ANSWER SET PROGRAMMING FOR CONTINUOUS DOMAINS: A FUZZY LOGIC APPROACH

J. JANSSEN
S. SCHOCKAERT
D. VERMEIR
M. DE COCK

ATLANTIS COMPUTATIONAL
INTELLIGENCE SYSTEMS
SERIES EDITORS | JIE LU, JAVIER MONTERO

ATLANTIS PRESS

ATLANTIS COMPUTATIONAL INTELLIGENCE SYSTEMS

VOLUME 5

SERIES EDITORS: **JIE LU, JAVIER MONTERO**

Atlantis Computational Intelligence Systems

(ISSN: 1875-7650)

Aims and scope of the series

The series 'Atlantis Computational Intelligence Systems' aims at covering state-of-the-art research and development in all fields where computational intelligence is investigated and applied. The series seeks to publish monographs and edited volumes on foundations and new developments in the field of computational intelligence, including fundamental and applied research as well as work describing new, emerging technologies originating from computational intelligence research. Applied CI research may range from CI applications in the industry to research projects in the life sciences, including research in biology, physics, chemistry and the neurosciences.

All books in this series are co-published with Springer.

For more information on this series and our other book series, please visit our website at:

www.atlantis-press.com/publications/books

AMSTERDAM – PARIS – BEIJING

Answer Set Programming For Continuous Domains: A Fuzzy Logic Approach

Jeroen Janssen

Vrije Universiteit Brussel, Department of Computer Science, Pleinlaan 2, 1050 Brussels, Belgium

Steven Schockaert

School of Computer Science & Informatics, Queen's Buildings, 5 The Parade, Roath, Cardiff CF24 3AA, United Kingdom

Dirk Vermeir

Vrije Universiteit Brussel, Department of Computer Science, Pleinlaan 2, 1050 Brussels, Belgium

Martine De Cock

Ghent University, Dept. of Applied Mathematics and Computer Science (WE02), Krijgslaan 281 (S9), 9000 Gent, Belgium

AMSTERDAM – PARIS – BEIJING

Atlantis Press

8, square des Bouleaux
75019 Paris, France

For information on all Atlantis Press publications, visit our website at: *www.atlantis-press.com*

Atlantis Computational Intelligence Systems

Volume 1: Linguistic Values Based Intelligent Information Processing: Theory, Methods, and Applications - Da Ruan
Volume 2: Computational Intelligence in Complex Decision Systems - Da Ruan (Editor)
Volume 3: Intelligence for Nonlinear Dynamics and Synchronisation - K. Kyamakya, A. Bouchachia, J.C. Chedjou
Volume 4: Trust Networks for Recommender Systems - P. Victor, C. Cornelis, M. De Cock

ISBNs
Print: 978-94-91216-58-9
E-Book: 978-94-91216-59-6
ISSN: 1875-7650

Preface

Answer set programming (ASP) is a form of logic programming that originated at the end of the 1980s and the beginning of the 1990s. It is especially tailored towards solving hard search problems, which it allows to encode concisely. In the past two decades it has known great success and has – among others – been applied to planning problems, musical composition, biological modeling and decision support systems for the space shuttle. Unfortunately, ASP is not very well equipped for modeling problems in continuous domains. In this book we attempt to augment ASP with the capability of expressing continuous problems by creating an answer set programming framework based on fuzzy logic. The resulting language is called fuzzy answer set programming (FASP). After two introductory chapters, also introducing the necessary technical background, we study FASP and its extensions in Chapters 3 and 4. Then we focus on the question of whether the many extensions of FASP can be compiled to a core language in Chapter 5 and succeedingly study an implementation method for a subset of FASP in Chapter 6. As such, we focus both on theoretical aspects of the language as on more practical aspects such as implementation.

This book originated from the doctoral thesis of the first author, which was successfully defended in June 2011. Encouraged by the enthusiastic reports of the committee members, we have decided to publish this book, and make the obtained results available to a larger audience. We are grateful to the external members of the doctoral jury, Umberto Straccia and Wolfgang Faber, for their useful suggestions and remarks on the first version of this thesis. Our special thanks go to Da Ruan, the former editor of the Atlantis Computational Intelligence Systems book series, who initiated the publication process of this book shortly before he passed away very unexpectedly in the Summer of 2011. For us, this book will always be associated with dear memories of Da's friendship and his enthusiasm and help to publish our work. We would also like to thank the new series editors Jie Lu and Javier Montero for their valued contributions in continuing Da Ruan's work, and for guiding

us through the final publication stages of this book. Finally, we would like to thank the Research Foundation - Flanders (FWO) for the financial support.

Contents

Preface **v**

1. Introduction **1**

1.1 Answer Set Programming 4
1.2 Fuzzy Logic 7
1.3 Overview 10

2. Preliminaries **11**

2.1 Order Theory 11
2.2 Answer Set Programming 13
2.2.1 Definitions 13
2.2.2 Classical negation vs negation-as-failure 21
2.2.3 Links to SAT 22
2.3 Fuzzy Logic 25
2.3.1 Fuzzy Sets 26
2.3.2 Logical Operators on Bounded Lattices 26
2.3.3 Formal Fuzzy Logics 32

3. Fuzzy Answer Set Programming **35**

3.1 Introduction 35
3.2 Definitions 36
3.2.1 Language 36
3.2.2 Semantics 38
3.3 Example: Fuzzy Graph Coloring 46

4. Aggregated Fuzzy Answer Set Programming 49

4.1 Introduction . . . 49
4.2 Aggregated Fuzzy Answer Set Programming . . . 50
4.2.1 Models . . . 51
4.2.2 Answer Sets . . . 58
4.3 Illustrative Example . . . 74
4.4 Relationship to Existing Approaches . . . 79
4.4.1 Fuzzy and Many-Valued Logic Programming Without Partial Rule Satisfaction . . . 81
4.4.2 Weighted Rule Satisfaction Approaches . . . 82
4.4.3 Van Nieuwenborgh *et al.* . . . 86
4.4.4 Valued Constraint Satisfaction Problems . . . 90
4.4.5 Answer Set Optimization . . . 90
4.5 Summary . . . 91

5. Core Fuzzy Answer Set Programming 97

5.1 Introduction . . . 97
5.2 The FASP Core Language . . . 99
5.3 Constraints . . . 100
5.3.1 Implementing Constraints . . . 100
5.3.2 Locking the Truth Value . . . 105
5.4 Monotonically Decreasing Functions . . . 107
5.5 Aggregators . . . 110
5.6 S-implicators . . . 117
5.7 Strong Negation . . . 121
5.8 Summary . . . 123
5.A Proofs . . . 125
5.B Diagram . . . 131

6. Reducing FASP to Fuzzy SAT 133

6.1 Introduction . . . 133
6.2 Completion of FASP Programs . . . 134
6.3 Loop Formulas . . . 139
6.4 Example: the ATM location selection problem . . . 146

6.5 Discussion 153
6.6 Summary 155

7. Conclusions **157**

Bibliography **161**

Index **171**

Chapter 1

Introduction

Language is one of the most important tools that exist. It allows humans to communicate efficiently and to transfer knowledge between generations. According to Benjamin Whorf, language even shapes views and influences thoughts[1]. Unfortunately, while human language is useful for communication between humans, it is not as efficient for communicating with our modern day devices. Therefore, ever since the rise of computers, the need has grown for languages that enable us to tell these machines what we expect them to do. Such languages are called *programming languages*. Their foundations can be dated back to the 1800s, where Joseph Marie Jacquard used punched cards to encode cloth patterns for his textile machine, called the "Jacquard loom"[2]. Charles Babbage improved on this idea when designing his "analytical engine" by allowing the machine to be reprogrammed using punched cards[3]. Hence, instead of merely using the punched cards as data, the analytical engine could perform arbitrary computations that were encoded in the punched cards. As such, we can consider this the first real programmable machine.

The 1940s witnessed the birth of the first machines that resemble our modern day electrical computers. Initially these machines were programmed using patched cables that encoded specific machine-instructions. Input and output was done using punched cards. Since the (re)programming of these computers was a laborious task requiring many people, the idea arose to unify programs with data and store them in memory. This led to the creation of stored-program computers, such as EDVAC (Electronic Discrete Variable Automatic Computer, successor of ENIAC[4]) and SSEM (Small-Scale Experimental Machine[5]). Contrary to the earlier designs, these systems could read programs from punched cards and store

[1] Source: `http://en.wikipedia.org/wiki/Benjamin_Lee_Whorf`. Retrieved on March 29, 2011
[2] Source: `http://en.wikipedia.org/wiki/Jacquard_loom`. Retrieved on Feb 25, 2011.
[3] Source: `http://en.wikipedia.org/wiki/Analytical_Engine`. Retrieved on Feb 25, 2011.
[4] Source: `http://en.wikipedia.org/wiki/Edvac`. Retrieved on Feb 25, 2011.
[5] Source: `http://en.wikipedia.org/wiki/Ssem`. Retrieved on Feb 25, 2011.

them in memory, thereby making (re)programming them as easy as inserting a new stack of punched cards.

While the creation of stored-program computers eliminated the physical burden of programming, the mental activity required was still high due to the use of machine-specific codes. These low-level languages allowed the programmer to greatly optimize their programs for specific machines, but also made it hard to express complex problems due to their poor readability and the fact that they are far removed from natural language. To solve these problems, so called "higher-level" programming languages were developed. One of the first such languages was "Plankalkül" ("planning calculus"). It was described by Konrad Zuse in 1943 [Zuse (1943, 1948–1949); Bauer and Wössner (1972)], but was only implemented in 1998[6] and independently in 2000 [Rojas *et al.* (2000)]. In the 1950s the first high-level programming languages with working implementations were created. The most important among them are Fortran (Formula Translator), COBOL (Common Business Oriented Language) and LISP (List Processor). Fortran was mostly oriented towards scientific computing, COBOL towards business and finance administration systems and LISP towards artificial intelligence systems. Though they focused on different domains, each of them could be used to write general purpose programs. In 1960 computer scientists from Europe and the United States developed a new language, called ALGOL 60 (algorithmic language). Though the language, and especially its formulation, contained many innovations, it did not gain widespread use. Its ideas influenced many of the languages created later, however.

In the 1960s through the 1970s many of the major programming language paradigms that are still in use today were developed. For example, Simula (end of the 1960s) was the first language supporting *object-oriented programming*, Smalltalk (mid 1970s) the first fully object-oriented programming language, Prolog (1972) the first *logic programming* language and ML (1973) the first *statically typed functional programming* language[7]. Most of our modern languages have clear influences from these languages and can thus be categorized in one of the associated paradigms. Other important programming languages created in this period were Logo (1968, a LISP offspring developed for teaching), PASCAL (1970, an ALGOL offspring) and C (1972, a systems programming language).

The 1980s mostly saw the creation of languages that recombined and improved upon the ideas from the paradigms and languages invented in the 1960s and 1970s. For example, C++ (1980) combined C with object-oriented programming, Objective-C (1983) combined

[6] Source: `http://en.wikipedia.org/wiki/Plankalkul`. Retrieved on Feb 25, 2011.

[7] Note that LISP was the first *dynamically* typed functional programming language.

C with Smalltalk-style messaging and object-oriented programming and Erlang (1986) combined functional programming with provisions for programming distributed systems. Next to these languages, a subparadigm of functional programming, called *purely* functional programming, was also created. Notable examples of the latter are Miranda (1985) and Haskell (1990).

In the 1990s general interest arose in programming languages that improve programmer productivity, so called *rapid development languages*[8]. Most of these languages incorporated object-oriented features or were fully object-oriented and had garbage-collection utilities to relieve the programmer of manual memory management. Examples are Python (1991), Visual Basic (1991), Ruby (1993), Java (1995) and Delphi (1995). The rise of the internet also spurred the development of scripting languages such as JavaScript (1995) and PHP (1995), which enabled the fast creation of interactive and dynamic websites. Due to the occurrence of computers with multiple cores, in the 2000s languages tailored for these machines were created, such as Clojure (2007) and Go (2009).

All programming languages mentioned above are *general-purpose* programming languages. This means they can be used to write software for many different application domains. While such languages have the advantage of only needing to learn one language for writing a variety of software, most of these languages do not support special constructs for specific application domains. This makes the translation of the requirements of a new software package into code much harder. *Domain-specific languages* are languages that are tailored towards one specific problem domain. Notable examples are regular expressions for handling text, SQL for describing database interactions and Yacc for creating compiler front-ends. Since the 1990s interest in domain-specific languages has increased. In fact, a new programming methodology, called language-oriented programming, has arisen that proposes to create a new language describing the domain first, and then use this language to write the final program [Ward (1994)].

Answer set programming is a declarative domain-specific language tailored towards solving combinatorial optimization problems. It has roots in logic programming and non-monotonic reasoning. In this book, we study a new domain-specific language, called *fuzzy answer set programming*, that is aimed towards solving continuous optimization problems. It combines answer set programming with *fuzzy logic* – a mathematical logic which can describe continuous concepts in an intuitive manner. In the next two sections we describe the history and general idea of these two cornerstones in more detail.

[8] Source: `http://en.wikipedia.org/History_of_programming_languages`. Retrieved on Mar 1, 2011

1.1 Answer Set Programming

To create systems that are capable of human-like reasoning, we need languages that are tailored towards representing knowledge and a method for reasoning over this knowledge. An idea that immediately comes to mind is to use logic to describe our knowledge and use model-finding algorithms (e.g. SAT solving) for the reasoning part. One of the limitations of classical logic when mimicking human reasoning is that it works *monotonically*: when new knowledge is added, the set of conclusions that can be inferred using classical logic grows. In contrast humans constantly revise their knowledge when new information becomes available. For example if we know that Pingu is a bird, we assume that he can fly. If we afterwards are told that he is a penguin, however, we need to revise our belief, as we know that penguins can't fly.

During the last decades, researchers have studied *non-monotonic* logics as a way to overcome this limitation of classical logic. Several such logics have been proposed, such as circumscription [McCarthy (1980)], default logic [Reiter (1980); Lukaszewicz (1984); Brewka (1991); Przymusińska and Przymusiński (1994)], auto-epistemic logic [Moore (1985)], non-monotonic modal logics in general [McDermott (1982)] and logic programming with negation-as-failure [Clark (1977); Van Gelder *et al.* (1991); Gelfond and Lifschitz (1988)]. In this book, we will focus on the latter.

Non-monotonicity in logic programming is obtained using a special construct called *negation-as-failure*, which is denoted as "not a" and intuitively means that the negation of a is true when we fail to derive a. Defining the semantics of this construct proved to be a challenge, however. The most important proposed definitions are the Clark completion [Clark (1977)], the stable model semantics [Gelfond and Lifschitz (1988)] and the well-founded semantics [Van Gelder *et al.* (1991)]. The stable model semantics refine the conclusions of the Clark completion in the presence of positive mutual dependencies between predicates [Fages (1994)]. The well-founded semantics on the other hand are more cautious in their conclusions than both the stable models and the Clark completion when there are mutual dependencies between predicates with the negation-as-failure construct. It has been shown that the well-founded semantics are an approximation of the stable model semantics [Baral and Subrahmanian (1993)]. A lot of research has also been devoted to the relationships between stable model semantics and other non-monotonic logic formalisms. For a good overview of these links we refer the reader to [Baral (2003)]. Attention has also been given to studying extensions of these semantics. In [Lifschitz and Woo (1991)] the stable model semantics is extended to programs with disjunctions, which has been shown

to make the language capable of modeling a larger class of problems [Eiter *et al.* (1994)]. Another important extension is the addition of a second form of negation, called *classical negation* [Gelfond and Lifschitz (1991)]. Whereas negation-as-failure denotes that the negation follows from a failure to derive a proof term, classical negation denotes that the negation of the proof term can explicitly be derived.

At the end of the 1990s researchers began to notice that the stable model semantics gives rise to a certain logic programming paradigm that is different from the proof-derivation based approach of languages such as Prolog [Marek and Truszczyński (1999); Niemelä (1999)]. Vladimir Lifschitz named this new paradigm "*answer set programming*" (ASP) in [Lifschitz (2002, 1999)]. The basic idea of answer set programming is that a programmer translates a certain problem into an answer set program (a logic program under the stable model semantics) such that the *answer sets* (stable models) of the program correspond to the problem solutions. This program is then given as input to an *answer set solver* which computes the answer sets of the program. This solver has three possible outputs:

(1) No answer set exists. In this case, the modeled problem does not have a solution.
(2) One answer set exists. In this case, the answer set corresponds to the solution of the modeled problem.
(3) Multiple answer sets exist. In this case, the modeled problem has multiple solutions. The user can ask the answer set solver to compute all answer sets, or only a single one if this suffices.

For example, consider the problem of finding a large clique, i.e. a subset V of an undirected graph such that: (i) there is an edge between every pair of vertices in V; (ii) the cardinality of V is greater than or equal to a given l. If we take $l = 3$, for example, we can solve this using the following answer set program P_{clique} (from [Lifschitz (2002)])[9]:

$$in(X) \leftarrow \text{not}\, out(X) \tag{1.1}$$

$$out(X) \leftarrow \text{not}\, in(X) \tag{1.2}$$

$$sizeOk \leftarrow in(X), in(Y), in(Z), X \neq Y, X \neq Z, Y \neq Z \tag{1.3}$$

$$joined(X,Y) \leftarrow edge(X,Y) \tag{1.4}$$

$$joined(X,Y) \leftarrow edge(Y,X) \tag{1.5}$$

[9]Note that existing answer set solvers support an extension that allows to write the combination of rules (1.1)–(1.3) and (1.7) as the single rule "$3\,\{in(X)\}$". Since we do not consider these extensions in this book, we opted to remove this syntactic sugar.