The Knowledge Engineering Review, Vol. 33, e19, 1–30.
doi:10.1017/S0269888918000164

Answer set programming and agents

ABEER DYOUB, STEFANIA COSTANTINI and GIOVANNI DE GASPERIS

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, Universit'a degli Studi dell'Aquila, Via Vetoio 1, Loc. Coppito, I-67100 L'Aquila, Italy;
e-mail: abeer.dyoub@graduate.univaq.it, stefania.costantini, giovanni.degasperis@univaq.it

Abstract

In this paper, we discuss the potential role of answer set programming (ASP) in the context of approaches to the development of agents and multi-agent systems especially in the realm of Computational Logic. After shortly recalling the main (computational-logic-based) agent-oriented frameworks, we introduce ASP; then, we discuss the usefulness of a potential integration of the two paradigms in a modular heterogeneous framework, and the feasibility of such integration. This also in the more general view of improving and empowering flexibility of agent-oriented frameworks. Relevant literature will be mentioned and discussed. Possible future directions and potential developments will be outlined.

1 Introduction

Answer set programming (ASP) (Baral, 2003; Gelfond, 2008; Truszczyńki, 2007; Borchert *et al*., 2004; Leone, 2007) is a well-established logic programming paradigm under the answer set semantics, whose origins go quite a long time back. In fact, ASP is a successful outcome of a long research line in the areas of knowledge representation (KR), logic programming, and constraint satisfaction. ASP finds applications in many areas such as, for example, information integration, constraint satisfaction, routing, planning, diagnosis, security analysis, configuration, computer-aided verification, biology/biomedicine, knowledge management, etc. The ASP approach to problem-solving consists basically in the following: (i) encoding of given problem via an ASP program; (ii) computing the 'answer sets' of such program via an inference engine, or 'ASP solver'; (iii) extracting the problem solution by examining such answer sets; answer set contents can be in general reformulated in order to presentat the solution in the terms of the given problem.

Many real-world and industrial applications have demonstrated the effectiveness of ASP in practice, as complex specifications can be implemented and deployed via ASP-based technologies at a lower development cost with respect to imperative languages. This in conjunction to other several advantages observed from the software engineering perspective while developing real-world applications: flexibility, readability, extensibility, ease of maintenance, etc. Indeed, the possibility of developing the implementation of complex problemsby simply editing a text file containing ASP rules, and testing it on site together with the customer by running an ASP solver, is a widely recognized advantage of ASP-based development. This aspect of ASP-based software development constituted a success factor for applications where the high complexity of the requirements was a main obstacle. Moreover, ASP inference engines implement advanced optimization techniques (e.g. Magic Set Optimization Alviano *et al*. (2012)) leading to a substantial speed up of the computation that allows to handle industrial applications where a timely response has to be provided for queries involving huge data sets. In addition, ASP has been endowed with effective programming tools to support the activities of researchers and implementers, and to simplify the integration of ASP into existing well-assessed development processes and platforms which are tailored for object-oriented programming languages, and into distributed and service-oriented frameworks.

Agent-based systems constitute one of the most important areas of research and development in information technology in the last few decades. The concept of agency has been exploited in a diverse range of sub-disciplines of information technology, including computer networks, software engineering, artificial intelligence, human-computer interaction, distributed and concurrent systems, mobile systems, telematics, computer-supported cooperative work, control systems, decision support systems (business intelligence), information retrieval and management and electronic commerce. In fact, designing and implementing intelligent agents has been the main focus of the field of artificial intelligence since its birth. Naturally, a prerequisite to constructing systems capable of reasoning and operating autonomously in the environment is to provide dedicated infrastructure and supporting tools: this includes the development of agent-oriented programming (AOP) languages and related software engineering methodologies. Several agent theories, architectures and languages have emerged, concerned with the definition of an agent and its properties, and with the mathematical formalisms for representing and reasoning about these properties. Nice surveys on agents can be found; we mention, for example, Bordini *et al*. (2006) and Fisher *et al*. (2007). Agents can then be organized into multi-agent systems (MAS).

Based on the wide success of the ASP approach in the artificial intelligence (AI) field, testified by the many works on theory and practice of ASP presented in the last years at the main AI Conferences, various approaches exist in the literature, which strongly support the adoption of ASP for the design of intelligent agents (please refer to Sections 4.2, 5). In the 'oldest' approaches, the use of ASP did mainly take the form of 'Action Description Languages,' which are formal models used to describe dynamic domains by focusing on the representation of effects of actions. In particular, an action specification represents the direct effects of each action on the state of the world, while the language semantics takes care of all the other aspects concerning the evolution of the world (e.g. the ramification problem). These approaches have been extended over the years in many ways in order to cope with, interpret, and recover from, exogenous events and unexpected observations. In subsequent work, ASP-based agent architectures have been proposed, where the description of both domain's behavior and agents' reasoning components is provided in ASP. The adoption of ASP is due to its ability to represent various forms of knowledge representation and reasoning including defaults, causal relations, statements referring to incompleteness of knowledge, etc. Later on, extensions of ASP to enable reactivity and forms of communication between agents have been introduced through, for example, the introduction of special named sets of fluents known as 'requests'. A different line of work focuses on modeling agent and MAS decisions in an extended ASP by means of game theory. Also, there is relevant work where ASP is exploited to model dynamic updates to an agent's knowledge base.

In our opinion, the ASP approach to problem-solving (based upon computing at each step the answer sets of given program), does not appear to be a fully appropriate modeling tool for the dynamic flexible functioning of agents as concerns reactivity, proactivity and communication. The superposition of such features 'on top' of the ASP approach appears forced. Thus, ASP-based architectures are under these aspects too rigid in comparison with other approaches to defining agents and MAS in computational logic, among which we mention at least METATEM, 3APL, AgentSpeak, Goal, Impact, KGP and DALI. All these architectures, and their operational models, are in practice more dynamic and flexible. Nonetheless, we believe that ASP can find a relevant role in agent-based applications, as it is suitable to represent important aspects of an agent's commonsense reasoning tasks. Such exploitation of ASP features in agents' design appears more smooth and feasible in the context of modular agent-oriented architectures. In our view in fact, an 'ideal' agent architecture should be able to exploit the potential of integrating several modules/components representing different behaviors/forms of reasoning, with these modules possibly based upon different formalisms. The 'overall agent' should emerge from dynamic, non-deterministic combination of these behaviors that should occur also in consequence of the evolution due to changes in an agent's environment.

As a practical example of such an integrated approach, Costantini (2011) advocated an agent architecture capable of smoothly incorporating several modules/components representing different forms of reasoning. In their architecture, the authors propose in particular to adopt ASP modules in the context of agent-oriented Prolog-like languages, and implemented this solution in the DALI language (cf. Costantini and Tocchio, 2002; Costantini & Tocchio, 2004). In a later paper (Costantini *et al*., 2015), they show the effectiveness of this solution by means of a case study where DALI agents cooperate in order to explore un

unknown territory upon occurrence of some kind of catastrophic event (earthquake, fire, flooding, terrorist attack, etc.). There, the DALI MAS in charge of managing the overall situation was equipped with ASP modules which provided flexible and affordable planning capabilities. In fact, there is significant research work on the exploitation of ASP modules within logical agents frameworks, where such modules are employed for tasks in which ASP is particularly well-suited: the provided results are filtered and exploited by agents.

As a matter of fact, approaches that rely on heterogeneous frameworks may have many interesting advantages: (i) different components (modules) of the system can be implemented using the most suitable language for the task at hand (task of the module); (ii) these approaches pave the way toward reusable modules libraries; (iii) they allow in principle parallel implementation of different parts; (iv) they naturally offer a structured view of the model. Approaches that do not provide support for integration of heterogeneous knowledge representation and reasoning technologies in a single agent limit their own applicability only to domains in which their particular technology is appropriate. In general in fact, the particular mental attitudes that a programming approach implements and the nature of interactions between these attitudes determines a certain programming style, which could hardly fit every particular domain. Instead, allowing the possibility to partition an agent's set of tasks into subsets and to choose for each subset an appropriate application-specific knowledge representation technology (with an associated reasoning technique) can more easily result in a flexible, scalable and robust (hybrid) agent system. Therefore, it should be left to the agent designer to choose the most suitable knowledge representation and reasoning technology for each task at hand. Thus, the underlying programming framework should be modular enough to accommodate a wide range of suitable tools. In this paper, we argue in favor of such a perspective, devoting particular attention to modular agent architectures based on Computational Logic, and in favor of the role that ASP might profitably play in such frameworks.

This paper is in fact intended as a position paper, aimed to: (i) revise the aspects of interest and mentioning the relevant literature; (ii) at the same time, propose a discussion to support our proposition. Specifically, the paper is organized as follows. We provide an overview on agents in Section 2; after shortly illustrating the concept of agency, we discuss the most prominent existing agent architectures and agent-oriented languages, and finally we shortly review MAS. Then, we dedicate Section 3 to agent-oriented software engineering (AOSE), that is crucial in view of complex and industrial applications. We provide an overview on the ASP approach in Section 4; specifically, after illustrating the basic aspects of the ASP paradigm, we shortly discuss software engineering issues for ASP, and the use of ASP in industrial applications. In Section 5, we review the application of ASP in agents and MAS. Section 6 is devoted to exploring the possibilities of integrating different computational approaches to agency. As, in our view, the key to doing so is modularity, we discuss modularity in logic programming, agents and ASP. Section 7 concerns modular distributed architectures, which are becoming increasingly important in view of Cyber-Physical Systems (CPS; cf. Khaitan & McCalley, 2015, for a survey) and of 'the Internet of Everything.' In Section 8, we propose a discussion about the possibility of integrating ASP and agent-based formalisms and frameworks on the basis of potential practical applications, outlining possible future directions and prospective developments. The paper is meant to be readable by the non-expert: so, it provides a wide perspective concerning all the issues involved. We assume some basic knowledge about logic and logic programming, and we often mention Datalog and Prolog; the reader may refer to Lloyd (1987), Apt and Bol (1994) and Przymusinski (1988) for comprehensive surveys.

2 An overview on agents

2.1 What is an agent

The definition of what is an 'agent' has been debated for quite some time, and researchers and developers tend to describe these entities in the context of specific domains of interest. According to Shardlow (1990), 'Agents do things, they act: that is why they are called agents.' Wooldridge *et al*. (1995) describe an agent as 'one who, or that which, exerts power or produces an effect,' exercised through actions. For further discussion, please refer to Franklin and Graesser (1996).

In the field of artificial intelligence, researchers are more concerned with 'Intelligent Agents,' that is, their objective is to add the concept of 'intelligence' in the description of agents' behavior. The original perspective in artificial intelligence focused on agents' reasoning process, thus identifying 'intelligence' with rationality. This view was strongly criticized by many researchers and proved in fact to be insufficient. A novel view of intelligent agents, able to be both rational and reactive, that is, capable of timely response to external events, was introduced in Kowalski and Sadri (1996). Later, Wooldridge and Jennings improved the definition of intelligent agent by adding the property of 'autonomy', and then of 'proactivity', in the sense of being able to take initiatives and to start and perform activities based on one's own decision. The reader can refer to Wooldridge *et al*. (1995) for a definition of agent that explicitly includes the concept of intelligence. Many other definitions appeared over time, emphasizing different properties like, for example, communication capabilities (Coen, 1994), mobility (Maes, 1993), typed-message agents (Petrie, 1996) or trying to classify software agents according to the tasks they perform (e.g. information gathering agents or email filtering agents) or according to their control architecture. Agents may also be classified by senses range and sensitivity, or by the actions range and effectiveness, or by how much internal state they are equipped with, etc. See, for example, a taxonomy of software agents in Brustoloni (1991), and the approach of Keil (1989). For a discussion about the theory of agency, the reader can refer to Wooldridge and Jennings (1994).

2.2 Agent architectures

An 'Agent Architecture' can be considered as the 'functional brain structure' of an agent in making decisions and reasoning for solving problems. An architecture describes an agent system's components, and how these components are organized to produce the overall agent behavior. A large number of architectures have been developed. Wooldridge *et al*. (1995) propose a description of an agent architecture following an abstract approach and introducing the concept of environment with which the agent interacts. It is also emphasized that agent architectures represent the move from specification to implementation. Kaelbling (1991) considers an 'agent architecture as a general methodology for designing particular modular decompositions for particular tasks.' Maes (1991) defines an agent architecture as a particular methodology for building agents. It specifies 'how the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact.' Sloman and Logan (1998) observes that a complete functioning agent, whether biological, or simulated in software, or implemented in the form of a robot, needs an integrated collection of diverse but interrelated capabilities, that is, an architecture.

Wooldridge *et al*. (1995) classify agent architectures into four categories:

- Logic-based architectures, where each decision is made through logical deduction.
- Reactive architectures, in which decision making is implemented via some form of direct mapping from situation to action.
- Belief–desire–intention (BDI) architectures, where an agent takes a decision according to the manipulation of data structures representing the beliefs, desires and intentions of the agent, that is, the informative, the motivational and the deliberative components of the system.
- Layered architectures, in which decision making is realized via various software layers.

We can add to these categories the cognitive architectures, which are based on the results of cognitive science, and aims at summarizing the results of cognitive psychology in a comprehensive computer model. Below we provide some more detail and references for each category.

2.2.1 Logic-based architectures

Also known as the symbolic-based or deliberative architectures, they are one of the earliest kinds of agent architectures. They are based on the symbolic artificial intelligence approach for representing and modeling the environment and the agent behavior. Examples of this architecture include classical planning agents such as STRIPS (Fikes & Nilsson, 1971), IPEM (Ambros-Ingerson & Steel, 1988), Autodrive (Wood, 1993), Softbots (Etzioni *et al*., 1994), the Phoenix systems (Cohen *et al*., 1989), IRMA (Bratman

et al., 1988), HOMER (Vere & Bickmore, 1990) and GRATE (Jennings, 1993). Despite the simplicity and elegance of logical semantics, this approach is somewhat problematic. First, the transduction problem; in fact, it is difficult to translate and model the environment's information into an accurate symbolic representation, especially for complex environments. Second, the adopted symbolic form should be suitable for the agents to reason in a time-constrained environment, which can be non-trivial, in relation to the computational complexity of theorem proving. Third, the transformation of percepts input may not be accurate enough to describe the environment due to certain faults such as sensor errors, reasoning errors, and etc.

2.2.2 Reactive architectures

In the mid-1980s, a new direction of thought emerged, strongly influenced by a refusal of symbolic AI. The reactive agent architecture is based on the direct mapping of situation to action. Brook's subsumption architecture is known as the best pure reactive architecture (Brooks, 1986), aimed to support intelligence via reactive behavior only. Related approaches are the 'situated automata' paradigm by Kaelbling (1991), and the Agent Network Architecture, where agent acts as a set of reactive 'competence modules,' presented in Maes (1991). An evolution of the subsumption architecture has been proposed by Togelius (2003). This approach combines incremental and modularized evolution with elements from the subsumption architecture. Other examples of reactive architectures are systems like Joseph Weizenbaum's Eliza and Agre and Chapman's Pengi. Reactive architectures offer many advantages like simplicity, economy, robustness and computational tractability. The disadvantages include: insufficient information about agent's current state, that may determine difficulties in actions activation; it is hard to see how reactive agents can learn from their experience; while small agent can be easily generated based upon a small number of behaviors, it is hard to engineer agents with large number of layers (behaviors). Dynamics of interactions become too complex to understand, and the development of the system may take a lot of time.

2.2.3 Belief–desire–intention architectures

Rao and Georgeff (1991) and Rao and Georgeff (1995) proposed an abstract architecture called BDI, standing for 'Belief, Desire, Intention,' that has become very successful and widely adopted both in practice, and as a conceptual reference. The BDI architecture is based on practical reasoning by Bratman Bratman (1999). Practical reasoning is reasoning toward actions—the process of figuring out what to do. In this architecture, an agent consists of three logic components referring to 'mental states' 'mental attitudes,' namely beliefs, desires and intentions. The *Beliefs* component represents knowledge about the world. An agent must also have information about the objectives to be accomplished; in BDI, these objectives are called *Desires*; desires represent goals that might in principle be selected for actual accomplishment. Finally, *Intentions* represent the current agent' course of action, that is, the objectives that are presently being pursued. Since the first formalization, the BDI model has evolved, thus generating several related approaches, as summarized in Georgeff *et al.* (1998). The design of the BDI architecture is clear and intuitive. However, the question is how to effectively implement its basic concepts and how to achieve a balance between commitment (to intentions and to specific plans) and reconsideration, especially in resource-bounded agents acting in a changing environment. Specification, design and verification of BDI agents have received a great dea: Dagli Oggetti agli Agenti. 6th AI*IA/TABOO Joint Workshop 'From Objects to Agents': Simulation and Formal Analysis of Complex Systemsl of attention. Although beliefs, desires and intentions represent a nice method to model intelligent agents in a dynamic environment, the BDI architecture is semantically too complex to be fully and efficiently implemented (Rao, 1996). Thus, the BDI model has been integrated with other approaches from computational logic. This integration resulted in the languages AgentSpeak(L) (Rao, 1996), 3APL (d'Inverno *et al.*, 2000) and others (cf. Mascardi *et al.*, 2005, for a review).

2.2.4 Layered (hybrid) architectures

Layered architectures respond to the need of having agents that exhibit different kinds of behavior. These architectures combine the advantages of both reactive and logic-based architecture. An agent can be

thought of as a system consisting of separate subsystems, decomposed into a layered structure, where each of them implements a particular behavior. Usually, in this architecture subsystems are arranged into two layers, reactive and proactive, but it is possible to have as many layers as the desired behaviors. Wooldridge (1999) notes that this kind of architectures may have two types of control flow: horizontal, and vertical. The problem with horizontal architectures is the complexity of interactions between layers. With vertical layered architectures this problem is solved because the control flows pass sequentially through each layer. The limit of the vertical architecture is however the lack of flexibility: in order to make a decision, control must pass through all layers and failure of one of them implies the failure of the overall decision process. An example of horizontal layered architecture is the TOURINGMACHINES of Ferguson (1992). InteRRaP is an example of vertically two-pass agent architecture (cf. Müller & Pischel, 1994). Another relevant architecture adopting the layered approach is JADE (cf. JADE Website, 2016). A JADE evolution towards the BDI model is JADEX (Pokahr *et al*., 2005). JADEX is an implementation of a reactive and deliberative architecture, integrating into JADE 'mentalistic' concepts typical of BDI model.

2.2.5 Cognitive architectures

These kind of architectures are aimed to construct intelligent systems/agents that models human performance, like in Newell (1990), Meyer and Kieras (1997). A cognitive architecture is a theory about the fixed computational structure of cognition (see, e.g. Anderson and Lebiere, 1998; Newell, 1990; Pylyshyn, 1990). The origin of cognitive architectures lies in 'production systems' (Newell, 1973; Neches *et al*., 1987), which can be used to form a 'recognition memory' or 'production memory' (for explanation please refer to the book 'Unified Theories of Cognition' by Newell (1990: 165, 486). Memory and learning are the two important key components in developing cognitive architecture. By combining these two components we can distinguish three main categories of cognitive agent architectures: symbolic, emergent, and hybrid models, as illustrated in Duch *et al*. (2008). Examples of cognitive symbolic architectures are SOAR (cf. Laird *et al*., 1987; Newell, 1990), EPIC (cf. Meyer & Kieras, 1997), ICARUS (cf. Langley, 2005). A comprehensive review of implemented cognitive architectures has been undertaken in Samsonovich (2010). For another review, the reader can also refer to Langley *et al*. (2009).

2.3 Agent-oriented programming languages

We now move to talk briefly about AOP, that is, the programming paradigm that allow developers to program agents which are closely related to a certain agent architecture. An AOP language should provide support to a wide range of key agent characteristics like autonomy, communication, distribution, concurrency, naming, mobility, decision making, representation of mental attributes and knowledge, etc. Since the seminal paper of more than two decades ago by Shoham (1993) on AOP, many specialized programming languages for the development of agents with mental states have been proposed. For surveys, the reader can refer to Bordini *et al*. (2006), Fisher *et al*. (2007), Mascardi *et al*. (2005). The first attempt to design an AOP was AGENT-0 (Shoham, 1993). AGENT-0 was refined in Thomas (1993), resulting in the 'Planning Communicating Agents' (PLACA) language. Concurrent METATEM language was developed by Fisher (1994) to address the drawbacks of both AGENT-0 and PLACA.

Many other agent oriented languages have been developed so far, with these languages following two main lines. On the one hand are the pragmatic approaches, where the most prominent examples are JACK (JADE Website, 2016) and Jadex (Pokahr *et al*., 2005), both based on Java as the underlying language. These approaches enjoy the practically proven object-oriented syntax, vast range of libraries and easy integration with legacy and third-party systems, which make them very popular among practitioners. However, a drawback of these approaches is the difficult integration of specialized knowledge-representation techniques.

On the other hand are the theoretically driven approaches, mainly based upon Computational Logic and Prolog, which aim at designing a stand-alone agent programming language using mental attitudes and knowledge representation as primary concepts. We consider this line as particularly interesting because Computational Logic brings the advantages of clarity, where the code can be understandable even by the

non-expert, and of verifiability of agents, MAS and protocols via formal techniques. In fact, agent models have been implemented for a long time by means of imperative programming languages, mainly for the sake of computational efficiency. While, however, efficiency may not always be an issue, while instead clear specification and correctness are. In addition, declarative approaches are suitable to handle the complexity of agent systems, where the efficiency and effectiveness of implementations is constantly improving. So, the logical approaches to building agents have now become effective in the direction of practical usability in real-world applications. Well-established languages based upon Computational Logic are 3APL (d'Inverno *et al*., 2000), AgentSpeak(L) (Rao, 1996; d'Inverno & Luck, 1998), GOAL, IMPACT (cf. Dix *et al*., 2000; Rogers *et al*., 2000), DALI (cf. Costantini & Tocchio, 2002; Costantini & Tocchio, 2004; Costantini & Tocchio, 2005). More generally, there are approaches fully based upon declarative paradigms, and others depending partially on computational logic (hybrid approaches which combine both declarative and imperative features) for both single-agents and MAS. For a detailed discussion of different computational programming approaches, the reader may refer to the surveys (Dastani, 2015; Bordini *et al*., 2006; Fisher *et al*., 2007).

2.4 Multi-agent systems

Agents are social entities:696 they exist in an environment with other agents, with which they will generally be expected to interact in some way. Agents can thus be organized into MAS composed of either homogeneous or heterogeneous agents (the latter being defined via different agent-oriented languages and architectures). According to the seminal work of Jennings *et al*. (1998), the features of MASs are the following: (i) each agent has incomplete information, or capabilities for solving the problem; (ii) there is no global system control; (iii) data are decentralized; (iv) computation is asynchronous.

Interaction is crucial in MAS, as it allows agents to share information which is necessary to achieve their goals. Often, communities of agents interact in order to resolve a common problem; sometimes, however, agents have egoistic interests to pursue independently or even in competition with others. These two modalities of interaction, cooperation and competition, has been treated for a long time in Distributed Artificial Intelligence.

MAS began to invade the scientific and industrial world since the 1980s. In fact, their ability to adapt themselves flexibly and intelligently to different contexts determined the development of interesting applications. From the mid-1980s, many control architectures for MAS have been developed, that we do not discuss here as MASs are not a central topic of this paper. For surveys, the reader may refer to Jennings *et al*. (1998), Baldoni *et al*. (2010), Torroni (2004). The wide applicability of agents and MAS has fostered the upcoming (since the seminal work of Wooldridge (1997)) of AOSE, which is a software engineering paradigm aimed at applying best practice in the development of such systems. AOSE is focused on the use of agents and organizations (communities) of agents as the main abstractions.

3 Agent-oriented software engineering

Software engineering is crucial for real-world (industrial) applications, as software systems are required to operate in an increasingly complex, distributed, large, open, unpredictable, heterogeneous and highly interactive applications environments. Agent-based computing is considered to be very promising and natural for building these systems in terms of MAS. AOSE has emerged from the interaction among agent-oriented computing and software engineering and can serve as a useful paradigm for the development of agent-based systems. There exist several methodologies for AOSE; many of them argue the adequacy of OOSE (Object-Orientes Software Engineering) to model AOSE and give emphasis to 'agentifying' software entities, namely turning software components (agents and non agents) into agent entities. Among these methodologies are Gaia (Wooldridge *et al*., 2000) and SODA (Omicini, 2001). Other approaches are based on object-oriented methodologies which provide an engineering process based, vice versa, on 'objectifying' the software system components, so as to model all components at the same level of abstraction. Examples of such approaches are MaSE (Wood & DeLoach, 2001) and KGR (Kinny *et al*., 1996). Other approaches give emphasis to the extension to AOSE of modeling techniques developed for

OOSE, specifically Unified Modeling Language, which becomes Agent-Oriented Unified Modeling Language (cf. Bauer *et al*., 2001).

Since AOSE combines software engineering and agency theory, it is necessary to establish evaluation criteria to help assess the suitability of a technique or a language for industrial software development. Such criteria are required to be coherent with software engineering objectives, while keeping agents' characteristics into account. General evaluation criteria concentrate on features that should be naturally supported by any given methodology Juneidi and Vouros (2004) and usually include the following.

- Preciseness: the semantics of a modeling technique must be unambiguous.
- Accessibility: modeling techniques should be understandable to both experts and novices.
- Expressiveness: a modeling technique should be able to present the structure of the system, the encapsulated knowledge, the data flow, the control flow within the system and also the interactions with other systems.
- Modularity and objectiveness: the ability to articulate a modeling technique into stages.
- Complexity management: a modeling technique should be expressed and then examined at various levels of detail.
- Executability and testability: prototyping capacity or simulation capacity.
- Refinability: ability to provide guidelines for refining a model.
- Analizability: availability of tools to check internal consistency.
- Openness and portability: implementation issues should be left to the programmers without obliging them to adopt a specific architecture or language.
- Formalization: modeling constructs should be complete and comprehensive, so as to instruct or even generate implementation.

Agency characteristics must however be checked in a given AOSE methodology, by means of at least the following criteria.

- Autonomy: the modeling technique should support the capability of describing an agent's self control feature.
- Complexity: agent-oriented systems are complex systems consisting of sets of components (agents) that interact with each other to achieve their goals. These systems may exploit many complex mechanisms and algorithms (learning, reasoning, decision making, etc.). Modeling them requires strong expressive power, so a modeling technique should support such expressiveness. Moreover, the complexity feature requires that a modeling technique should be modular, and able to support complexity management.
- Adaptability: agent-oriented systems should be flexible enough to adjust their activities to the dynamic environmental changes; so, the adaptability feature requires the modeling technique to be modular and able to specify activation of each component according to the environmental state.
- Concurrency: an agent may need to perform several tasks concurrently; for this reason, some agent-oriented systems must be designed as parallel processing systems. So, this requires the ability to express parallelism and concurrency in design and implementation stages.
- Distribution: MAS are sometimes distributed over the network. This requires the ability to express distribution in design and implementation stages.
- Communications richness: a modeling technique should be able to express a characterization of communication in order to produce agent communication commands or sentences during the implementation stage.

For logical agents, relevant existing work includes JaCaMo (http://jacamo.sourceforge.net), which is a methodology for the design and implementation of agents and MAS based upon the AgentSpeak Rao and Georgeff (1991) language (a very popular language based on the BDI agent model) under the Jason interpreter Bordini and Hübner (2006), Bordini *et al.* (2007), integrated with CArtAgO, which is a platform for programming distributed artifact-based environments Ricci *et al.* (2007), according to the 'Agents and Artifacts' metamodel. MAS in JaCaMo should be built in accord to the Moise organizational model Hübner *et al.* (2007) where a MAS is considered to be an organization structured as agents groups and sub-groups, in term of their roles and objectives. A survey about AOSE and its prospective employment in industry can be found in Akbari (2010). The reader may also refer to the EMAS series of workshops, http://emas.in.tu-clausthal.de/.

4 Answer set programming: an overview

4.1 The answer set programming approach

ASP is a logic programming paradigm under answer set (or 'stable model') semantics Gelfond and Lifschitz (1988), which applies ideas of autoepistemic logic Moore (1985) and default logic Reiter (1980). ASP features a highly declarative and expressive programming language, oriented towards difficult search problems. It has been used in a wide variety of applications in different areas like problem solving, configuration, information integration, security analysis, agent systems, semantic Web and planning (for more information about ASP and its application the reader can refer, among many, Baral, 2003; Borchert *et al.*, 2004; Leone, 2007; Truszczyński, 2007; Gelfond, 2008 and the references therein) and, more generally, to represent combinatorial problems in knowledge representation and automated reasoning.

ASP has emerged from interaction between two lines of research: first on the semantics of negation in logic programming, and second on applications of satisfiability solvers to search problems Kautz and Selman (1992). It was (independently) identified as a new programming paradigm in Lifschitz (1999), Marek and Truszczyński (1999), Niemelä (1999). In ASP, search problems are reduced to computing answer sets (or 'stable models' as they were originally denominated), and an answer set solver (i.e. a program for generating stable models) is used to find solutions (if any exists). The search algorithms adopted in the design of answer set systems are mostly enhancements of the Davis-Putnam Logemann-Loveland procedure. Unlike SLDNF resolution employed in Prolog, such algorithms, in principle, always terminate.

The expressiveness of ASP, the readability of its code and the performance of the available 'solvers' gained ASP an important role in the field of artificial intelligence; ASP applications have in fact been developed in several fields, as seen below. In the answer set semantics Gelfond and Lifschitz (1988, 1991), a (logic) program Π is a collection of *rules* of the form: $H \leftarrow L_1, \ \ldots \ , L_n$

The left-hand side and right-hand side of rules are called *head* and *body*, respectively, where H is an atom, $n \geq 0$ and each literal L_i either an atom A_i or its *default negation* $notA_i$. A rule can be rephrased as $H \leftarrow A_1, \ \ldots \ , A_m, notA_{m+1}, \ \ldots \ , notA_n$ where $A_1, \ \ldots \ , A_m$ can be called *positive body* and $notA_{m+1}, \ \ldots \ , notA_n$ can be called *negative body*. In practical programming, ' $\leftarrow$ ' is usually indicated as ':–'. A rule with empty body ($n = 0$) is called a *unit rule*, or *fact*. A rule with empty head, of the form $\leftarrow L_1, \ \ldots \ , L_n$, is a *constraint*, and states that literals $L_1, \ \ldots \ , L_n$ cannot be simultaneously true in any answer set.

The answer sets semantics Gelfond and Lifschitz (1988, 1991) can be considered a view of a logic program as a set of inference rules (more precisely, default inference rules), or, equivalently, a set of constraints on the solution of a problem: each answer set represents a solution compatible with the constraints expressed by the program. Consider simple program: $\{q \leftarrow not\ p\ p \leftarrow not\ q\}$. The first rule is read as 'assuming that p is false, we can *conclude* that q is true.' The second rule is analogous though complementary, as truth of p depends upon assuming falsity of q. This program has two answer sets. In the first one, q is true while p is false; in the second one, p is true while q is false. So, negation in this paradigm is understood as *default negation*, that is, *nota* indicates the (default) assumption that a is false and thus its negation holds. Unlike other semantics, a program may have several answer sets or may have no answer set, where however every answer set is a minimal model of given program[1], and thus the answer sets form an anti-chain. Whenever a program has no answer sets (as some minimal models may not be answer sets), the program is said to be inconsistent . For instance, the following program has no answer set: $\{a \leftarrow notb\ b \leftarrow notc\ c \leftarrow nota.\}$. The reason is that in every minimal model of this program there is a true atom that depends (in the program) on the negation of another true atom, which is strictly forbidden in this semantics, where every answer set can be considered as a self-consistent and self-supporting set of consequences of a given program. The program $\{p \leftarrow not\ p\}$ has no answer sets either as it is contradictory not in the sense of classical logic, but in the sense that p cannot be derived from *notp*, that is, from the (default) assumption that p itself is false. Constraints of the form defined above can be in fact simulated by plain rules of the form $p \leftarrow notp, L_1, \ \ldots \ , L_n$. where p is a fresh atom. Consistency of an ASP program is

[1] By 'minimal model of program Π' we mean a minimal model of Π intended as a classical first-order logic theory, where $\leftarrow$ is intended as implication, the comma as conjunction, and *not* as negation in classical logic terms.

related (as discussed at length in Costantini, 1995, 2006; Costantini *et al*., 2002) to the occurrence of 'odd cycles' (of which $p \leftarrow notp$ is the basic case, though odd cycles may involve any odd number of atoms) and to their connections to other parts of the program. The reason is that, in principle, the negation *notA* of atom *A* is an assumption, that must be dropped whenever *A* can be proved, as answer sets are by definition non-contradictory. Many extensions of the original 'Answer Set Prolog' have been proposed, mainly motivated by applications. Some of them are syntactic sugar and some other strictly adds expressiveness to the language. For details about these extensions, the reader can reefr to Gelfond (2008), Gelfond and Kahl (2014,) and to references therein.

In the ASP paradigm, as mentioned each answer set is seen as a solution of given problem, encoded as an ASP program (or, better, the solution is extracted from an answer set by ignoring irrelevant details and possibly re-organizing the presentation). So, differently from traditional logic programming, the solutions of a problem are not obtained through substitutions of variables values in answer to a query. Rather, a program Π describes a problem, of which its answer sets represent the possible solutions. To find these solutions, an ASP-solver is used. Several solvers have become available, see *Answer Set Programming Solvers* (2016), each of them being characterized by its own prominent valuable features. Recently 'meta-solvers' such as multi-engine ASP system Maratea *et al.* (2015) have been developed, which select the most appropriate solver according to specific syntactic features of given program. The expressive power of ASP and its computational complexity have been deeply investigated (cf., e.g., Dantsin *et al*., 2001).

4.2 Industrial answer set programming applications

In this section we intend to shed light on some of the many challenges addressed by ASP in industrial applications. ASP-based software development provides many advantages, such as flexibility, readability, extensibility and ease of maintenance due to its powerful and expressive framework. One of the success factors of ASP-based software development in many industrial applications has been the flexibility in developing complex reasoning tasks by simply editing a text file containing the ASP rules, and testing it on-site together with the customer. This has been evident, for example, for the workforce-management application concerning the employees of the Gioia Tauro seaport (Ricca *et al*., 2012),where the main obstacle was the high complexity of the requirements; the use of ASP as an executable specification language made the process of clarifying and formalizing the requirements much easier. This in addition to the advantages of lower (implementation) costs by realizing complex features of an application in such a way, compared to traditional imperative languages.

Another challenge in industrial applications addressed by ASP is computational efficiency, which is handled by optimization techniques implemented by ASP solvers. For instance, in the Intelligent call-routing application (the company 'Exeura' developed a platform for customer profiling for phone-call routing based on ASP that is called *Zlog* (n.d.), an immediate response has to be given to queries over huge data sets. The DLV solver adopts the Magic Set optimization (Alviano *et al*., 2012) technique can localize the computation to the small part of the database that is relevant for the specific query at hand; this leads to a huge speedup of the computation. Telecom Italia employs the zLog platform in a production system that handles its call centers.

In many other applications of ASP, concerning knowledge representation and reasoning, robotics, and bioinformatics and computational biology as well as some other related applications, ASP addresses different types of challenges. For instance, some of the important challenges in knowledge representation and reasoning concern the representation of defaults to handle exceptions and the commonsense law of inertia, so as to be able to reason about the effects of actions. In robotic applications ASP addresses other type of challenges like hybrid reasoning, reasoning about commonsense knowledge and exceptions, optimizations over plans or diagnoses; Erdem *et al*. (2012) uses ASP to plan the actions of multiple robots that collaborate to organize a house within a specific time and Erdem *et al.* (2013) uses ASP to find an optimal global plan for a number of teams of heterogeneous robots in a cognitive factory to manufacture a number of orders within a specific time. In Erdem *et al*. (2015), ASP is exploited for diagnosing plan failures during monitoring of plan execution. ASP is also used for geometric rearrangement of multiple movable objects on a cluttered surface in Havur *et al*. (2014), where locations of objects can be changed by

pick and/or push actions. Zhang *et al.* (2015) use ASP to describe objects and relations between them, where this knowledge is used to improve localization of target objects. Provability of computational problems formulation, expressing sophisticated concepts that require recursion and/or aggregates, and integration of heterogeneous knowledge are some of many challenges addressed by ASP in bioinformatics. For example, Nam and Baral (2009) introduce a method to model a biological signaling network as an action description in ASP which allows for prediction, planning and explanation generation about the network. Gebser *et al.* (2011) introduce a method to model biochemical reactions and genetic regulations as influence graphs in ASP, to detect and explain inconsistencies between experimental profiles and influence graphs. In Brooks *et al.* (2007), ASP was used to solve the problem of reconstructing phylogenies for specified taxa, with a character-based cladistics approach. Erdem *et al.* (2011) and Erdem and Öztok (2015) use ASP to answer complex queries over biomedical ontologies and databases, and to generate the shortest explanations to justify these answers. Many similar challenges are also addressed by ASP in other industrial applications, such as data cleaning, extraction of relevant knowledge from large databases and software-engineering challenges.

ASP has been profitably applied in applications concerning tourism industry. For instance, Ricca *et al.* (2010) has developed an intelligent advisor to select the most promising offers for travel agency's customers. ASP has been used as the intelligent engine of a semantic-based information extractor (cf. Manna *et al.*, 2011), which analyzes touristic offers, extracts the relevant information (e.g. place, date, prize), and classifies them in an ontology. In addition, ASP has been used for developing several search modules which simplify selecting the holiday packages that best fit the customer needs. This helps the employees of a travel agency to find the best possible travel solutions in a short time.

ASP has been successfully employed in context of e-Medicine, in particular for data-cleaning of medical knowledge bases. These knowledge bases are in fact the result of the integration of different databases: thus, they often present errors and anomalies that severely limit their usefulness. Leone and Ricca (2015) developed a data-cleaning system, based on ASP and called DLVCleaner; this system detects and automatically corrects both syntactic and semantic anomalies in medical knowledge bases on the basis on ontological domain descriptions and measures for string similarities (cf. Greco & Terracina, 2013). In this application, ASP allows for a simplified and flexible specification of the logic of the data-cleaning task.

One of the first industrial applications of ASP was for product configuration (Tiihonen *et al.*, 2003), used by Variantum Oy. More recently, the group of Gerhard Friedrich at Alpen-Adria Universität Klagenfurt, Austria has developed a configuration and reconfiguration application, deployed by Siemens. In particular, ASP has been used for configuration of railway safety systems. This configuration problem is NP-hard and has proved challenging for problem-solving frameworks such as SAT, constraint programming, MIP and ASP (cf. Aschinger *et al.*, 2011). Thus, ASP made it possible to solve configuration problems that could not be solved by specialized configuration tools. ASP is applied to model possible changes of existing systems and to compute reconfiguration solutions that optimize the adaptation actions. For example, in Friedrich *et al.* (2011) the aim is to maximize the number of reused modules and minimize the costs of additional equipment. In addition to configuration tasks, ASP was applied in Friedrich *et al.* (2010) to diagnose and repair systems. For more details on ASP applications, the reader can refer Erdem *et al.* (2016), Grasso *et al.* (2013) and to the references therein.

4.3 Software engineering for answer set programming

ASP has become one of the most increasingly used tools for knowledge representation and reasoning; in fact, ASP has proved to be very effective in developing complex knowledge intensive applications at a lower price (for implementation) than the imperative languages. In addition, it has been observed by developers that the use of ASP brings many benefits from the point view of software engineering practices like expressiveness, flexibility, extensibility, readability, ease of maintenance, etc.

However, developing complex systems in any programming language is not an easy task, and ASP is not an exception. In fact, developers have faced many practical obstacles, due to the shortage of development tools and software engineering methodologies tailored particularly for ASP. This issue was

recognized by the research community, thus the SEA (Software Engineering and ASP) series of workshops was initiated, which provide an international forum to discuss software engineering problems that the field may experience either currently or in perspective. The aim of this forum is to start a general discussion on the requirements and specification of input, output and intermediate languages for answer set solvers. In addition, there has been the development of many interfaces, frameworks and development tools like APIs, ASPIDE, Jazzyk, and JDLV to ease the use of ASP in real world applications.

The first step in addressing the above-mentioned need was the development of APIs (cf. Gallucci & Ricca, 2007; Ricca, 2003) which offer a method for interacting with ASP solvers from an embedding Java program. This is relevant since ASP is not a general purpose language; thus, ASP programs often need to be embedded within components built by using some imperative language, for interfacing with the environment or for user interaction. ASPIDE (cf. Febbraro *et al*., 2011) is an IDE that supports the entire life-cycle of the development of ASP-based applications. ASPIDE is written in Java and includes a DLV wrapper (cf. Ricca, 2003) for interacting with DLV, which is one of the most famous and well-developed ASP solvers (http://www.dlvsystem.com/). SeaLion (sealion.at) is an Integrated Development Environment for Answer-Set Programming which comes as an Eclipse plugin and aims at offering support to write, evaluate, debug and test answer-set programs.

JDLV (available at http://www.dlvsystem.com/dlvsystem/index.php/JDLV), is an advanced platform for integrating Java with DLV, which comes as a plug-in for the Eclipse platform, offering a seamless integration of ASP-based technologies within the most popular development environment for Java. JDLV is based on JASP (cf. Febbraro *et al*., 2012), a hybrid language that transparently supports a bilateral interaction between ASP and Java. JASP embeds ASP code in a Java program, where a logical ASP module can access Java variables; the answer sets resulting from the execution of ASP code are automatically stored in Java objects, where the collections of Java objects are mapped to ASP facts. In addition, JASP allows a programmer to include arbitrary Java expressions in logic rules, to be evaluated at runtime.

5 Answer set programming and agents

A large number of papers have proposed the application of ASP in the modeling of intelligent agents. In fact this is not surprising, since ASP was developed specially for non-monotonic reasoning, including forms of common-sense reasoning required in agent modeling (cf. Gelfond & Kahl, 2014). In this section we provide a review of such proposals.

5.1 Agents in answer set programming

ASP have been used in agents since quite a long time, though this use was initially mainly in the form of action description languages. They were first introduced in Gelfond and Lifschitz (1998) and in Baral and Gelfond (2000), where a simple ASP-based agent architecture was proposed. This work was then extended and refined by many authors: Balduccini and Gelfond (2003), Balduccini *et al*. (2006), Balduccini (2007a). This architecture is further refined and described in Balduccini and Gelfond (2008), where it is named ‘AAA’ (Autonomous Agents Architecture) and is intended for intelligent agents acting and reasoning in a changing environment. The agent behavior is based on the simple control loop of the kind that Kowalski, in Kowalski and Sadri (1996), called ‘observe-think-act’ cycle. The description of domain’s behavior and of the reasoning components are written in Answer Set Prolog. The architecture gives the agent capabilities of planning, detecting, interpreting and recovering from, unexpected observations. In addition, the design and the knowledge bases are elaboration-tolerant. Another special feature of the architecture is that the same domain description is shared by all the reasoning modules. A prototype of the implementation of the architecture can be found at: http://mbal.tk/APLAgentMgr/

Building upon this work, a new architecture, AIA, was proposed in Blount *et al*. (2015). The AIA framework is an architecture for intelligent agents whose behavior is driven by their intentions and that reason about, and act in a changing environment. The domain description includes both the agents’ environment and a general Theory of Intentions. The main difference between AAA and AIA architectures is in the organization of the control loop that is in AIA centered around the notion of intention, which is

something absent in AAA. The AIA architecture is meant to enable agents explain unexpected observations and decide the intended action at the present moment. Reasoning tasks are reduced to computing answer sets of CR-Prolog programs constructed automatically from an agent's knowledge. A prototype can be found at http://www.depts.ttu.edu/cs/research/krlab/software-aia.php. An evolution of the AAA architecture was proposed for UAVs (unmanned aerial vehicles) operation in Balduccini *et al.* (2014). There, both agent architecture and reasoning algorithms based on ASP were developed. ASP has been chosen for this task because it enables high flexibility of representation, both of knowledge and of reasoning tasks. This work is part of a project aimed at investigating how traditional AI reasoning techniques can be extended to coordinate teams of UAVs in dynamic environments. This work demonstrates the reliability and performance gains deriving from network-aware reasoning. Overall, this work was the first practical application of a complete ASP-based agent architecture. It is also the first practical application of ASP involving a combination of centralized reasoning, decentralized reasoning, execution monitoring and reasoning about network communications.

Jazzyk (cf. Novák, 2009) is a hybrid, special-purpose programming language for development of knowledge intensive (intelligent) agent systems. Jazzyk agents consist of: a number of knowledge bases, each realized by a separate specialized knowledge representation module (called 'plug-in'), and an agent program in a form of a set of possibly nested rules of the basic form: 'when Query then Update'. Jazzyk integrates an ASP solver for handling ASP modules.

5.2 Answer set programming and multi-agent systems

In the realm of MAS, ASP has been used for defining and for modeling aspects of MASs, for reasoning about these systems, and for configuration of (features of) MAS. De Vos and Vermeir (2004) presents systems composed of logic programming agents systems (LPAS) to model the interactions between decision-makers in the process of reaching a conclusion. The system consists of a group of agents connected by means of unidirectional communication channels. These agents communicate with each other by passing answer sets obtained by updating the information received from connected agents with their own private information. The authors introduce a simple answer set semantics for LPAS. This kind of systems is useful for modeling decision problems. As an application, the authors show how extensive games with perfect information can be conveniently represented as LPAS, where each agent embodies the reasoning of a game player, such that the equilibria of the game correspond with the semantics agreements among the agents in the LPAS.

Amongst the most demanding properties with respect to the design and implementation of MAS systems is how the agents may individually reason about and communicate their knowledge and beliefs, within a perspective of cooperation and collaboration. Here it is vital for the agents to be able to verify the reliability of information providers especially when the pieces of information coming from different sources contradict each other. In De Vos *et al.* (2005), the authors present a hybrid multi-agent platform, called T-LAIMA, using an extension of ASP. They show that their framework is capable of dealing with the specification and implementation of the system's architecture, and with the communication and the individual reasoning capabilities of the agents. Agents use ASP to represent their reasoning capabilities. Agents interact with each other, and via such interactions the agents may update trust values, make or break connections with other agents, etc. As steps in the agent loop, the agent may also choose goals to try and satisfy, plan to achieve such goals, and can also make observations, perform diagnosis, execute actions, etc.

Buccafurri and Gottlob (2002) introduced a new notion of 'compromises' for MAS desiring to reach a common decision using joint fixpoints and stable models. In a multi-agent setting, a joint decision of the agents, reflecting a compromise of the various requirements, corresponds in this approach to a suitable joint model of the respective ogic programs. The authors proposed an appropriate semantics, namely the joint fixpoint semantics, where intended joint models are defined to be the (minimal) joint fixpoints of the agent programs. This work was then extended in Buccafurri and Caminiti (2005) to model social behavior among a community of agents represented by logic programs. This by focusing on a language derived from logic programming which supports the representation of mental states of agent communities

and provides each agent with the capability of reasoning about other agents' mental states and acting accordingly. The proposed semantics is shown to be translatable into stable model semantics of logic programs with aggregates.

'Communicating answer set programming' (CASP) is an intuitive extension of ASP, in which multiple ASP programs collaborate to solve some problem. The expressiveness of CASP has not been thoroughly studied yet. Bauters *et al*. (2013) presents a systematic study of the additional expressiveness offered by allowing ASP programs to communicate. This paper aggregates and extends the work from Bauters (2011b) and Bauters *et al*. (2010).

Van Nieuwenborgh *et al*. (2006) proposed a multi-agent framework that is capable of modeling hierarchical decision problems. This framework is a multi-agent formalism based on extended ASP. The system consists of independent agents connected via a communication channel, where knowledge and beliefs of each agent are represented by a logic program. The resulting semantics is very expressive, and it essentially captures the polynomial hierarchy, thus enabling complex applications.

In Cliffe *et al*. (2006), the authors describe the use of ASP as an institutional modeling technique. Basically, they demonstrate how formal specifications of institutions for the purpose of obligations, permissions and violations, can be translated into ASP programs in such a way that ordered traces which record their actions can be obtained as answer sets. These answer sets have a one-to-one relationship with the institutional event traces of the formal model. This provides an easy way to inspect properties of models. The ability to reason about and query time-related information is a strong point for using ASP. In contrast to symbolic model checking, ASP allows the statement of problems and queries in domain-specific terms as executable logic programs therefore eliminating the gap between specification and verification language.

Resource allocation in MAS is a growing area of research. It involves many aspects: economical, game-theoretical and purely computational ones. The work of Leite *et al*. (2009) is concerned with computational aspects of resource allocation, particularly concerning complex combinatorial problems. In spite of the extensive theoretical work and the raising number of practical applications, many fundamental problems in multi-agent resource allocation still lack proper solutions. Here, the authors show how ASP can be used for providing general and flexible solutions of these problems in a compact and declarative manner.

Baral *et al*. (2010) discussed and explored the use of ASP for defining action techniques that address one of the most challenging aspects of reasoning, namely planning and acting in a multi-agent domain: which is, reasoning about what the agents know about the knowledge of their fellows, and take such knowledge into account when planning and acting. This approach presents a number of advantages; from the theoretical perspective, the property of nonmonotonicity of ASP (and many other features) allows one to express causality in an elegant way; from the practical point of view, recent implementations of ASP solvers have become very efficient. Finally, the use of ASP for reasoning about action techniques allows for the adaptation of a large body of research developed for single-agent to multiagent domains.

Bauters (2011a) presents a case study exploring how one can use 'multi-focused answer sets' to model the process of coalition formation; via multi focused answer sets, one can impose constraints so as to reach a unique agreement, represented by an answer set, that optimizes the preferences of the parties involved. This work uses CASP to encode preferences and soft constraints. CASP allows in fact a number of ASP programs to collaborate by asking each other questions. In CASP, there is no longer a unique way to define a stable model. By using a mechanism called 'focusing' it becomes possible to eliminate some of these stable models, that is, to assign preferences to the stable models of a CASP program.

In Ianni *et al*. (2003), the authors propose an innovative architecture where they integrate a framework based on ASP into an Information Retrival Agent named GSA (Global Search Agent, illustrated in Castellucci *et al*., 2001; Ianni, 2001). In order to improve the original system effectiveness, the 'GSA2' system introduces a new internal architecture based on a message-passing framework and on an ontology description formalism (WOLF (Web Ontology Framework)). The system consists of a set of cooperating agents where each one is a specialized module. The role of central intelligence is played by a reasoning system based on ASP, and makes the agents able to take independent decisions. The high expressive power of ASP allows to describe program and plan behaviors of the agents easily and quickly, and to

experiment with any (even future) information retrieval strategy. Both the system architecture and WOLF are very general and reusable, and constitute a very good and interesting example of actual exploitation of ASP for real applications.

In Osorio *et al.* (2005), the authors propose the use of ASP and of the argumentation framework proposed by Dung (1995) to represent a medical transplantation knowledge base. They define CARREL-ASP, namely CARREL Cortés *et al.* (2005) extended with ASP, to perform decision making based on argumentation framework.

In Amendola *et al.* (2016), the authors propose a compact representation for the non-transferable utility games based on ASP. The representation is fully expressive, in that it can capture all games defined over a finite set of alternatives. The approach can easily accommodate the definition of games within wide range of application domains, ranging from scheduling to routing and planning, etc.

In Son *et al.* (2016), the authors present a framework based on ASP to reason about the truthfulness of statements made by an agent. The framework does not assume complete knowledge about the agent being observed and the reasoning process builds on observations about the state of the world and the occurrences of actions. The authors explore the use of their framework in the detection of Man-In-The-Middle attacks targeting computers and CPS.

In Nogueira *et al.* (2001), authors describe a medium-sized decision support system for shuttle flight controllers written in ASP. The system consists of a collection of largely independent modules represented by lp-functions and a graphical Java interface.

A general overview of approaches investigating and specifying methodologies for modeling agents by means ASP can be found in Gelfond (2004). Overall however, with due recognition to the wide corpus of existing work it is possible to observe that the features of the ASP approach (4) are not fully appropriate to modeling the dynamic flexible functioning of agents as concerns reactivity, proactivity and communication. The superposition of such features 'on top' of the ASP approach appears forced, awkward and not fully satisfactory. Instead, ASP can clearly model important aspects of agents, concerning their commonsense reasoning and planning tasks. So, in our opinion ASP can find its relevant place in agent architectures provided that such architectures are sufficiently well-structured and flexible to accommodate and integrate ASP components (as done, e.g. in Costantini, 2011; Costantini *et al.*, 2015).

6 Integrating different computational logic approaches in agents: discussion

In our view, an 'ideal' agent architecture should be able to exploit the potential of integrating several modules/components representing different behaviors/forms of reasoning, with these modules possibly based upon different formalisms. The 'overall agent' should emerge from dynamic, non-deterministic combination of these behaviors that should occur also in consequence of the evolution of the agent's environment. The developers of large and open systems in fact usually handle the complexity and heterogeneity of these systems by applying component-based approaches, where each part/component of the system could be designed and implemented using a specific suitable formalism. Thus, a premise for the integration of different agent programming approaches within an overall framework, requires such framework to be 'modular,' that is, able to accommodate (possibly heterogeneous) modules that are combined through suitable interfaces. In this way, parts of a program can be developed and verified independently and then can be more easily reused. In addition, the system should be maintained in a modular fashion; system's errors and deficiencies should be traced to specific system modules, thus limiting the scope of detailed error searching. A modular programming framework should ideally satisfy the following properties:

- allow abstraction, parametrization, and information hiding;
- ease program development and maintenance of large programs;
- facilitate re-usability;
- have a non-trivial notion of program equivalence to justify replacement of program components;
- preserve the declarativity of logic-based languages, if employed.

In the following subsections we discuss modularity in logic programming, agent-oriented logic programming and ASP, in view of the potential integration of such approaches for building agents.

6.1 Modularity in logic programming

The need for modular extensions to logic programming has been always widely agreed upon. In the modern approaches to software development, complex systems are usually built by composing different modules, also in view of code reusability and extensibility. An important point here which is a key factor in the effectiveness of the composed program is the possibility of reasoning on the composition process. The quest for a structured Logic Programming has motivated a large number of research, which evolved in the area of modular logic programming, for example, Miller (1986), O'Keefe (1985), Mancarella and Pedreschi (1988), Gaifman and Shapiro (1989), Giordano and Martelli (1994), Bugliesi *et al.* (1994). The reader may refer to Bugliesi *et al.* (1994) for a comprehensive survey.

The research conducted in the last decades on logic programming with modules can be divided into two directions (according to Bugliesi *et al.*, 1994). The first one is programming-in-the-large where modules are combined by means of compositional operators for building programs as combinations of separate and independent components. The second direction is programming-in-the-small in which logic programming is enriched with new logical connectives.

In the programming-in-the-large approaches, for example, O'Keefe (1985), Mancarella and Pedreschi (1988), Gaifman and Shapiro (1989), the adopted method is to provide algebraic operators for programs composition such as union, deletion, overriding union, and closure, avoiding the need of an extension of the underlying language of Horn clauses. This provides high flexibility, due to the possibility of obtaining new composition mechanisms by introducing a corresponding operator in the algebra or by combining existing ones. This approach by its nature supports code reusability. Different components of programs can be combined, equivalent components can be replaced using a suitable equivalence relation. Finally, one can obtain encapsulation and information hiding by introducing suitable interfaces between components (cf., e.g. Mancarella & Pedreschi, 1988; Gaifman & Shapiro, 1989; Brogi *et al.*, 1994). The other direction is programming-in-the-small, which uses logical connectives of an extended form of Horn clause language to model the composition of modules (cf, e.g. Miller, 1986; Giordano & Martelli, 1994).

The aim of many researches has been to find a general model-theoretic approach that combines programming in-the-large and in-the-small in a satisfactory way. Many approaches resort to a well-known concept in formal logic, which appears to be a valuable tool for modular logic programming: 'generalized quantifiers'. Generalized quantifiers (Mostowski, 1957; Lindström, 1966) have been conceived as devices for expressing higher order properties which are not first-order definable. Enriching a first order language by such quantifiers allows to increase its expressive capabilities. The extension of first-order logic by generalized quantifiers was studied in depth in the seminal work by Lindström (1966). A logic program can be seen as a generalized quantifier; for modularly nested logic programs, a logic programs P1 can refer to a module P2 by using P2 as a generalized quantifier. A clean and simple semantics for modular logic programs can be obtained by using this approach, which enjoys the following advantages: it extends the classical semantics of logic programming in a conservative way, complies with various extensions of basic Horn-clause semantics, and facilitates the use of modules employing different semantics; it allows the use of the same module in a program multiple times (re-usability); also, it is highly expressive and it supports information hiding.

Eiter *et al.* (1997) presents a general model-theoretic approach to modular logic programming which combines programming in-the-large and in-the-small in a satisfactory way. The authors use generalized quantifiers instead of inventing completely new constructs, showing how generalized quantifiers can be incorporated into logic programs. They observe observed that, since a logic program can be seen as a generalized quantifier, a semantics for modular logic programs follows immediately.

MAD is a Modular Action Description language introduced in Lifschitz and Ren(2006); it is a descendant of the C+ action language described in Giunchiglia *et al.* (2004). MAD extends C+ by adding the capability of splitting action descriptions into modules. In MAD, an action description consists of several modules; each module describes a set of interrelated fluents and actions. 'Import statements' allow the user to provide

references to other modules and thus characterize new fluents and actions by relating them to others, introduced earlier. This capability is a key for designing a repository of background knowledge involving actions, because descriptions of specific action domains will need to 'import' parts of the repository.

6.2 Modular agent-oriented programming languages

There is still big gap between industrial agent programming languages and those developed in academia. To fill this gap or at least to provide a bridge between research-level and industrial-level approaches, agent-oriented languages should provide transparent interfaces for easy integration with external code and legacy software. In addition, the underlying architecture of such programming languages has to be flexible enough to support various approaches to knowledge representation and agent reasoning models.

3APL (Hindriks *et al.*, 1999; Dastani *et al.*, 2005) is the most advanced rule-based language w.r.t. agent programs modularity. Later works by Dastani (cf. Dastani *et al.*, 2004; van Riemsdijk *et al.*, 2006) introduce a semantically oriented modularity to 3APL. In Dastani *et al.* (2004) the authors emphasized the need for defining what it means that the agent 'takes up' a role and 'enacts' it, because agents in open systems like, for example, e-commerce applications usually take up roles temporarily. So, they introduced a notion of role, grouping together beliefs, goals, plans, and reasoning rules of a BDI agent. This role-based approach in the context of AOP facilitates reusability. In van Riemsdijk *et al.* (2006), the authors introduce a concept of goal-oriented modularity for 3APL. It is based on decomposing a set of practical reasoning rules of 3APL agent into modules, according to goals they are supposed to achieve. A rule can call a module to achieve a subgoal in the context of a plan. When the subgoal is achieved, control returns back to the context from which the module was called. This is similar to the stack of routine calls in procedural languages. Implementation of these 3APL extensions required modification to 3APL semantics and to the language interpreter. Both role- and goal- oriented approaches to modularize an agent program are based on particularities of the internal BDI architecture of the agent (the 'mental state'). 3APL was thus modified and extended into 2APL (Dastani, 2008), which adds new programming constructs that allow the programmer to control and determine the activation and use of modules. These constructs and other modifications further support modularity in 2APL.

AgentSpeak(L)/Jason (cf. Rao, 1996; d'Inverno and Luck, 1998) and GOAL (Hindriks *et al.*, 2000) do not facilitate modularity in the sense of decomposition of an agent program to conceptually encapsulated components. It is worth mentioning that GOAL was later extended with the notion of modularity by Hindriks (2008).

MINERVA (cf. Leite *et al.*, 2001; Leite, 2003) and DALI (cf. Costantini & Tocchio, 2002; Costantini & Tocchio, 2004) do not support source code modularity. These two languages exploit the strengths of declarative logic programming and the semantics of both of them is closely connected to logic program updates. The IMPACT (cf. Dix *et al.*, 2000; Rogers *et al.*, 2000) programming framework introduces a vertical modularity to agent programming, where the programmer encodes how an agent system should move from one mental state to another by means of 'updates', which are independent of the internal structure of the agent's mental states. The programmer is free to choose any knowledge representation technique for developing the agent's mental state.

Novák and Dix (2008) presents a method for extending the syntax of an existing specialized AOP languages to meet the requirements of modern programming. A 'core' language is extended by means of higher-level syntactic constructs, thus improving the support for source code modularity and readability. Their approach follows the tiered approach to programming language design (cf. Meyer, 1990). The authors introduce what they call a 'mental state transformer' (mst) based on a functional view of an agent program. Then, they construct 'named mental state transformers' on top of previously-introduced plain mst. Named mst provide a powerful means for agent-program decomposition and modularization. The tiered structure helps to maintain the simplicity and clarity of the programming language semantics in addition to facilitating easier extension of the language. mst constructs can be used to break the agent program into several functionally encapsulated subunits (the higher level ones can 'call' the lower level ones, which facilitates the agent program structuring into several conceptually separated layers). This method is particularly tailored to programming languages oriented to agents with mental states.

6.3 Modular architectures

A modular, flexible and practical agent architecture is one that does not impose specific techniques on design and implementation of an application, but rather allows the programmer to freely choose among the techniques at hand. Several frameworks were proposed driven by the need for such architecture. Here we try to give an insight into some of them.

Fisher (1998) considers an abstract representation of agent architectures which is very general. The author describes the basic organization of an agent architecture by using group structuring, in which the components of an agent can be considered as sub-agents, and the internal organization can be characterized via appropriate patterns of interaction and structuring between these sub-agents. Particularly, the author shows how the layered architectures consisting of various modeling layers (reactive, deliberative, etc.) might be represented by grouping these sub-agents together. Grouping provides a natural modeling technique which is clear and easy to understand, thus supporting both the system designer and the user.

In Kakas *et al*. (2004) and Bracciali *et al*. (2005), the KGP agent model is presented as a declarative model based on computational logic (particularly, abductive logic programming and logic programming with priorities). The KGP model adopts a hierarchical architecture with a highly modular structure that encompasses various reasoning and sensing capabilities which allow an agent to be designed for acting in an open and dynamic environment. The modular separation of concerns leads to flexibility in developing the various components of an agent independently of each other. The control of agents' operational behavior is provided through a context-sensitive cycle-theory component, which breaks the conventional 'one-size-fits-all' control of operation. This facilitates the design of heterogeneous agents. The 'core' of the KGP model is the knowledge base of the agent, which is accessed by a modular collection of capabilities that enable the agent to plan or react, decide new goals, reason temporally and sense the environment in order to check whether goals or (for cautious agents) the preconditions of actions in plans are satisfied.

DALI (cf. Costantini & Tocchio, 2002; Costantini & Tocchio, 2004) is an AOP language and framework which, though not supporting code modularity, is modular by nature. In fact, a DALI agent 'emerges' as the synthesis of a collection of capabilities interconnected among them, that can even be exchanged among agents in a MAS (cf. Costantini & Tocchio, 2005, 2008). Also, DALI allows for sub-agents generation and DALI agents may employ ASP modules. Specifically, Costantini (2011) proposed a framework for integrating ASP modules into virtually any agent architecture so as to allow for complex reactivity, hypothetical reasoning based upon possibility and necessity, planning, etc. From the implementation point of view, the integration of such modules into logic-based architectures is straightforward. In fact, this approach has been implemented within the DALI interpreter. Costantini *et al*. (2015) exploits the extended DALI framework to implement affordable and flexible planning capabilities, with the possibility to select plans according to a suite of possible preferences. The effectiveness of this solution was shown by means of a case-study where DALI agents (organized in a MAS) cooperate in order to explore an unknown territory. The DALI programming environment at current stage of development (De Gasperis *et al*., 2014) offers a multi-platform folder environment, built upon the DALI interpreter, shells scripts, Python scripts to integrate external applications, a JSON/HTML5/jQuery Web interface integrated with DALI applications, with a Python/Twisted/Flask Web server capable to interact with a DALI MAS at the back-end. Thus, DALI is fully compatible with legacy/industrial applications. In addition, a cloud DALI implementation, reported in Costantini, De Gasperis and Nazzicone (2017), Costantini, De Gasperis, Pitoni and Salutari (2017), has recently been developed in view of distributed modular applications, for instance in the Cognitive Robotics and eHealth fields (cf. Aielli *et al*., 2016, for a recently proposed DALI-related project in this field).

Novák and Dix (2008) and Novák (2008) propose a 'modular BDI agent programming architecture,' which is component-based, and independent of the internal structure and underlying formalism of its components. Components of such a BDI system are connected by 'interaction rules.' Thus, a clear distinction is made between knowledge representation in the system's components, and overall system's dynamics. The architecture is equipped with transparent interfaces to existing 'mainstream' programming languages so as to make integration with external code and legacy software easily feasible. In related work,

Novák (2008) introduces the theoretical foundations of an agent programming framework called 'Behavioral State Machines' (BSM). The proposed framework draws a strict and clear distinction between the knowledge representation layer and the behavioral layer of an agent program. It supports a high level of modularity w.r.t. employed KR technologies, thus facilitating the implementation of hybrid agents (i.e. agents mixing both reactive as well as sequential behavior), and provides a clear and concise semantics based on the well-studied computational model of Gurevich's Abstract State Machines (Börger and Stärk (2003)). The Knowledge Representation layer is kept abstract and open, thus giving the possibility of integrating different KR modules (possibly realized in different KR technologies) into the agent's knowledge base. The main focus of the BSM computational model is in fact the agent's behaviors. Basically, in this framework a KR module has to provide a language of query and update formulas and two sets of interfaces: entailment operators for querying the knowledge base and update operators to modify it. To demonstrate the flexibility of the architecture, the aforementioned paper introduces a practical example involving three different KR technologies, namely ASP, Prolog and Java. It formally states the compatibility of the three modules. BSM do not natively feature any of the usual characteristics of rational agents Wooldridge (2000), such as goals, beliefs, intentions, commitments and alike. The BSM programming framework only guides a programmer to the design of a practical agent system while leaving a high degree of freedom. In order to practically test the BSM approach to programming agent systems, the programming language 'Jazzyk' has been designed and implemented (see Novák, 2009).

6.4 Modularity in answer set programming

Research on modularity in ASP is still at early stages. There are several approaches to modularization of ASP programs, but in fact only few of them provide a flexible, easy-to-use modular architecture with a clearly defined interface for module interaction. Below we try to give an insight into some of them.

Eiter *et al*. (1997) approaches modularity in ASP in the programming-in-the-small sense. In this approach, program modules are viewed as generalized quantifiers (Lindström, 1966). The programming-in-the-small approach of Ianni *et al*. (2004) addresses modularity in ASP based on templates for defining subprograms. Tari *et al*. (2005) developed a declarative language for modular ASP; their language allows to declaratively state how one ASP module can import processed answer sets from another ASP module (by means of introducing 'import rules') where no cycles are admitted among modules.

Programming-in-the-large approaches to modularity in ASP are mostly based on Lifschitz and Turner's 'splitting set theorem' (cf. Lifschitz & Turner, 1994) or on variants of it. The basic idea is that a program can in general be divided into two parts: a 'bottom' part and a 'top' part, such that the former does not refer to predicates defined in the latter. Computation of the answer sets of a program can be simplified when the program is split into such parts. Faber *et al*. (2007) applies the magic set method in the evaluation of Datalog programs with negation, that is, normal logic programs without function symbols. The notion of modules and independent sets are introduced to guarantee 'query equivalence' for consistent programs. The modularity results provide a better understanding of Datalog's structural properties. The authors show by experiments that their techniques allow to solve very advanced data-integration tasks.

Janhunen *et al*. (2009) considers the stable-model semantics of disjunctive logic programs (DLPs), and define the notion of a DLP-function where a well-defined input/output interface is provided, and a suitable compositional semantics for modules is introduced. Oikarinen (2008) establishes a simple and intuitive notion of a logic program module that interacts through an input/output interface. This is achieved by accommodating modules as proposed by Gaifman and Shapiro (1989) to the context of ASP. Full compatibility of the module system with the stable model semantics is achieved by allowing positive recursion to occur inside modules only.

Dao-Tran *et al*. (2009) focuses on modular non-monotonic logic programs (MLP) under the answer set semantics, whose modules may have contextually dependent input provided by other modules. Moreover, (mutually) recursive module calls are allowed. The authors define a model-theoretic semantics for this extended setting, showing that many desired properties of ordinary logic programming generalize to their modular ASP. They characterize the computational complexity of modular ASP programs. Then,

they investigate the relationship of modular programs to DLPs with well-defined input/output interface (DLP-functions), and show that they can be embedded into MLPs.

Balduccini (2007b) provides modules specification with information hiding, where modules exchange information with a global state. Baral *et al.* (2006) defines modules in terms of macros that can be called from a program. This with the aim to facilitate the creation and use of a repository of modules that can be used by knowledge engineers without having to re-implement basic knowledge representation concepts from scratch.

In Faber and Woltran (2009) a technique is proposed that allows an answer set program to access the brave or cautious consequences of another answer set program. Lierler and Truszczyński (2013) propose 'modular logic programs' as a modular version of ASP. This work consider programs as modules and define modular programs as sets of modules. The authors introduce 'input answer sets,' the key semantic object for facilitating the communication between modules.

7 Modular distributed architectures

In many application fields, agents and MAS are situated in complex, open, and dynamic computational environments which include heterogeneous software components, physical devices and sensors including wearable devices, third part services, data centers, expert systems and other knowledge sources available on the 'Internet of Everything.' The availability of such components may evolve in time, as new knowledge can be discovered, and components may join or leave the environment, or become momentarily unreachable. Such environments can actually constitute CPS (see Khaitan & McCalley, 2015, for a survey), and they may include physical components that interact with or are integrated into the computational 'ecosystem.' Architectures suitable for modeling such scenarios have been recently proposed, and prototype implementations are or will be soon available.

7.1 Multi-context systems

Multi-context systems (MCS), introduced by Brewka *et al.* (2011a, 2011b, 2014), model the information flow among multiple possibly heterogeneous data sources. The device for doing so is constituted by 'bridge rules,' which are similar to Prolog or, more precisely, Datalog rules, but allow for knowledge acquisition from external sources; in fact, in each element of their 'body' the 'context,' that is, the source, from which information is to be obtained is explicitly indicated. In Managed MCS (mMCS) the conclusion of a bridge rule is not simply added to the 'destination' context's knowledge base; rather, it is subjected to an elaboration via specific operators; the new knowledge can thus be processed (for instance with the aim of making it compatible in terms of format and terminology) and can lead to any (even non-monotonic) re-elaboration of the knowledge base. In order to account for heterogeneity of sources each context is supposed to be based on its own *logic*. Semantics of mMCS is in terms of *equilibria*, where an equilibrium is a global data state composed of partial data states, one for each context, encompassing however inter-context communication determined by bridge rules, and where each element is 'acceptable' w.r.t. the corresponding context's semantics. An equilibrium is thus 'stable' w.r.t. bridge-rules. In view of practical applications it can be noticed that, being logic-based, any context in an mMCS can be a logical agent, to which MCSs have in fact already been explicitly extended (cf. Gonçalves *et al.*, 2014; Costantini. 2015*b*; Costantini & De Gasperis, 2015).

7.2 Agent computational environments

Costantini (2015a) proposes the general software engineering approach of transforming an agent into an agent computational environment (ACE) composed of: (1) the 'main' agent program, or 'basic agent'; (2) a number of reasoning modules available to the main agent program; (3) a number of data sources ('contexts') that the agent is able to access; some contexts will be internal, or local; some will be external, where the set of accessible contexts may vary over time. There is no assumption about how the various components are defined, except that they are based upon Computational Logic. As an interesting

development, an ACE component can in turn be an ACE, thus leading to K-ACE, which is a multi-level modular architecture suitable for highly distributed applications.

Costantini (2015a) proposes a full formalization with a semantics, which draws inspiration from MCSs' equilibrium semantics, and discusses an application to Complex Event Processing. Costantini and Formisano (2016) enhances the approach by introducing the evolution of the system in time under components' update, and by making bridge-rules application tailored to agents on the line of Costantini (2015*b*), Costantini and De Gasperis (2015): in fact, bridge rules are proactively triggered upon specific conditions and the obtained knowledge is reactively elaborated via a *management function* which generalizes the analogous MCS's concept. Also, bridge rules are generalized to 'bridge-rule patterns' where the involved contexts are left unspecified: such patterns can be thus dynamically specialized and the resulting bridge rules can be activated so as to put them into action. Capabilities of Enhanced ACE are demonstrated on a case study concerning quantitative reasoning in agents; a situation is outlined where data might be potentially accessible to an agent so as to build a plan to reach some objective; however, in the case study the agent is constrained by the available budget since access to knowledge bases is subject to fees where one or more stages of the devised plan also imply to pay. The involved quantitative reasoning module is defined in RASP (resource-based answer set programming) (cf. Costantini & Formisano, 2010; Costantini *et al*., 2010; Formisano & Petturiti, 2010) plus suitable preference mechanisms (Costantini & Formisano, 2011), where RASP is an extension of ASP which handles resources and preferences on resource usage. RASP has been proven in Costantini and Formisano (2013) to be equivalent to an interesting fragment of Linear Logic.

ACEs share with Modular BDI Architecture and BSM the possibility of accommodating heterogeneous modules. However, ACE go towards distribution, and bridge rules do not define control but rather knowledge interchange among modules. Nonetheless, BDI and BSM might be employed to define any of an ACE's composing modules.

8 Discussion and concluding remarks

A flexible and robust AOP framework should, in our view, neither enforce a specific knowledge representation technique nor commit to one single scheme of behavior. Rather, it should facilitate the integration of heterogeneous knowledge representation techniques and different schemes of behavior in building agents or MAS. The implementation of a software development platform based upon a modular architecture should indeed include support for the development of components in various programming and knowledge representation languages and formalisms, where the underlying infrastructure should serve to 'glue' components together.

The modular BDI and BSM architectures (mentioned earlier) introduce a clear distinction between the knowledge representation layer of an agent system and its dynamics (behavioral layer), with the former composed of a set of heterogeneous KR modules, particularly of logic-based ones (where ASP modules can also be exploited), connected by means of interaction rules. However, related rule-based languages are constrained only to BDI-type agents and this does not facilitate compositional modularity w.r.t. behavioral decomposition. Both AgentSpeak(L) and 3APL in fact enforce a particular software development methodology together with a specific knowledge representation technique. In addition, there is a tight coupling of interactions between the components of an agent system, such as the belief base and goal base. This has a strong influence on the possibility of their integration with other systems.

METATEM and Golog (cf. Mascardi *et al*., 2004, for a review) in their basic form are not easily extensible to handle several heterogeneous knowledge bases, because of a tight coupling of the KR approach (usually Prolog, or an FOL-style language) with the language for encoding agent's behaviors. IMPACT provides insteaf a high degree of freedom w.r.t. heterogeneous KR technologies and their integration. An IMPACT agent consists of a set of packages with a clearly defined interface composed of a set of data types, data access functions and type composition operations. An agent program is, there, a set of declarative rules involving invocations of the underlying components via predefined data access functions.

KGP knowledge bases describe the knowledge of the agent, also concerning its environment. A KGP agent consists of separate modules supporting the different reasoning capabilities (capabilities modules). These capabilities are based upon abductive logic programming but the architecture might in principle accommodate modules defined in different ways. This modularity in the architecture facilitates in fact the integration of heterogeneous modules in one agent program. ASP modules might in principle be exploited in this architecture to implement various capabilities, for instance planning.

Lots of research and experiments have been done on integrating ASP modules in different agent systems. Various kinds of ASP modules have been integrated into DALI agents, and experimented in significant applications. In the architecture proposed in Sridharan (2016), the coarse-resolution symbolic domain representation is translated into an ASP program, which is solved to provide a tentative plan of abstract actions, and to explain unexpected outcomes. BSM have demonstrated by practical examples the potential of the integration of several knowledge representation techniques among which ASP. As a matter of fact, after reviewing a good corpus of papers about agents, and about the use of ASP in agents and MAS, we are able to remark that ASP is usefully exploited mainly in the form of modules for handling knowledge representation and implementing reasoning capabilities in tasks where ASP proved to be very effective. Few approaches suggest a complete ASP-based agent architecture like done, for example, in Gelfond (2004), Balduccini *et al.* (2014) (see also Gelfond & Kahl, 2014, and the many references therein).

We believe that modularization is the key for flexibility, and naturally for efficiency and scalability. Modularizing an agent and/or transforming it into an ACE is in our opinion the best way to design a robust agent through the integration of heterogeneous modules representing different behaviors or forms of reasoning in one agent system. A flexible agent system or, more generally, an MCS or an ACE, should be able to integrate modules represented by using imperative approaches; this can in fact be practically useful for some functions, for example, for interacting with the environment, and for the user interfaces. Such an agent system might also integrate non-symbolic reasoning modules as done in SOAR, a very well-known 'cognitive architecture' (presented in Laird *et al.*, 1987; Laird, 2008, 2012); a list of references can be found at the SOAR Website, URL http://soar.eecs.umich.edu/) which integrates connectionistic and software components. Declarative approaches will be profitably exploited in domains such as knowledge representation and for 'executable specification' of different behaviors/forms-of-reasoning where declarative programming and ASP have since long proved to bring many advantages, among which clear and robust semantics, high-level concepts like beliefs or goals, flexibility, ease of maintenance, and verifiability[2]. This modular way of representing agents could exploit, where deemed convenient, the advantages of existing well-established imperative approaches like JACK and JADEX, which have object-oriented syntax and large range of available libraries, and the flexibility and generality of architectures such a SOAR for representing, for example, long-term and short-term memory (in Gero & Peng (2009) the reader can find a nice summary with many references of cognitive studies on memory and their applications to agents).

So, the resulting agent systems will be able to explain at best the advantages of all approaches so as to avoid the 'brittleness' that may allegedly result, according to SOAR-Research-Group (2010), from over-commitment to logic-based approaches. The use of agent technologies in industry is growing since the 1990s. The rate of this growth can be further increased by concentrating more on the issues of standards and on adoption of software development tools and methodologies to facilitate and empower the use of integrated multi-paradigm solutions, and the development of highly distributed heterogeneous systems.

Acknowledgement

Abeer Dyoub acknowledges support by the Project 'Machine Ethics for Cybersecurity' of the Department of Information Engineering, Computer Science and Mathematics of the University of L'Aquila. The authors wish to thank the anonymous reviewers for the useful comments and advice.

[2] For lack of space we cannot discuss the—very important—issue of verification of agents and MAS; the reader may refer to Belardinelli *et al.* (2014), Kouvaros and Lomuscio (2016), and to the references therein.

References

Aielli, F., Ancona, D., Caianiello, P., Costantini, S., De Gasperis, G., Marco, A. D., Ferrando, A. & Mascardi, V. 2016. FRIENDLY & KIND with your health: human-friendly knowledge-intensive dynamic systems for the e-health domain. In *'PAAMS (Workshops)', Communications in Computer and Information Science*, 616, 15–26. Springer.

Akbari, O. Z. 2010. A survey of agent-oriented software engineering paradigm: towards its industrial acceptance. *Journal of Computer Engineering Research* **1**(2), 14–28.

Alviano, M., Faber, W., Greco, G. & Leone, N. 2012. Magic sets for disjunctive datalog programs. *Artificial Intelligence* **187**, 156–192.

Ambros-Ingerson, J. A. & Steel, S. 1988. Integrating planning, execution and monitoring. In *Proceedings of the 7th National Conference on Artificial Intelligence*, 83–88, August 21–26.

Amendola, G., Greco, G., Leone, N. & Veltri, P. 2016. Modeling and reasoning about NTU games via answer set programming. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI 2016, 38–45. IJCAI/AAAI Press, July 9–15.

Anderson, J. R. & Lebiere, C. 1998. The Atomic Components of Thought. Lawrence Erlbaum Associates.

Answer Set Programming Solvers 2016. Available at `http://assat.cs.ust.hk`; `http://www.cs.utexas.edu/users/tag/ccalc/`; `https://potassco.org/clasp/`; `http://www.cs.utexas.edu/users/tag/cmodels/`; `http://www.cs.uky.edu/ai/`; `http://www.dlvsystem.com/dlv/`; `http://www.tcs.hut.fi/Software/smodels/`

Apt, K. R. & Bol, R. N. 1994. Logic programming and negation: a survey. *Journal of Logic Programming* **19/20**, 9–72.

Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A. & Thorstensen, E. 2011. Optimization methods for the partner units problem. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011. Proceedings*, T. Achterberg and J. C. Beck (eds), 'Lecture Notes in Computer Science **6697**, 4–19. Springer.

Baldoni, M., Baroglio, C., Mascardi, V., Omicini, A. & Torroni, P. 2010. Agents, multi-agent systems and declarative programming: what, when, where, why, who, how? In *A 25-Year Perspective on Logic Programming*, Dovier, A. & Pontelli, E. (eds). Springer-Verlag, 204–230.

Balduccini, M. 2007*a*. Learning action descriptions with a-prolog: action language c. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, AAAI, Technical Report SS-07-05, 13–18.

Balduccini, M. 2007*b*. Modules and signature declarations for a-prolog: progress report. In Workshop on Software Engineering for Answer Set Programming (SEA'07), 41–55.

Balduccini, M. & Gelfond, M. 2003. Diagnostic reasoning with a-prolog. *Theory and Practice of Logic Programming* **3**(4), 425–461.

Balduccini, M. & Gelfond, M. 2008. The AAA architecture: an overview. In *Architectures for Intelligent Theory-Based Agents, Papers from the 2008 AAAI Spring Symposium*, Technical Report SS-08-02, 1–6. AAAI, March 26–28.

Balduccini, M., Gelfond, M. & Nogueira, M. 2006. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence* **47**(1–2), 183–219.

Balduccini, M., Regli, W. C. & Nguyen, D. N. 2014. An ASP-based architecture for autonomous UAVs in dynamic environments: Progress report, *CoRR* **abs/1405.1124**.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Baral, C., Dzifcak, J. & Takahashi, H. 2006. Macros, macro calls and use of ensembles in modular answer set programming. In *Logic Programming*, 376–390. Springer.

Baral, C., Gelfond, G., Son, T. C. & Pontelli, E. 2010. Using answer set programming to model multi-agent scenarios involving agents' knowledge about other's knowledge. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1*, 259–266. International Foundation for Autonomous Agents and Multiagent Systems.

Baral, C. & Gelfond, M. 2000. Reasoning agents in dynamic domains. In *Logic-Based Artificial Intelligence*, van der Hoek, W., Kaminka, G. A., Lespérance, Y., Luck, M. & Sen, S. (eds). Springer, 257–279.

Bauer, B., Müller, J. P. & Odell, J. 2001. Agent UML: a formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering* **11**(3), 207–230.

Bauters, K. 2011*a*. Modeling coalition formation using multi-focused answer sets. In *Proceedings of ESSLLI*, **11**, 25–33.

Bauters, K. 2011*b*. Modeling negotiation using multi-focused answer sets. In *2011 European Summer School in Logic, Language and Information (ESSLLI 2011): Student session*, 25–33.

Bauters, K., Janssen, J., Schockaert, S., De Cock, M. & Vermeir, D. 2010. Communicating answer set programs. In *26th International Conference of Logic Programming (ICLP 2010)*, **7**, 34–43. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Bauters, K., Schockaert, S., Janssen, J., Vermeir, D. & De Cock, M. 2013. Expressiveness of communication in answer set programming. *Theory and Practice of Logic Programming* **13**(3), 361–394.

Belardinelli, F., Lomuscio, A. & Patrizi, F. 2014. Verification of agent-based artifact systems. *Journal of Artificial Intelligence Research* **51**, 333–376.

Blount, J., Gelfond, M. & Balduccini, M. 2015. A theory of intentions for intelligent agents - (extended abstract). In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR 2015*, Lecture Notes in Computer Science **9345**, 134–142. Springer.

Borchert, P., Anger, C., Schaub, T. & Truszczynski, M. 2004. Towards systematic benchmarking in answer set programming: the dagstuhl initiative. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR 2004*, Lecture Notes in Computer Science **2923**, 3–7. Springer.

Bordini, R. H., Braubach, L., Dastani, M., Fallah-Seghrouchni, A. E., Gómez-Sanz, J. J., Leite, J., O'Hare, G. M. P., Pokahr, A. & Ricci, A. 2006. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)* **30**(1), 33–44.

Bordini, R. H. & Hübner, J. F. 2006. BDI agent programming in AgentSpeak using *Jason* (tutorial paper). In *Computational Logic in Multi-Agent Systems, 6th International Workshop, CLIMA VI, Revised Selected and Invited Papers*, F. Toni and P. Torroni (eds), LNCS **3900**. Springer, 143–164.

Bordini, R. H., Hübner, J. F. & Wooldridge, M. 2007. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons. Wiley Series in Agent Technology.

Börger, E. & Stärk, R. F. 2003. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer.

Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Mancarella, P., Sadri, F., Stathis, K., Terreni, G. & Toni, F. 2005. The KGP model of agency: computational model and prototype implementation. In *Global Computing: IST/FET Intl. Workshop, Revised Selected Papers*, LNAI **3267**, Springer-Verlag, 340–367.

Bratman, M. E. 1999. *Intention, Plans, and Practical Reason*. Cambridge University Press.

Bratman, M. E., Israel, D. J. & Pollack, M. E. 1988. Plans and resource-bounded practical reasoning. *Computational Intelligence* **4**(3), 349–355.

Brewka, G., Eiter, T. & Fink, M. 2011a. Nonmonotonic multi-context systems: a flexible approach for integrating heterogeneous knowledge sources. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, M. Balduccini and T. C. Son (eds), Lecture Notes in Computer Science **6565**. Springer, 233–258.

Brewka, G., Eiter, T., Fink, M. & Weinzierl, A. 2011b. Managed multi-context systems. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, T. Walsh (ed.), 786–791. IJCAI/AAAI.

Brewka, G., Ellmauthaler, S. & Pührer, J. 2014. Multi-context systems for reactive reasoning in dynamic environments. In *ECAI 2014, Proceedings of the 21st European Conference on Artificial Intelligence*, T. Schaub (ed.), 159–164. IJCAI/AAAI.

Brogi, A., Mancarella, P., Pedreschi, D. & Turini, F. 1994. Modular logic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **16**(4), 1361–1398.

Brooks, D. R., Erdem, E., Erdogan, S. T., Minett, J. W. & Ringe, D. 2007. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning* **39**(4), 471–511.

Brooks, R. A. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* **2** (1), 14–23.

Brustoloni, J. C. 1991. *Autonomous Agents: Characterization and Requirements*, Technical report, Carnegie Mellon University.

Buccafurri, F. & Caminiti, G. 2005. A social semantics for multi-agent systems. In *Logic Programming and Nonmonotonic Reasoning*, 317–329. Springer.

Buccafurri, F. & Gottlob, G. 2002. Multiagent compromises, joint fixpoints, and stable models. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, A. C. Kakas and F. Sadri (eds), Lecture Notes in Computer Science **2407**, 561–585. Springer.

Bugliesi, M., Lamma, E. & Mello, P. 1994. Modularity in logic programming. *The Journal of Logic Programming, Elsevier* **19**, 443–502.

Castellucci, A., Ianni, G., Vasile, D. & Costa, S. 2001. Searching and surfing the web using a semi-adaptive meta-engine. In *2001 International Symposium on Information Technology (ITCC 2001)*, 416–420. IEEE Computer Society, April 2–4.

Cliffe, O., De Vos, M. & Padget, J. 2006. Answer set programming for representing and reasoning about virtual institutions. In *Computational Logic in Multi-Agent Systems*, 60–79. Springer.

Coen, M. H. 1994. Sodabot: a software agent environment and construction system. In *Proceedings of the 12th National Conference on Artificial Intelligence, Volume 2*, 1433. AAAI Press/MIT Press.

Cohen, P. R., Greenberg, M. L., Hart, D. M. & Howe, A. E. 1989. Trial by fire: understanding the design requirements for agents in complex environments. *AI Magazine* **10**(3), 32–48.

Cortés, U., Tolchinsky, P., Nieves, J., López-Navidad, A. & Caballero, F. 2005. Arguing the discard of organs for tranplantation in CARREL In CATAI 2005, 93–105.

Costantini, S. 1995. Contributions to the stable model semantics of logic programs with negation. *Theoretical Computer Science* **149**(2), 231–255.

Costantini, S. 2006. On the existence of stable models of non-stratified logic programs. *Theory and Practice of Logic Programming* **6**(1–2), 169–212.

Costantini, S. 2011. Answer set modules for logical agents. In *Datalog Reloaded - First International Workshop, Datalog 2010. Revised Selected Papers*, O. de Moor, G. Gottlob, T. Furche and A. J. Sellers (eds), Lecture Notes in Computer Science **6702**, 37–58. Springer.

Costantini, S. 2015*a*. ACE: a flexible environment for complex event processing in logical agents. In *Engineering Multi-Agent Systems, Third International Workshop, EMAS 2015, Revised Selected Papers*, M. Baldoni, L. Baresi and M. Dastani (eds), Lecture Notes in Computer Science **9318**, 70–91. Springer.

Costantini, S. 2015*b*. Knowledge acquisition via non-monotonic reasoning in distributed heterogeneous environments. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR 2015*, M. Truszczyński, G. Ianni and F. Calimeri, (eds), Lecture Notes in Computer Science **9345**, 228–241. Springer.

Costantini, S., D'Antona, O. M. & Provetti, A. 2002. On the equivalence and range of applicability of graph-based representations of logic programs. *Information Processing Letters* **84**(5), 241–249.

Costantini, S., De Gasperis, G. & Nazzicone, G. 2017. DALI for cognitive robotics: principles and prototype implementation. In *Practical Aspects of Declarative Languages - 19th International Symposium, Proceedings*, Y. Lierler and W. Taha (eds), Lecture Notes in Computer Science **10137**, 152–162. Springer.

Costantini, S. & De Gasperis, G. 2015. Exchanging data and ontological definitions in multi-agent-contexts systems. In *Challenge + DC@RuleML*, CEUR Workshop Proceedings **1417**. CEUR-WS.org.

Costantini, S., De Gasperis, G. & Nazzicone, G. 2015. Exploration of unknown territory via DALI agents and ASP modules. In *Distributed Computing and Artificial Intelligence, 12th International Conference*, DCAI 2015, S. Omatu, Q. M. Malluhi, S. Rodríguez-González, G. Bocewicz, E. Bucciarelli, G. Giulioni and F. Iqba (eds), Advances in Intelligent Systems and Computing **373**, 285–292. Springer.

Costantini, S., De Gasperis, G., Pitoni, V. & Salutari, A. 2017. DALI: a multi agent system framework for the web, cognitive robotic and complex event processing. In *Proceedings of the 32nd Italian Conference on Computational Logic*, CEUR Workshop Proceedings **1949**, 286–300. CEUR-WS.org. `http://ceur-ws.org/Vol-1949/CILCpaper05.pdf`

Costantini, S. & Formisano, A. 2010. Answer set programming with resources. *Journal of Logic and Computation* **20** (2), 533–571.

Costantini, S. & Formisano, A. 2011. Weight constraints with preferences in ASP. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, Lecture Notes in Computer Science **6645**. Springer.

Costantini, S. & Formisano, A. 2013. 'RASP and ASP as a fragment of linear logic'. *Journal of Applied Non-Classical Logics* **23**(1–2), 49–74.

Costantini, S. & Formisano, A. 2016. Augmenting agent computational environments with quantitative reasoning modules and customizable bridge rules. In *Autonomous Agents and Multiagent Systems - AAMAS 2016 Workshops, - Visionary Papers, Revised Selected Papers*, Lecture Notes in Computer Science **10003**, 104–121.

Costantini, S., Formisano, A. & Petturiti, D. 2010. Extending and implementing RASP. *Fundamenta Informaticae.* **105**(1–2), 1–33.

Costantini, S. & Tocchio, A. 2002. A logic programming language for multi-agent systems. In *Logics in Artificial Intelligence, Proceedings of the 8th European Conference*, JELIA 2002, LNAI **2424**, 1–13. Springer-Verlag.

Costantini, S. & Tocchio, A. 2004. The DALI logic programming agent-oriented language. In *Logics in Artificial Intelligence, Proceedings of the 9th European Conference*, Jelia 2004, LNAI **3229**, 685–688. Springer-Verlag.

Costantini, S. & Tocchio, A. 2005. Learning by knowledge exchange in logical agents. In *WOA 2005: Dagli Oggetti agli Agenti. 6th AI*IA/TABOO Joint Workshop "From Objects to Agents": Simulation and Formal Analysis of Complex Systems*, F. D. Paoli, E. Merelli and A. Omicini (eds). Pitagora Editrice Bologna, 1–8.

Costantini, S. & Tocchio, A. 2008. DALI: an architecture for intelligent logical agents. In *AAAI Spring Symposium: Emotion, Personality, and Social Behavior*, 13–18. AAAI.

Dantsin, E., Eiter, T., Gottlob, G. & Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys* **33**(3), 374–425.

Dao-Tran, M., Eiter, T., Fink, M. & Krennwallner, T. 2009. Modular nonmonotonic logic programming revisited. In *Logic Programming*, 145–159. Springer.

Dastani, M. 2008. 2APL: a practical agent programming language, *Autonomous Agents and Multi-Agent Systems* **16** (3), 214–248.

Dastani, M. 2015. Programming multi-agent systems. *Knowledge Engineering Review* **30**(4), 394–418.

Dastani, M., van Birna Riemsdijk, M. & Meyer, J.-J. C. 2005. Programming multi-agent systems in 3APL. In *Multi-Agent Programming*, 39–67. Springer.

Dastani, M., Van Riemsdijk, M. B., Hulstijn, J., Dignum, F. & Meyer, J.-J. C. 2004. Enacting and deacting roles in agent programming. In *Agent-oriented software engineering V*, 189–204. Springer.

De Gasperis, G., Costantini, S. & Nazzicone, G. 2014. Dali multi agent systems framework, doi 10.5281/zenodo.11042, DALI GitHub Software Repository. DALI. `http://github.com/AAAI-DISIM-UnivAQ/DALI`

De Vos, M., Cliffe, O., Watson, R., Crick, T., Padget, J. A., Needham, J. & Brain, M. 2005. T-laima: answer set programming for modelling agents with trust. In *EUMAS*, Koninklijke Vlaamse Academie van Belie voor Wetenschappen en Kunsten, 126–136.

De Vos, M. & Vermeir, D. 2004. Extending answer sets for logic programming agents. *Annals of Mathematics and Artificial Intelligence, Springer* **42**(1–3), 103–139.

d'Inverno, M., Hindriks, K. & Luck, M. 2000. A formal architecture for the 3APL agent programming language. In *ZB 2000: Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK, August 29 - September 2, 2000, Proceedings*, 168–187. Springer.

d'Inverno, M. & Luck, M. 1998. Engineering agentspeak (l): a formal computational model. *Journal of Logic and Computation* **8**(3), 233–260.

Dix, J., Eiter, T., Kraus, S., Ozcan, F. & Subrahmanian, V. S. 2000. *Heterogeneous agent systems*. The MIT Press.

Duch, W., Oentaryo, R. J. & Pasquier, M. 2008. Cognitive architectures: Where do we go from here? In *AGI Conference*, **171**, 122–136. IOS Press.

Dung, P. M. 1995. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence* **77**(2), 321–358.

Eiter, T., Gottlob, G. & Veith, H. 1997. Modular logic programming and generalized quantifiers. In *Logic Programming and Nonmonotonic Reasoning*, 289–308. Springer.

Erdem, E., Aker, E. & Patoglu, V. 2012. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics* **5**(4), 275–291.

Erdem, E., Erdem, Y., Erdogan, H. & Öztok, U. 2011. Finding answers and generating explanations for complex biomedical queries. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI 2011, W. Burgard and D. Roth, (eds). AAAI Press, August 7–11.

Erdem, E., Gelfond, M. & Leone, N. 2016. Applications of answer set programming. *AI Magazine* **37**(3), 53–68.

Erdem, E. & Öztok, U. 2015. Generating explanations for biomedical queries. *Theory and Practice of Logic Programming* **15**(1), 35–78.

Erdem, E., Patoglu, V. & Saribatur, Z. G. 2015. Integrating hybrid diagnostic reasoning in plan execution monitoring for cognitive factories with multiple robots. In *IEEE International Conference on Robotics and Automation*, ICRA 2015, 2007–2013. IEEE, May 26–30.

Erdem, E., Patoglu, V., Saribatur, Z. G., Schüller, P. & Uras, T. 2013. 'Finding optimal plans for multiple teams of robots through a mediator: a logic-based approach'. *Theory and Practice of Logic Programming* **13**(4–5), 831–846.

Etzioni, O., Lesh, N. & Segal, R. 1994. *Building Softbots for Unix (Preliminary Report)*. Technical report. AAAI Press.

Faber, W., Greco, G. & Leone, N. 2007. Magic sets and their application to data integration. *Journal of Computer and System Sciences* **73**(4), 584–609.

Faber, W. & Woltran, S. 2009. Manifold answer-set programs for meta-reasoning. In *Logic Programming and Nonmonotonic Reasoning*, 115–128. Springer.

Febbraro, O., Leone, N., Grasso, G. & Ricca, F. 2012. JASP: a framework for integrating answer set programming with java. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference*, KR 2012. AAAI Press, June 10–14.

Febbraro, O., Reale, K. & Ricca, F. 2011. ASPIDE: integrated development environment for answer set programming. In *Logic Programming and Nonmonotonic Reasoning*, 317–330. Springer.

Ferguson, I. A. 1992. Touring machines: autonomous agents with attitudes. *IEEE Computer* **25**(5), 51–55.

Fikes, R. E. & Nilsson, N. J. 1971. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial intelligence* **2**(3–4), 189–208.

Fisher, M. 1994. A survey of concurrent METATEM the language and its applications. In *Temporal Logic* 480–505. Springer.

Fisher, M. 1998. Representing abstract agent architectures. In *Intelligent Agents V: Agents Theories, Architectures, and Languages*, 227–241. Springer.

Fisher, M., Bordini, R. H., Hirsch, B. & Torroni, P. 2007. Computational logics and agents: a road map of current technologies and future trends. *Computational Intelligence Journal* **23**(1), 61–91.

Formisano, A. & Petturiti, D. 2010. RASP and P-RASP: an implementation. `http://www.dmi.unipg.it/formis/raspberry/`

Franklin, S. & Graesser, A. C. 1996. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Intelligent Agents III, Agent Theories, Architectures, and Languages, ECAI '96 Workshop (ATAL), Budapest, Hungary, August 12-13, 1996, Proceedings*, 21–35. Springer.

Friedrich, G., Fugini, M., Mussi, E., Pernici, B. & Tagni, G. 2010. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering* **36**(2), 198–215.

Friedrich, G., Ryabokon, A., Falkner, A. A., Haselböck, A., Schenner, G. & Schreiner, H. 2011. (Re)configuration based on model generation. In *Proceedings Second Workshop on Logics for Component Configuration*, LoCoCo 2011, C. Drescher, I. Lynce and R. Treinen (eds), EPTCS **65**, 26–35, September 12.

Gaifman, H. & Shapiro, E. 1989. Fully abstract compositional semantics for logic programs. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 134–142. ACM.

Gallucci, L. & Ricca, F. 2007. Visual querying and application programming interface for an ASP-based ontology language. *Proceedings of SEA* **7**, 56–70.

Gebser, M., Schaub, T., Thiele, S. & Veber, P. 2011. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming* **11**(2–3), 323–360.

Gelfond, M. 2004. Answer set programming and the design of deliberative agents. In *Logic Programming, Proceedings of the 20th International Conference*, *ICLP 2004*, B. Demoen and V. Lifschitz (eds), Lecture Notes in Computer Science **3132**, 19–26. Springer.

Gelfond, M. 2008. Answer sets. In *Handbook of Knowledge Representation. Chapter 7*, van Harmelen F., Lifschitz V. & Porter B. W. (eds). Foundations of Artificial Intelligence **3**, 285–316. Elsevier.

Gelfond, M. & Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press.

Gelfond, M. & Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen (eds), 1070–1080. MIT Press.

Gelfond, M. & Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* **9**, 365–385.

Gelfond, M. & Lifschitz, V. 1998. Action languages. *Electronic Transactions on Artificial Intelligence* **2**, 193–210.

Georgeff, M., Pell, B., Pollack, M., Tambe, M. & Wooldridge, M. 1998. The belief-desire-intention model of agency. In *Intelligent Agents V: Agents Theories, Architectures, and Languages, 5th International Workshop, ATAL '98, Paris, France, July 4-7, 1998, Proceedings*, 1–10. Springer.

Gero, J. S. & Peng, W. 2009. Understanding behaviors of a constructive memory agent: a Markov chain analysis. *Knowledge-Based Systems* **22**(8), 610–621.

Giordano, L. & Martelli, A. 1994. Structuring logic programs: a modal approach. *The Journal of Logic Programming* **21**(2), 59–94.

Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N. & Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence* **153**(1), 49–104.

Gonçalves, R., Knorr, M. & Leite, J. 2014. Evolving bridge rules in evolving multi-context systems. In *Computational Logic in Multi-Agent Systems - 15th International Workshop, CLIMA XV. Proceedings*, N. Bulling, L. W. N. van der Torre, S. Villata, W. Jamroga and W. W. Vasconcelos (eds), 52–69.

Grasso, G., Leone, N. & Ricca, F. 2013. Answer set programming: language, applications and development tools. In *Web Reasoning and Rule Systems - 7th International Conference, RR 2013, Proceedings*, W. Faber and D. Lembo (eds), Lecture Notes in Computer Science **7994**. Springer.

Greco, G. & Terracina, G. 2013. Frequency-based similarity for parameterized sequences: formal framework, algorithms, and applications. *Information Science* **237**, 176–195.

Havur, G., Ozbilgin, G., Erdem, E. & Patoglu, V. 2014. Geometric rearrangement of multiple movable objects on cluttered surfaces: a hybrid reasoning approach. In *2014 IEEE International Conference on Robotics and Automation*, ICRA 2014, 445–452. IEEE, May 31 to June 7.

Hindriks, K. 2008. Modules as policy-based intentions: modular agent programming in GOAL. In *Proceedings of the 5th International Conference on Programming Multi-agent Systems ProMAS07*, 156–171. Springer. `http://dl.acm.org/citation.cfm?id=1793534.1793546`

Hindriks, K. V., De Boer, F. S., Van der Hoek, W. & Meyer, J.-J. C. 1999. Agent programming in 3APL, *Autonomous Agents and Multi-Agent Systems* **2**(4): 357–401.

Hindriks, K. V., De Boer, F. S., Van Der Hoek, W. & Meyer, J.-J. C. 2000. Agent programming with declarative goals. In *Intelligent Agents VII, Agent Theories Architectures and Languages*, 228–243. Springer.

Hübner, J. F., Sichman, J. S. & Boissier, O. 2007. Developing organised multiagent systems using the MOISE. *IJAOSE* **1**(3/4), 370–395.

Ianni, G. 2001. Intelligent anticipated exploration of web sites. *AI Communications* **14**(4), 197–214.

Ianni, G., Calimeri, F., Lio, V. & Galizia, S. 2003. Reasoning about the semantic web using answer set programming. In *Proceedings of the 2003 Joint Conference on Declarative Programming*, AGP-2003, F. Buccafurri (ed.), 324–336.

Ianni, G., Ielpa, G., Pietramala, A., Santoro, M. C. & Calimeri, F. 2004. Enhancing answer set programming with templates. In *10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Proceedings*, 233–239.

JADE website 2016. Available at `http://jade.tilab.com/`

Janhunen, T., Oikarinen, E., Tompits, H. & Woltran, S. 2009. Modularity aspects of disjunctive stable models. *Journal of Artificial Intelligence Research* **35**, 813–857.

Jennings, N. R. 1993. Specification and implementation of a belief-desire-joint-intention architecture for collaborative problem solving. *International Journal of Intelligent and Cooperative Information Systems, World Scientific* **2**(3), 289–318.

Jennings, N. R., Sycara, K. & Wooldridge, M. 1998. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems* **1**(1), 7–38.

Juneidi, S. J. & Vouros, G. A. 2004. Survey and evaluation of agent oriented software engineering. In *IASTED International Conference on Software Engineering, part of the 22nd Multi-Conference on Applied Informatics, 2004*, M. H. Hamza (ed.), 433–440. IASTED/ACTA Press.

Kaelbling, L. P. 1991. A situated-automata approach to the design of embedded agents. *SIGART Bulletin* **2**(4), 85–88.

Kakas, A. C., Mancarella, P., Sadri, F., Stathis, K. & Toni, F. 2004. The KGP model of agency. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, ECAI 2004, R. L. de Mántaras and L. Saitta (eds), 33–37. IOS Press.

Kautz, H. A. & Selman, B. 1992. Planning as satisfiability. In *ECAI*, 359–363.

Keil, F. 1989. *Concepts, Kinds, and Cognitive Development*. The MIT Press.

Khaitan, S. K. & McCalley, J. D. 2015. Design techniques and applications of cyberphysical systems: a survey. *IEEE Systems Journal* **9**(2), 350–365.

Kinny, D., Georgeff, M. P. & Rao, A. S. 1996. A methodology and modelling technique for systems of BDI agents. In *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Proceedings*, W. V. de Velde and J. W. Perram (eds), Lecture Notes in Computer Science **1038**, 56–71. Springer.

Kouvaros, P. & Lomuscio, A. 2016. Parameterised verification for multi-agent systems. *Artificial Intelligence.* **234**, 152–189.

Kowalski, R. & Sadri, F. 1996. Towards a unified agent architecture that combines rationality with reactivity. In *Logic in Databases*, Lecture Notes in Computer Science **1154**, 135–149. Springer.

Laird, J. 2012. *The SOAR Cognitive Architecture*. MIT Press.

Laird, J. E. 2008. Extending the SOAR cognitive architecture. In *Proceedings of the First Artificial General Intelligence Conference*, 224–235.

Laird, J. E., Newell, A. & Rosenbloom, P. S. 1987. Soar: an architecture for general intelligence. *Artificial Intelligence* **33**(1), 1–64.

Langley, P. 2005. An adaptive architecture for physical agents. In *The 2005 IEEE/WIC/ACM International Conference on Web Intelligence, 2005. Proceedings*, 18–25. IEEE.

Langley, P., Laird, J. E. & Rogers, S. 2009. Cognitive architectures: research issues and challenges. *Cognitive Systems Research* **10**(2), 141–160.

Leite, J.-A. 2003. Evolving knowledge bases: specification and semantics, Frontiers in Artificial Intelligence and Applications, **81**. IOS Press.

Leite, J., Alferes, J. J. & Mito, B. 2009. Resource allocation with answer-set programming. In *8th International Joint Conference on Autonomous Agents and Multiagent Systems AAMAS 2009, Proceedings*, C. Sierra and C. Castelfranchi and K. S. Decker and J. Simão Sichman (eds), 649–656. IFAAMAS.

Leite, J. A., Alferes, J. J. & Pereira, L. M. 2001. MINERVA - a dynamic logic programming agent architecture. In *Intelligent Agents VIII, 8th International Workshop, ATAL 2001, Revised Papers*, J. C. Meyer and M. Tambe, (eds), Lecture Notes in Computer Science **2333**, 141–157. Springer.

Leone, N. 2007. Logic programming and nonmonotonic reasoning: from theory to systems and applications. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR 2007*, Lecture Notes in Computer Science **4483**, 1. Springer.

Leone, N. & Ricca, F. 2015. Answer set programming: a tour from the basics to advanced development tools and industrial applications. In *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Tutorial Lectures*, W. Faber and A. Paschke, (eds), Lecture Notes in Computer Science **9203**, 308–326. Springer.

Lierler, Y. & Truszczyński, M. 2013. Modular answer set solving, *Late-Breaking Developments in the Field of Artificial Intelligence*, **WS-13-17**. AAAI Press.

Lifschitz, V. 1999. Action languages, answer sets, and planning. In *The Logic Programming Paradigm*, 357–373. Springer.

Lifschitz, V. & Ren, W. 2006. A modular action description language. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, **6**, 853–859. AAAI Press.

Lifschitz, V. & Turner, H. 1994. Splitting a logic program. In *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming*, P. V. Hentenryck (ed.), 23–37. MIT Press.

Lindström, P. 1966. First order predicate logic with generalized quantifiers. *Theoria, Wiley Online Library* **32**(3), 186–195.

Lloyd, J. W. 1987. *Foundations of Logic Programming*, 2nd Edition. Springer.

Maes, P. 1991. The agent network architecture (ANA). *SIGART Bulletin* **2**(4), 115–120.

Maes, P. 1993. Modeling adaptive autonomous agents. *Artificial Life* **1**(1–2), 135–162.

Mancarella, P. & Pedreschi, D. 1988. An algebra of logic programs. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, 1006–1023. MIT Press.

Manna, M., Scarcello, F. & Leone, N. 2011. On the complexity of regular-grammars with integer attributes. *Journal of Computer and System Sciences* **77**(2), 393–421.

Maratea, M., Pulina, L. & Ricca, F. 2015. Multi-level algorithm selection for ASP. In *Proceedings of the Logic Programming and Nonmonotonic Reasoning - 13th International Conference*, LPNMR 2015, Lecture Notes in Computer Science **9345**, 439–445. Springer.

Marek, V. W. & Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm*, 375–398. Springer.

Mascardi, V., Demergasso, D. & Ancona, D. 2005. Languages for programming BDI-style agents: an overview. In *WOA 2005: Dagli Oggetti agli Agenti. 6th AI*IA/TABOO Joint Workshop "From Objects to Agents": Simulation and Formal Analysis of Complex Systems*, F. D. Paoli, E. Merelli and A. Omicini (eds), 9–15. Pitagora Editrice Bologna.

Mascardi, V., Martelli, M. & Sterling, L. 2004. Logic-based specification languages for intelligent software agents. *Theory and Practice of Logic Programming* **4**(4), 429–494.

Meyer, B. 1990. *Introduction to the Theory of Programming Languages*. Prentice Hall.

Meyer, D. E. & Kieras, D. E. 1997. A computational theory of executive cognitive processes and multiple-task performance: part I. basic mechanisms. *Psychological Review* **104**(1), 3.

Miller, D. 1986. A theory of modules for logic programming. In *SLP*, 106–114. IEEE-CS.

Moore, R. C. 1985. Semantical considerations on nonmonotonic logic. *Artificial Intelligence* **25**(1), 75–94.

Mostowski, A. 1957. On a generalization of quantifiers. *Fundamenta Mathematicae* **44**(1), 12–36.

Müller, J. P. & Pischel, M. 1994. An architecture for dynamically interacting agents. *International Journal of Cooperative Information Systems* **3**(1), 25–46.

Nam, T. H. & Baral, C. 2009. Hypothesizing about signaling networks. *Journal of Applied Logic* **7**(3), 253–274.

Neches, R., Langley, P. & Klahr, D. 1987. *Learning, Development, and Production Systems*. The MIT Press.

Newell, A. 1973. Production systems: models of control structures. In *Visual Information Processing*, 463–526. Elsevier.

Newell, A. 1990. *Unified Theories of Cognition*. Harvard University Press.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* **25**(3–4), 241–273.

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R. & Barry, M. 2001. An A prolog decision support system for the space shuttle. In *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop*, A. Provetti and T. C. Son (eds).

Novák, P. 2008. An *Open Agent Architecture: Fundamentals*. Technical Report No. IfI-07-10, Department of Informatics, Clausthal University of Technology (November 2007).

Novák, P. 2009. Jazzyk: a programming language for hybrid agents with heterogeneous knowledge representations. In *Programming Multi-Agent Systems, 6th International Workshop, ProMAS 2008. Revised Invited and Selected Papers*, K. V. Hindriks, A. Pokahr and S. Sardiña (eds), Lecture Notes in Computer Science **5442**, 72–87. Springer.

Novák, P. & Dix, J. 2008. Adding structure to agent programming languages. In *Programming Multi-Agent Systems, 5th International Workshop, ProMAS 2007, Honolulu, HI, USA, May 15, 2007, Revised and Invited Papers*, M. Dastani, A. E. Fallah-Seghrouchni, A. Ricci and M. Winikoff (eds), Lecture Notes in Computer Science **4908**, 140–155. Springer.

Oikarinen, E. 2008. *Modularity in Answer Set Programs*. PhD thesis, Helsinki University of Technology.

O'Keefe, R. A. 1985. Towards an algebra for constructing logic programs. In *Proceedings of the 1985 Symposium on Logic Programming*, 152–160. IEEE-CS.

Omicini, A. 2001. SODA: societies and infrastructures in the analysis and design of agent-based systems. In *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000, Revised Papers*, P. Ciancarini and M. Wooldridge (eds), Lecture Notes in Computer Science **1957**, 185–193. Springer.

Osorio, M., Zepeda, C., Nieves, J. C. & Cortés, U. 2005. Inferring acceptable arguments with answer set programming. In *Sixth Mexican International Conference on Computer Science (ENC) 2005)*, I198–205. EEE Computer Society.

Petrie, C. J. 1996. Agent-based engineering, the web, and intelligence. *IEEE Expert* **11**(6), 24–29.

Pokahr, A., Braubach, L. & Lamersdorf, W. 2005. Jadex: a BDI reasoning engine. In *Multi-Agent Programming: Languages, Platforms and Applications*, R. H. Bordini, M. Dastani, J. Dix and A. E. Fallah-Seghrouchni (eds), Multiagent Systems, Artificial Societies, and Simulated Organizations **15**, 149–174. Springer.

Przymusinski, T. C. 1988. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*, 193–216. Morgan Kaufmann.

Pylyshyn, Z. W. 1990. *Computation and Cognition*. Bradford/MIT Press.

Rao, A. S. 1996. AgentSpeak (L): BDI agents speak out in a logical computable language. In *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Proceedings*, 42–55. Springer.

Rao, A. S. & Georgeff, M. 1991. Modeling rational agents within a BDI-architecture. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, 473–484. Morgan Kaufmann.

Rao, A. S. & Georgeff, M. 1995. BDI agents: from theory to practice. in *Proceedings of the First International Conference on Multiagent Systems ICMAS95*, V. R. Lesser and L. Gasser (eds), 312–319. The MIT Press.

Reiter, R. 1980. A logic for default reasoning. *Artificial intelligence* **13**(1), 81–132.

Ricca, F. 2003. The DLV java wrapper. In *APPIA-GULP-PRODE*, 263–274. Citeseer.

Ricca, F., Dimasi, A., Grasso, G., Ielpa, S. M., Iiritano, S., Manna, M. & Leone, N. 2010. A logic-based system for e-tourism. *Fundamenta Informaticae* **105**(1–2), 35–55.

Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S. & Leone, N. 2012. Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming* **12**(3), 361–381.

Ricci, A., Viroli, M. & Omicini, A. 2007. CArtAgO: a framework for prototyping artifact-based environments in MAS. In *Environments for Multi-Agent Systems III, Third International Workshop, E4MAS 2006, Selected Revised and Invited Papers*, D. Weyns, H. Van Dyke Parunak and F. Michel (eds), LNCS **4389**, 67–86. Springer.

Rogers, T. J., Ross, R. & Subrahmanian, V. 2000. Impact: a system for building agent applications. *Journal of Intelligent Information Systems* **14**(2–3), 95–113.

R.Thomas, S. 1993. *PLACA, An Agent Oriented Programming Language*. PhD thesis, Computer Science Department, Stanford University. Available as Technical Report STAN-CS-93-1487.

Samsonovich, A. V. 2010. Toward a unified catalog of implemented cognitive architectures. *BICA* **221**, 195–244.

Shardlow, N. 1990. *Action and Agency in Cognitive Science*, Master's thesis, Department of Psycology, University of Manchester.

Shoham, Y. 1993. Agent-oriented programming. *Artificial Intelligence* **60**(1), 51–92.

Sloman, A. & Logan, B. 1998. Architectures and tools for human-like agents. In *Proceedings of the 2nd European Conference on Cognitive Modelling*, **58**, 65. University of Nottingham Press.

SOAR-Research-Group 2010. SOAR: a comparison with rule-based systems. `http://sitemaker.umich.edu/soar/home`

Son, T. C., Pontelli, E., Gelfond, M. & Balduccini, M. 2016. An answer set programming framework for reasoning about truthfulness of statements by agents. In *Technical Communications of the 32nd International Conference on Logic Programming*, ICLP 2016, OASICS **52**, 8:1–8:4. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Sridharan, M. 2016. Towards an architecture for representation, reasoning and learning in human-robot collaboration. In *2016 AAAI Spring Symposium Series*.

Tari, L., Baral, C. & Anwar, S. 2005. A language for modular answer set programming: application to ACC tournament scheduling. In *Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 3rd International ASP'05 Workshop*, CEUR Workshop Proceedings **142**, CEUR-WS.org.

Tiihonen, J., Soininen, T., Niemelä, I. & Sulonen, R. 2003. A practical tool for mass-customising configurable products. In *DS 31: Proceedings of ICED 03, the 14th International Conference on Engineering Design*.

Togelius, J. 2003. *Evolution of the Layers in a Subsumption Architecture Robot Controller*, Master's thesis, University of Sussex.

Torroni, P. 2004. Computational logic in multi-agent systems: recent advances and future directions. *Annals of Mathematics and Artificial Intelligence* **42**(1–3), 293–305.

Truszczyński, M. 2007. Logic programming for knowledge representation. In *Logic Programming*, 76–88. Springer.

Van Nieuwenborgh, D., De Vos, M., Heymans, S. & Vermeir, D. 2006. Hierarchical decision making in multi-agent systems using answer set programming. In *Computational Logic in Multi-Agent Systems*, 20–40. Springer.

van Riemsdijk, M. B., Dastani, M., Meyer, J.-J. C. & de Boer, F. S. 2006. Goal-oriented modularity in agent programming. In *5th International Joint Conference on Autonomous Agents and Multiagent Systems AAMAS 2006*, 1271–1278. ACM Press.

Vere, S. & Bickmore, T. 1990. A basic agent. *Computational Intelligence* **6**(1), 41–60.

Wood, M. F. & DeLoach, S. A. 2001. An overview of the multiagent systems engineering methodology. In *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000, Revised Papers*, P. Ciancarini and M. Wooldridge (eds), Lecture Notes in Computer Science **1957**, 207–222. Springer.

Wood, S. 1993. *Planning and Decision-Making in Dynamic Domains*. Ellis Horwood Series in Artificial Intelligence.

Wooldridge, M. 1997. Agent-based software engineering. *IEEE Proceedings on Software Engineering* **144**(1), 26–37.

Wooldridge, M. 1999. Multiagent systems. In *Multiagent Systems*, G. Weiss, (ed.), chapter on Intelligent Agents, 27–77. MIT Press. `http://dl.acm.org/citation.cfm?id=305606.305607`

Wooldridge, M. J. 2000. *Reasoning About Rational Agents*. MIT Press.

Wooldridge, M. & Jennings, N. R. 1994. Agent theories, architectures, and languages: a survey In *Intelligent Agents, ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Proceedings*, 1–39. Springer.

Wooldridge, M., Jennings, N. R. & Kinny, D. 2000. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems* **3**(3), 285–312.

Wooldridge, M. & Jennings, N. R. 1995. Intelligent agents: theory and practice. *Knowledge Engineering Review* **10**(2), 115–152.

Zhang, S., Sridharan, M. & Wyatt, J. L. 2015. Mixed logical inference and probabilistic planning for robots in unreliable worlds. *IEEE Transaction on Robotics* **31**(3), 699–713.

Zlog n.d. `www.exeura.eu/en/solution/customer-profiling`