z \\ z &= (x \otimes \tfrac{1}{2}) \oplus (x \otimes \tfrac{1}{2}) \\ f_2(x,y) &= x \oplus x \end{aligned}$$

where z is an auxiliary variable.[1] *We thus construct the following FASP rules to represent the update on each node.*

$$\begin{aligned} x' &\leftarrow (\max(x, \tfrac{1}{2}) \otimes \textbf{\textit{not}}\ y) \oplus z \\ z &\leftarrow (x \otimes \tfrac{1}{2}) \oplus (x \otimes \tfrac{1}{2}) \\ y' &\leftarrow x \oplus x \end{aligned}$$

Finally, we add the following constraints to find only steady-states:

$$\begin{aligned} \overline{0} &\leftarrow x' \otimes \textbf{\textit{not}}\ x \\ \overline{0} &\leftarrow x \otimes \textbf{\textit{not}}\ x' \\ \overline{0} &\leftarrow y' \otimes \textbf{\textit{not}}\ y \\ \overline{0} &\leftarrow y \otimes \textbf{\textit{not}}\ y' \end{aligned}$$

It can be verified that the resulting program has exactly one answer set which contains $\{(x,1),(y,1)\}$, corresponding to the only steady state $\langle 1,1\rangle$ of the network.

For finding attractors of size ≥ 2, we need to explicitly "simulate" time steps during the transitions. We can do this by turning the atoms p_x and n_x considered previously into predicate symbols with a variable parameter T, denoting the time steps. Without going into much technical details, we can then search for attractors up to certain size (say s) by simulating the transition from $T = 0$ up to $T = s$ and check for any repeated states. Consider the following example.

Example 6 *For the network in Example 4, finding the cyclic attractors of size up to 4 can be performed as follows. First, generate a guess for the initial state.*

$$\begin{aligned} x(0) \oplus nx(0) &\leftarrow \overline{1} \\ y(0) \oplus ny(0) &\leftarrow \overline{1} \\ \overline{0} &\leftarrow x(0) \otimes nx(0) \\ \overline{0} &\leftarrow y(0) \otimes ny(0) \end{aligned}$$

Since we want node y to be Boolean, we add the following rule:

$$y(T) \leftarrow y(T) \oplus y(T)$$

[1] The variable z is an auxiliary variable only intended to allow us to present a more concise expression here.

We then simulate the updating in each node using the following rules:

$$\begin{aligned} x(T+1) &\leftarrow time(T) \otimes (\max(x(T), \tfrac{1}{2}) \otimes \textbf{\textit{not}}\ y(T)) \oplus z(T)) \\ z(T) &\leftarrow (x(T) \otimes \tfrac{1}{2}) \oplus (x(T) \otimes \tfrac{1}{2}) \\ y(T+1) &\leftarrow time(T) \otimes (x(T) \oplus x(T)) \end{aligned}$$

We then add the following rules for all $i = 1, \ldots, 4$:

$$\begin{aligned} a_i &\leftarrow x(0) \otimes \textbf{\textit{not}}\ x(i) \\ a_i &\leftarrow x(i) \otimes \textbf{\textit{not}}\ x(0) \\ a_i &\leftarrow y(0) \otimes \textbf{\textit{not}}\ y(i) \\ a_i &\leftarrow y(i) \otimes \textbf{\textit{not}}\ y(0) \\ \overline{0} &\leftarrow \min(a_1, a_2, a_3, a_4) \end{aligned}$$

Intuitively, each a_i becomes true if the state at time $T = i$ is equal to the guessed initial state, which means that we have found an attractor of size i.

One can check that the resulting program has exactly five answer sets. One of these answer sets encodes the static transitions of the steady-state $\langle 1, 1\rangle$, by having the same values for $x(0), \ldots x(4)$ and $y(0), \ldots y(4)$. The other four answer sets encode the cyclic attractor $\langle 0, 0\rangle \hookrightarrow \langle \frac{1}{2}, 0\rangle \hookrightarrow \langle \frac{1}{2}, 1\rangle \hookrightarrow \langle 0, 1\rangle \hookrightarrow \langle 0, 0\rangle$, with each answer set encoding the different initial conditions.

1.6 Conclusion

In this chapter, we have provided a brief introduction to recent developments in Fuzzy Answer Set Programming (FASP), which is an extension of Answer Set Programming (ASP), a well-known declarative programming paradigm for encoding and solving combinatorial search and optimization problems. FASP extends ASP by allowing fuzzy/many-valued predicates in its programs, making it more suitable to encode problems in continuous domains. Despite the promising theoretical aspects of FASP, it is still lacking behind in terms of applicability, in comparison to ASP. This is mainly because of the – until recently – limited availability of efficient methods to evaluate FASP programs, whereas efficient solvers for ASP have been around for quite some time.

Our recent work contributes to reducing the gap by proposing new methods to efficiently evaluate FASP programs. We first described how non-disjunctive FASP programs can be efficiently translated into ASP programs whose answer sets correspond to answer sets of the original FASP programs. This opens up the possibility of using current ASP solvers to evaluate FASP programs. We then showed that disjunctive FASP programs can subsequently be evaluated by adding an extra step of checking the minimality of candidate answer sets returned by the translation method. We also showed how this minimality check can be performed by encod-

ing the problem into a MIP problem that can be solved by off-the-shelf MIP solvers. Finally, we described an application of FASP in the biological domain, namely modeling and computing the trajectory of multi-valued networks to study the behavior of gene regulatory networks. The availability of our solver paves the way for the use of FASP to tackle other real life combinatorial search problems in continuous domains, thereby taking the work on FASP from a study of its theoretical foundations into the development of practical applications.

References

1. Special issue on answer set programming. AI Magazine **37**(3) (2016)
2. Alviano, M., Peñaloza, R.: Fuzzy answer sets approximations. Theory and Practice of Logic Programming **13**(4-5), 753–767 (2013)
3. Alviano, M., Peñaloza, R.: Fuzzy answer set computation via satisfiability modulo theories. Theory and Practice of Logic Programming **15**, 588–603 (2015)
4. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
5. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Annals of Mathematics and Artificial intelligence **12**(1-2), 53–87 (1994)
6. Blondeel, M., Schockaert, S., Vermeir, D., De Cock, M.: Fuzzy answer set programming: An introduction. In: Soft Computing: State of the Art Theory and Novel Applications, pp. 209–222. Springer (2013)
7. Blondeel, M., Schockaert, S., Vermeir, D., De Cock, M.: Fuzzy answer set programming: An introduction. In: R.R. Yager, A.M. Abbasov, M.Z. Reformat, S.N. Shahbazova (eds.) Soft Computing: State of the Art Theory and Novel Applications, *Studies in Fuzziness and Soft Computing*, vol. 291, pp. 209–222. Springer Berlin Heidelberg (2013)
8. Blondeel, M., Schockaert, S., Vermeir, D., De Cock, M.: Complexity of fuzzy answer set programming under Łukasiewicz semantics. International Journal of Approximate Reasoning **55**(9), 1971–2003 (2014)
9. Bockmayr, A., Siebert, H.: Programming Logics: Essays in Memory of Harald Ganzinger, chap. Bio-Logics: Logical Analysis of Bioregulatory Networks, pp. 19–34. Springer Berlin Heidelberg (2013)
10. Damásio, C.V., Pereira, L.M.: Antitonic logic programs. In: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, pp. 379–392 (2001)
11. Delgrande, J.P., Grote, T., Hunter, A.: A general approach to the verification of cryptographic protocols using answer set programming. In: Proceedings of the 10th International Conference in Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), pp. 355–367 (2009)
12. Dix, J., Gottlob, G., Marek, W.: Reducing disjunctive to non-disjunctive semantics by shift-operations. Fundamenta Informaticae **28**(1), 87–100 (1996)
13. Dworschak, S., Grell, S., Nikiforova, V.J., Schaub, T., Selbig, J.: Modeling biological networks by action languages via answer set programming. Constraints **13**(1-2), 21–65 (2008)
14. Erdem, E.: Theory and applications of answer set programming. Ph.D. thesis, The University of Texas at Austin (2002)
15. Erdem, E., Lifschitz, V., Wong, M.: Wire routing and satisfiability planning. Computational Logic—CL 2000 pp. 822–836 (2000)
16. Espinosa-Soto, C., Padilla-Longoria, P., Alvarez-Buylla, E.R.: A gene regulatory network model for cell-fate determination during arabidopsis thaliana flower development that is robust and recovers experimental gene expression profiles. The Plant Cell Online **16**(11), 2923–2939 (2004)
17. Garg, A., Di Cara, A., Xenarios, I., Mendoza, L., De Micheli, G.: Synchronous versus asynchronous modeling of gene regulatory networks. Bioinformatics **24**(17), 1917–1925 (2008)

18. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Communications **24**(2), 107–124 (2011)
19. Gebser, M., Konig, A., Schaub, T., Thiele, S., Veber, P.: The BioASP library: ASP solutions for systems biology. In: Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2010, vol. 1, pp. 383–389. IEEE (2010)
20. Grasso, G., Leone, N., Manna, M., Ricca, F.: Asp at work: Spin-off and applications of the dlv system. Logic programming, knowledge representation, and nonmonotonic reasoning **6565**, 432–451 (2011)
21. Guespin-Michel, J., Kaufman, M.: Positive feedback circuits and adaptive regulations in bacteria. Acta Biotheoretica **49**(4), 207–218 (2001)
22. Harvey, I., Bossomaier, T.: Time out of joint: Attractors in asynchronous random boolean networks. In: Proceedings of the Fourth European Conference on Artificial Life, pp. 67–75 (1997)
23. Janssen, J., Schockaert, S., Vermeir, D., De Cock, M.: General fuzzy answer set programs. In: Fuzzy Logic and Applications, pp. 352–359. Springer (2009)
24. Kauffman, S.A.: The origins of order: Self-organization and selection in evolution. Oxford university press (1993)
25. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. on Computational Logic **7**(3), 499–562 (2006)
26. Lifschitz, V.: What is answer set programming?. In: Proceedings of the 23rd AAAI Conference in Artificial Intelligence, vol. 8, pp. 1594–1597 (2008)
27. Mushthofa, M., Schockaert, S., De Cock, M.: A finite-valued solver for disjunctive fuzzy answer set programs. In: Proceedings of European Conference in Artificial Intelligence 2014, pp. 645–650 (2014)
28. Mushthofa, M., Schockaert, S., De Cock, M.: Solving disjunctive fuzzy answer set programs. In: Proceedings of the 13th International Conference on Logic Programming and Non-monotonic Reasoning, pp. 453–466 (2015)
29. Mushthofa, M., Torres, G., Van de Peer, Y., Marchal, K., De Cock, M.: ASP-G: an ASP-based method for finding attractors in genetic regulatory networks. Bioinformatics **30**(21), 3086 (2014)
30. Thomas, R.: Boolean formalization of genetic control circuits. Journal of Theoretical Biology **42**(3), 563–585 (1973)
31. Thomas, R.: Regulatory networks seen as asynchronous automata: a logical description. Journal of Theoretical Biology **153**(1), 1–23 (1991)
32. Van Nieuwenborgh, D., De Cock, M., Vermeir, D.: Fuzzy answer set programming. In: Proceedings of the 10th European Conference on Logics in Artificial Intelligence, pp. 359–372. Springer (2006)