The Knowledge Engineering Review, Vol. 30:4, 394–418. © Cambridge University Press, 2015
doi:10.1017/S0269888915000077

Programming multi-agent systems

MEHDI DASTANI

Intelligent Systems Group, Utrecht University, Princetonplein 5, 3584 CC Utrecht, The Netherlands;
e-mail: m.m.dastani@uu.nl

Abstract

With the significant advances in the area of autonomous agents and multi-agent systems in the last decade, promising technologies for the development and engineering of multi-agent systems have emerged. The result is a variety of agent-oriented programming languages, development frameworks, execution platforms, and tools that facilitate building and engineering of multi-agent systems. This paper provides an overview of the multi-agent programming research field and explains the aim and characteristics of various multi-agent programming languages and development frameworks. This overview is complemented with a discussion on the current trends and challenges in this research community.

1 Introduction

Multi-agent systems consist of interacting autonomous systems called agents. Agents are assumed to be autonomous in the sense that they decide for themselves which actions to perform in order to achieve their own individual objectives. Agents interact either with each other through communication or with their external environments through their sensors and actuators. One way to ensure global properties of multi-agent systems (i.e. overall system properties) is to control and coordinate the interaction between agents as well as the interaction between agents and environments (see e.g. Ferber, 1999; Weiss, 1999; Wooldridge, 2009). The coordination of agents can be done either endogenously or exogenously (see e.g. Arbab, 1998; Dastani *et al.*, 2005a). In an endogenous approach the coordination models reside within the agents, while in an exogenous approach the coordination models reside outside the agents. In particular, in an endogenous approach agents are internally designed and developed to behave in a controlled and coordinated manner, while in an exogenous approach agents are controlled and coordinated by means of an external component.

Multi-agent systems constitute a promising software engineering approach for the development of applications in complex domains where interacting application components are autonomous and distributed, operate in dynamic and uncertain environments, have to respect some organisational rules and laws, and can join and leave the system at runtime (see e.g. Zambonelli *et al.*, 2003). Examples of such applications are systems that manage and optimise the generation and distribution of electricity among consumers, systems that optimally schedule and assign loads to vehicles in transportation systems, or conference management systems. In a conference management system, application components are, for example, individual agents (associated to the individuals involved in a conference) that interact by submitting, reviewing, or accepting/rejecting papers. The external environment with which individual agents interact consists of databases that store information about authors and reviewers, submitted papers, reviews, and the final decision for papers. While individual agents can decide to submit and review papers, their activities are exogenously controlled and coordinated by some conference rules and laws. For example, reviewers are not permitted to review their own papers, the conference chair is responsible that each paper receives a certain number of reviews, and papers with more than a certain number of pages are rejected. The development of such applications requires the development of software agents (or interfaces that enable humans to interact with the system), environments (e.g. databases for papers and reviews), and organisation processes that control and coordinate the activities of individual agents (e.g. monitoring the upload of papers and reviews, while enforcing organisational rules and laws).

A distinguishing feature of multi-agent systems compared with other software development approaches is the repertoire of high-level social/cognitive concepts and abstractions that multi-agent systems provide to specify, design, and implement software systems in complex domains. Examples of such concepts and abstractions are beliefs, goals, plans, actions, events, roles, organisational rules and structures, communication, norms and sanctions. In order to build multi-agent systems in an effective and systematic way, different methodologies (e.g. Bergenti *et al*., 2004), specification languages (e.g. Cohen & Levesque, 1990; Rao & Georgeff, 1991; Meyer *et al*., 1999; Dastani *et al*., 2010), programming languages and development tools (e.g. Hindriks *et al*., 1999; De Giacomo *et al*., 2000; Leite *et al*., 2001; Bracciali *et al*., 2004; Kakas *et al*., 2004; Sardina *et al*., 2004; Bordini *et al*., 2005, 2009; Fisher, 2005; Pokahr *et al*., 2005; Sadri, 2005; Winikoff, 2005; Bordini *et al*., 2007; Dastani, 2008) have been proposed. While methodologies assist system developers to specify and design software systems in terms of multi-agent system concepts and abstractions, the main challenge of the proposed multi-agent programming languages and development frameworks is to provide programming constructs and operations that facilitate explicit and effective implementation of multi-agent system concepts and abstractions.

In general, the development of multi-agent systems requires the development of three different types of entities: individual agents, multi-agent organisations, and multi-agent environments. Although ontological differences between these entities and their implications for programming multi-agent systems have been emphasised during the early ProMAS technical fora (see e.g. Dastani & Gomez-Sanz, 2006), the main focus of the multi-agent programming community has been the development of individual agents. Multi-agent organisations and environments have been active research areas for many years and have resulted in a variety of models and architectures. However, the need for programming languages that support the implementation of multi-agent organisations and environments has only recently been recognised. It is important to emphasise that the term multi-agent system environment has been used with different meanings. For example, it is used to refer to what is external to the system (i.e. the embedding environment of the system), to the supporting run-time infrastructures of the system (i.e. execution platform or communication middleware), or to the non-agentified part of the system with which individual agents interact (i.e. databases or services). As the focus of this paper is on multi-agent programming languages and development frameworks, we ignore the first and the second uses of the notion of environment. The first notion of environment does not refer to a software component that needs to be developed, and the second notion of the environment refers to an infrastructure that is assumed to be given and used by the software developers to build and execute their software systems.

This paper starts with a discussion on the aims and objectives of the multi-agent programming research field. Subsequently, it presents and discusses concepts and abstractions for which multi-agent programming languages and development frameworks intend to support their implementations. The paper proceeds with presenting an overview of the state of the art in multi-agent programming and explains the aims and characteristics of some existing multi-agent programming languages and development frameworks. Of course, there are too many programming languages and development frameworks to mention in this paper. The programming languages covered in this paper are selected because they have an interpreter and an execution platform. Current trends in this research field will be explained and discussed by means of recent foci and developments of multi-agent programming languages. Finally, the paper discusses issues that are currently challenging the multi-agent programming research field.

2 Aims and objectives

Multi-agent systems can be seen as an advance in software engineering that has resulted in new software development methodologies. A characteristic feature of these methodologies is that they provide high-level concepts and abstractions to model and develop complex distributed intelligent systems. Like other conventional software development approaches, multi-agent systems cover different phases such as requirement, specification, design, implementation, and testing. Various multi-agent system software methodologies have been proposed, for example, Gaia (Zambonelli *et al*., 2003), Prometheus (Padgham & Winikoff, 2003), Tropos (Bresciani *et al*., 2003), INGENIAS (Gomez-Sanz & Pavon, 2003), and others (see Bergenti *et al*., 2004). Each methodology focuses on specific phases of the software development process. For example, Gaia focuses mainly on the analysis and design phases, while Prometheus covers the implementation phase as well. Existing multi-agent system software development methodologies propose concepts and abstractions such as beliefs, goals, plans, events, roles, interaction, agents, environment, organisation rules, norms, permission, responsibility, and capability.

The main aim of the multi-agent programming research field is to propose programming languages and development frameworks that can facilitate direct and effective implementations of multi-agent systems. From the software development perspective, the proposed programming languages and development frameworks should support the implementation phase of multi-agent development methodologies. In fact, one can see a multi-agent programming language or a development framework as a computational specification language for implementing a certain class of multi-agent system architectures. As different multi-agent system development methodologies propose different abstractions and architectures, one may argue that a standard general purpose multi-agent programming language or development framework cannot emerge as long as different multi-agent system development methodologies do not converge in the sense of agreeing on a shared set of abstractions and an architecture. Although multi-agent programming languages provide different sets of programming constructs, most of them are considered as general purpose programming languages that are not aimed at a particular application domain. It should be emphasised that while these programming languages are application independent, they are dedicated to multi-agent systems in the sense that they are designed to support the implementation of any multi-agent system.

Multi-agent programming languages and development frameworks can be characterised along different dimensions. First, they can be characterised by means of concepts and abstractions for which they provide dedicated programming constructs. The concepts and abstractions, which are often motivated by multi-agent system development methodologies, can be classified as being concerned with individual agents, multi-agent organisations, or multi-agent environments. Some programming languages and development frameworks provide constructs to implement individual agent concepts and abstractions such as beliefs, goals, plans, autonomous behaviour, reactive behaviour, social awareness, reasoning about norms and organisations, and communication with other agents. Other languages and frameworks aim at facilitating the implementation of social and organisational concepts and abstractions such as norms (obligation, prohibition, permission) that should be respected or followed by agents, sanctions that should be imposed on agents if they violate norms, roles that can be enacted by agents, delegation of tasks and responsibilities, or the synchronisation of agent's actions. Yet, other languages and frameworks focus on the multi-agent environment and provide programming constructs to implement concepts such as sense and act abilities of agents (i.e. to implement the effect of actions in environment), tools, artifacts, services, and resources.

Second, multi-agent programming languages and development frameworks can be characterised by their language styles and their formal or practical foundations. In particular, they can be declarative, imperative, or a combination of them. Some multi-agent programming languages and development frameworks such as Jade (Bellifemine *et al*., 2005), JACK (Winikoff, 2005), and KGP (Kakas *et al*., 2004) are based on (or extend) existing programming languages such as Java or Prolog, while others such as 2APL (Dastani, 2008), GOAL (Hindriks, 2009), and Jason (Bordini *et al*., 2007) combine Java and Prolog. Declarative languages support the representation of and reasoning with concepts such as beliefs, goals, norms, and actions, while imperative languages facilitate the implementation of tasks, services, and processes. Some multi-agent programming languages have been proposed as a theoretical contribution by means of abstract syntax and operational semantics without having a corresponding interpreter or an execution platform (e.g. van Riemsdijk *et al*., 2003), while other languages and frameworks come without operational semantics but with their corresponding development tools and execution platforms (e.g. Bellifemine *et al*., 2005; Pokahr *et al*., 2005; Winikoff, 2005). Finally, some multi-agent programming languages come with formal and computational semantics, an implemented interpreter, or both (e.g. Bordini *et al.*, 2007; Dastani, 2008; Hindriks, 2009). The existence of formal semantics for multi-agent programming languages is essential for a better understanding of the programming constructs and the verification of multi-agent programs. Without a formal semantics one cannot guarantee the correctness of programs.

Third, multi-agent programming languages and development frameworks can be characterised by means of general programming principles that they support. Examples of such principles are modularity, encapsulation, reuse, separation of concerns, recursion, abstraction, exception handling facilities, and support for legacy codes. Of course, the very concept of agent itself supports some of these principles such as encapsulation and reuse. Moreover, individual agents, environments, and organisations are different concerns and constitute different foci of multi-agent systems. Therefore, the idea of implementing agents, environments, and organisations separately supports the separation of concerns principle. Multi-agent programming languages can be used in a more efficient and effective manner when they support various principles at different levels. For example, at the individual agent level modularity can be used to support the implementation of different functionalities and roles, recursion can be used to implement complex plans, and exception handling can be used to implement plan failure operations.

Finally, multi-agent programming languages and development frameworks can be characterised by means of their integrated development environments and their corresponding functionalities. An integrated development environment supports the development of multi-agent programs with functionalities such as editors with syntax highlighting and autocomplete features, search operations allowing easy browsing of code, debugging tools that help to localise errors and anomalies, and automatic testing tools allowing the automatic generation of test cases for specific parts of the programs. A major difficulty for building an integrated development environment is the distributed nature of multi-agent programs, for example, how to browse through a program that consists of agents, environments, and organisations possibly running on different machines. Debugging multi-agent programs is even harder as it is not clear how to debug an agent program when the execution of the agent program depends on the execution of other agent programs, the environment programs, and the organisation programs. It is important to emphasise that an integrated development environment is different from and should not be confused with a multi-agent system environment.

3 Abstractions in multi-agent programming

In this section, we present key concepts and abstractions that need to be addressed when programming multi-agent systems. The following subsections reflect the highest level of abstraction in multi-agent systems: agents, organisations, and environments. Each subsection discusses these abstractions in detail and explores different ways that they can be implemented.

3.1 Individual agents

The focus of most multi-agent programming languages and development frameworks has been on programming individual agents (see e.g. Bordini *et al.*, 2005, 2009). In these works, multi-agent programs are considered as comprising a set of individual agent programs that are executed concurrently.

An essential characteristic of individual agents is their autonomy. An agent is called autonomous if it has the ability to decide and perform actions at each moment of time in order to achieve its objectives (e.g. Woolridge, 2002). Without getting into the exact nature of autonomy, we consider an agent as autonomous if it has a decision-making component that governs its decisions based on its informational (e.g. belief, probability distribution, knowledge), motivational (e.g. desire, utility, preference), and deliberational (e.g. intention, plan, commitment) attitudes. One can argue that any computational system that interacts with other systems (e.g. other agents or software components) can be seen as autonomous, at least from an external point of view. However, the development of an autonomous software agent can also be considered from an internal point of view. Such a view requires an explicit decision-making component that can be specified, designed, implemented in terms of informational, motivational, and deliberational attitudes. The decision-making component should allow a programmer to implement different issues related to an agent's decisions such as decision strategies, resolving decision conflicts, and rationality of decisions. In this sense, programming languages that support the implementation of autonomous agents should provide programming constructs to support the implementation of decision concepts and mechanisms.

Different decision models can be used for developing the decision component of autonomous agents, for example, Partially Observable Markov Decision Process model (POMDP) (see e.g. Tasaki *et al.*, 2008), Belief-Desire-Intention model (BDI) (see e.g. Cohen & Levesque, 1990; Rao & Georgeff, 1991), or a combination of both (see e.g. Nair & Tambe, 2005). POMDP is a quantitative framework that can be used to model a sequential decision process in terms of actions, states, transition probability, observation probability, and reward function. In order to determine an agent's decisions, its corresponding POMDP (the agent's decision model) should be solved. However, solving POMDP's are in general computationally intractable such that approximate methods are often proposed to solve POMDP's. Moreover, POMDPs are not suitable to model agents that have complex goals and need to interact with dynamically changing environments (see e.g. Rens *et al.*, 2009). These problems make POMDP less plausible to be integrated in programming languages and development frameworks for individual agents.

In contrast, the BDI model can be seen as a qualitative decision model that explains an agent's rational decision in terms of the agent's information about the current state of the world (Belief), the states the agent wants to achieve (Desire), and its commitments to already made choices (Intention). The BDI model has proven to be an efficient model for reactive planning and for the agents that have complex goals interacting with highly dynamic environments (see e.g. Bratman *et al.*, 1988). Some existing BDI-based agent programming languages provide constructs to implement an agent's beliefs, goals, and conditional plans. The conditions that are assigned to plans are specified in terms of beliefs and goals such that a plan can be decided/selected if its belief and goal conditions are satisfied by the agent's beliefs and goals, respectively, and moreover the plan is not in conflict with the existing plans. Plan conflict can be defined in various ways such as realising inconsistent states or being in conflict with respect to the use of resources. In some BDI-based programming languages (e.g. Dastani, 2008) a plan is selected if it is not already generated to achieve the same goal. The reasoning engine of the BDI agents often involves a process that continuously decides a plan to execute.

Another characteristic of an individual agent is its reactive behaviour (see e.g. Muller, 1996). The implementation of reactive agents requires an event handling mechanism that generates reactions to the received events. There are many types of events such as messages that are received from other agents, information that are originated from the environment, and the information about the internal working of agents (e.g. the failure of a plan or the updates of beliefs and goals). Programming languages and development frameworks that support the implementation of reactive agents provide constructs to implement conditional plans where the plan's conditions are defined in terms of events. The reasoning engine of reactive agents continuously checks if plans can be selected based on the received events. It should be noted that there is an essential difference between events and goals. In principle, an event causes the generation of a plan and, as soon as the plan is generated, the event is deleted and considered as being processed. Goals are similar to events in the sense that they cause the generation of plans. However, and in contrast to events, goals are not dropped after they have caused the generation of plans. An achieve goal is, for example, dropped if the state denoted by it is achieved, that is, if the agent believes that the state denoted by the goal is achieved. The relation between beliefs and goals is an essential characteristic of BDI agents, which is formulated by means of rationality axioms in the BDI logics (see e.g. Cohen & Levesque, 1990; Rao & Georgeff, 1995). It should also be noted that autonomy and reactive behaviour are two different characteristics and that agents can be both autonomous and reactive, be autonomous without being reactive, or vice versa.

3.2 Multi-agent environment

Soon after the emergence of the first generation of the execution platforms for agent-oriented programming languages and development frameworks, the need for the implementation of shared environments with which software agents can interact became apparent. This need was manifested by applications in which software agents have to interact with non-agentified software components, for example, shared resources or services. An immediate solution was to model an environment as an external software component with which individual agents interact. Such a software component can be an implementation of resources and services, or an interface to other (software of natural) systems that are external to the developing multi-agent system. In most BDI-based multi-agent programming languages (e.g. Pokahr *et al.*, 2005; Bordini *et al.*, 2007; Dastani, 2008; Hindriks, 2009) an environment is simply a software component that is implemented in the same programming language as that of the interpreter of the agent-oriented programming language (e.g. Java or C++). In other programming frameworks (e.g. Bellifemine *et al.*, 2005; Winikoff, 2005) an environment is a software component implemented in Java. In both cases, the state of the software component is considered as the state of the environment, while the methods that allow the interaction with the software component are used to implement the effect of the actions that agents could perform in the environment. In fact, the repertoire of actions that an agent can perform in an environment is determined by the methods of the corresponding software component. In this view, a software agent decides which method to call (i.e. which action to select and perform) and the environment program determines the effect of the action.

One of the first overview papers in the field of multi-agent system environments emphasised that the concept of environment was originally used with different meanings causing confusion about the exact nature of this abstraction (see e.g. Weyns *et al.*, 2005; Van Dyke Parunak & Weyns, 2007). As argued in this overview paper, some researchers consider a multi-agent system environment as the run-time environment and as equivalent of infrastructures such as message transport system and other infrastructural tools, for example, brokers and management tools, while other researchers consider a multi-agent system environment as the embedding environment of the multi-agent system consisting of entities external to the multi-agent software system. Yet, other researchers consider a multi-agent environment as a component that encapsulates resources, services, and objects. Existing agent programming languages (e.g. Pokahr *et al.*, 2005; Bordini *et al.*, 2007; Dastani, 2008; Hindriks, 2009) consider an environment as a first-class abstraction in multi-agent system that encapsulates resources and services and has its own state and processes.

It should be emphasised that from the programming point of view, it does not make sense to consider other notions of environment. For example, the notion of environment as an entity external to the multi-agent system refers to something to be assumed by the system developer and not an entity to be programmed. In addition, the notion of environment as an infrastructure denotes a set of tools and facilities that a multi-agent system developer can assume as given and not something that the system developer has to program for each application separately. Note also that the concept of multi-agent environment as encapsulating resources and services is different from agent-based simulation environment. An agent-based simulation environment can be used to model, execute, and analyse agent-based simulations. An agent-based simulation environment can be compared with an integrated development environment for multi-agent systems. In particular, agent-based simulation environments can be seen as special cases of integrated development environments where the focus is on tools for analysing simulation behaviours rather than general purpose tools for the development of multi-agent systems.

In the same overview paper, a multi-agent system environment is claimed to be used for various purposes, for example, to facilitate and coordinate agent's interactions by means of exchanging information through it (blackboard architectures and tuple spaces)[1], or to provide agents the sense and act abilities in order to observe and modify the environment's state, respectively. For example, an environment can provide artifacts or services to allow agents to manage their coordination or to exchange information. An environment can also provide various sense and act modalities such as blocking and non-blocking sense operations, event broadcasting, event subscription mechanisms, and synchronous or asynchronous actions.

The implementation of environments therefore requires dedicated programming languages and development frameworks that allow direct and effective implementations of its related concepts

[1] *In* contrast to direct communication by means of send and receive messages, a shared environment can be used to communicate indirectly by reading and writing information from/to it.

and abstractions such as resources, services, and objects. Like the development in agent-oriented programming languages, one may expect typical architectures for multi-agent environments. Such architectures would suggest specific concepts, concerns, or components that often need to be implemented when developing a multi-agent system environment. In particular, a dedicated environment programming language or development framework should provide programming constructs to implement resources, services, artifacts, processes, several sense, and action types and mechanisms. The A&A model (see e.g. Omicini, 2007) has been proposed as a generic approach for modelling environments. In the A&A model, an application is composed of agents as well as the so-called artifacts. An implementation of the A&A model is available in form of the distributed architecture and middleware infrastructure Cartago (see e.g. Ricci *et al.*, 2006). Such an environment architecture consists of a dynamic set of artifacts, each of which encapsulates resources, services, or objects designed by the environment developer.

3.3 Multi-agent organisation

The overall objectives of a multi-agent system depend on the behaviours of its participating individual agents that pursue their own objectives. The objectives of the participating individual agents may be different from, and even be in conflict with, the overall objectives of the multi-agent system. One way to guarantee the overall objectives of a multi-agent system is to control and coordinate the behaviour of its individual agents and their interactions. This can be done either endogenously by integrating the organisation mechanism within the agents themselves, or exogenously by designing the organisation mechanism outside the agents, or a combination of both. An endogenous organisation implies that agents are internally designed and implemented to follow, for example, specific interaction protocols, norms, or organisational rules. In an exogenous approach agents are coordinated by means of external components that control the agent's actions according to some interaction protocols, norms, or organisational rules. Generally speaking, endogenous coordination mechanisms can be used for the development of closed multi-agent systems where the set of participating agents is predefined by the system designer. Exogenous coordination mechanisms can be used for open multi-agent systems where software agents can join and leave the system at run time. It should be noted that an exogenous organisation can also be effective for the development of closed multi-agent systems as such an approach supports the separation of concerns and encapsulation principles.

There have been various proposals for regulating and organising the behaviours of individual agents. Some of these proposals advocate the use of coordination artifacts that are specified in terms of low-level coordination concepts such as synchronisation (see e.g. Arbab *et al.*, 2009). Other approaches are motivated by organisational models, normative systems, or electronic institutions (e.g. Jones & Sergot, 1993; Esteva *et al.*, 2002, 2004; Grossi, 2007; Dastani *et al.*, 2009b; Hübner *et al.*, 2010). In these approaches, the behaviours of individual agents are regulated by means of norms and organisational rules that are either used by individual agents to decide how to behave, or being enforced or regimented through monitoring and sanctioning mechanisms. In these approaches, the social and normative perspective is conceived as a way to make the development and maintenance of multi-agent systems easier to manage. A plethora of social concepts (e.g. roles, groups, social structures, organisations, institutions, norms) has been introduced in multi-agent system methodologies such as Gaia (Zambonelli *et al.*, 2003), models such as OperA (Dignum, 2004), specification and modelling languages such as $\mathcal{S}-\mathcal{M}$OISE$^+$ (Hübner *et al.*, 2006) and ISLANDER (Esteva *et al.*, 2002), and computational frameworks such as AMELI (Esteva *et al.*, 2004).

The implementation of organisations requires programming languages and development frameworks that provides programming constructs to implement social and organisational concepts and abstractions. In particular, the implementation of endogeneous mechanisms implies that the agent programming languages provide constructs to allow the representation and reasoning about norms, sanctions, and organisational rules. Such constructs should allow multi-agent programmers to implement agents that make their decisions not only based on their individual goals and beliefs, but also based on the existing interaction protocols, norms, sanctions, and other organisational rules. The idea is that individual agents can be implemented in terms of cognitive and social abstractions such that their behaviours are determined based on reasoning about such abstractions. The implementation of exogenous mechanisms requires abilities to monitor and control the behaviours of individual agents. The idea is to have an external organisation software component that is able to monitor and control the behaviour of individual agents. The question is what should be monitored and how the agent's behaviours can be influenced. As the internals of individual agents cannot be assumed in general, their external behaviours (i.e. communication and interaction with the environment) are the only controllable entities. The organisation software component can thus observe agent's external behaviour and determine what needs to be done. For example, if the organisation specification disallows certain agents to interact, then the organisation software should be able to block or respond to such interactions. This suggests that the agent's actions (e.g. communication, environment actions including sense actions) should be processed and managed through the external organisation component, that is, the organisation component intermediates the interaction between agents as well as the interaction between agents and the environment.

4 The state of the art in multi-agent programming

In this section, we provide an overview of some of the existing multi-agent programming languages and development frameworks. This overview is by no means complete and does not cover some related or relevant multi-agent programming languages and development frameworks. The programming languages and frameworks in this overview are chosen because they illustrate different ways to program (some of) the abstractions discussed in the previous section, have execution platforms, and of course, because of the author's familiarity with the languages and frameworks. Other multi-agent programming languages and development frameworks can be found in Bordini *et al.* (2005, 2009). This overview will be structured along the main focus of the languages and development frameworks on individual agents, multi-agent environments, and multi-agent organisations. The programming languages and development frameworks together with their corresponding execution platforms will be discussed in terms of concepts and abstractions as explained in Section 3.

4.1 Programming frameworks for individual agents

One of the earliest agent-oriented programming languages is `AGENT-0` proposed by Shoham (1993). In his seminal paper, Shoham proposes to implement agents in terms of mental components such as beliefs, commitments, capabilities, and actions. An agent program in `AGENT-0` consists of an initial belief base, a set of capabilities, a set of commitment rules, together with a repertoire of private actions. Agents can perform different types of actions such as communication, private, conditional, and unconditional actions. Agents enter into new commitments by means of commitment rules. A commitment rule consists of conditions on an agent's mental state and the incoming messages. The application of a commitment rule generates a commitment consisting of an action together with the agent identifier towards whom the commitment is made. In fact, the commitments define the actions that an agent have to perform. The execution of an agent is a continuous cyclic process. At each cycle, the received messages are processed, commitments are generated, and actions are performed. `AGENT-0` is undoubtedly one of the first attempts to develop an agent programming language that supports the implementation of autonomous agents, that is, agents that decide actions based on their mental states. However, as indicated in the discussion section of this seminal paper, the state of an `AGENT-0` agent lacks motivational attitudes such as utility, desire, goal, or preference such that an agent's decisions are based only on events and messages rather than the agent's motivational attitude.

As the introduction of `AGENT-0` various agent-oriented programming languages have been proposed that extend `AGENT-0` with a larger repertoire of agent concepts and abstractions. The aim of these programming languages is to support the implementation of multi-agent systems, although most of them do not support the implementation of abstractions concerning multi-agent environment and organisation. Some of these agent-oriented programming languages have an imperative programming style as they extend Java with agent concepts and abstractions, some languages have a declarative programming style as they extend logic programming languages, and yet other programming languages combine both imperative and declarative styles by integrating, for example, Java and Prolog. The programming languages that are based on Java have no explicit formal semantics. In the following, we give a brief overview of some of these programming languages.

4.1.1 Imperative style programming frameworks

Jade (Java Agent DEvelopment framework) as presented in Bellifemine *et al.* (2005) extends Java with a set of agent concepts and abstractions. An agent is created by extending a predefined Jade agent class and redefining its `setup` method. After an agent is created, it receives an identifier and is registered with the agent management system (a Jade built-in service). The agent is then put in the active state and its `setup` method is executed. The `setup` method is therefore the point where any agent activity starts. Jade agents are behaviour based in the sense that they can create and execute behaviours. A behaviour can be created by extending the Jade behaviour class via a special construct that adds behaviours (initially in the `setup` method). The created behaviours are added to a behaviour pool. Behaviours are selected for execution from this pool based on a scheduler that constitutes the execution model of the Jade agents. Agents are executed concurrently as different pre-emptive Java threads. The Jade framework is developed for practical and industrial applications and comes with a development environment providing a set of graphical tools that can be used to monitor and log the execution of multi-agent programs. The Jade execution platform is based on a middleware that facilitates the development of distributed multi-agent applications based on a peer-to-peer communication architecture. The platform is distributed in the sense that it can run over multiple machines, while seen as a whole from the outside world. The Jade platform implements the basic services and infrastructure of a distributed multi-agent application. It supports agent life-cycle, agent mobility, and agent security, and provides services such as white and yellow pages that can be used by the agents to register their services and search for each other.

Other Java-based agent programming languages are Jadex and JACK. Jadex, as proposed in Pokahr *et al.* (2005), builds on Jade and extends it with programming constructs to implement BDI concepts such as beliefs, goals, plans, and events. It uses XML notation to define and declare an agent's BDI ingredients and Java constructs to implement the agent's plans. JACK, as presented in Winikoff (2005), extends Java with programming constructs to implement BDI concepts. In both JACK and Jadex, a number of syntactic constructs are added to Java to allow programmers to declare beliefsets, to post events, and to select and execute plans. The execution of agent programs in both languages are motivated by the classical sense–reason–act cycle, that is, processing events, selecting relevant and applicable plans, and execute applicable plans. Beliefs and goals in JACK and Jadex have no logical semantics such that an agent cannot reason about its beliefs and goals. It should be noted that the ability to reason with beliefs and goals allows agents to be more flexible in goal achievement in the sense that they can achieve goals partially and gradually. Moreover, the consistency and the rational balance of an agent's state in JACK and Jadex, as far as they are defined, is left to the agent programmer, that is, the agent programmer is responsible to make sure that state updates preserve the state consistency and that the rational balance (e.g. between beliefs and goals) is maintained. In these programming languages, an agent's goal is not automatically dropped because it is derivable from the agent's beliefs. Jadex provides a programming construct to implement non-interleaving execution of plans. Jadex and JACK come also with integrated development environments and provide monitoring and logging facilities, similar to those proposed in the Jade framework.

Another Java-based agent programming framework is Agent Factory (AF), which comes with its corresponding programming language called Agent Factory Agent Programming Language (AFAPL), as described in Muldoon *et al.* (2009). Although this programming framework has been the subject of continuous development and modification, its distinguishing feature is its practical focus on application domains that involve mobile and ubiquitous devices such as wireless sensor networks. AFAPL supports the implementation of agents based on cognitive concepts such as beliefs, goals, commitments, plans, and roles. An agent is implemented in terms of commitment rules and the applications of these rules, which are based on the agent's state, generate commitments. An agent's behaviour is then determined based on the notion of commitment management using commitment strategies. These strategies are responsible for the adoption and maintenance of commitments, refining commitments to plans and actions, and managing failed commitments. This programming framework consists of a distributed run-time environment, development kits, and a structured approach to the deployment of multi-agent systems.

4.1.2 Declarative style programming frameworks

KGP (Knowledge, Goal, and Plan), as presented in Kakas *et al.* (2004), Bracciali *et al.* (2004), and Sadri (2005), is a declarative model of agency characterised by a set of modules. The model is based on computational logic and logic programming techniques, and has an internal state module consisting of a collection of knowledge bases, the current agent's goals and plans. The knowledge bases represent different types of knowledge such as the agent's knowledge about observed facts, actions, and communication, but also knowledge to be used for planning, goal decision, reactive behaviour, and temporal reasoning. The KGP agent model includes also a module consisting of a set of capabilities such as planning, reactivity, temporal reasoning, and reasoning about goals. These capabilities are specified by means of abductive logic programming or logic programming with priorities. Another KGP module contains a set of transitions to change the agent's internal state. Each transition performs one or more capabilities, which in turn use different knowledge bases, in order to determine the next state of the agent. Finally, the KGP model has a module, called cyclic theory, that determines which transition should be performed at each moment of time.

Minerva, as presented in Leite *et al.* (2001), aims at specifying an agent's state and its dynamics. A Minerva agent consists of a set of specialised subagents manipulating a common knowledge base, where subagents (i.e. planner, scheduler, learner, etc.) evaluate and manipulate the knowledge base. These subagents are assumed to be implemented in arbitrary programming languages. Minerva gives both declarative and operational semantics to agents allowing the internal state of the agent, represented by logic programs, to modify. Minerva is based on multidimensional dynamic logic programming and uses explicit rules for modifying its knowledge bases.

The family of Golog languages as presented in De Giacomo *et al.* (2000) and Sardina *et al.* (2004) propose high-level program execution as an alternative for controlling the behaviour of agents that operate in dynamic environments with partial observation. In fact, the high-level (agent) program consists of a set of actions, including the sense action (e.g. IndiGolog as presented in Sardina *et al.*, 2004), composed by means of conditionals, iteration, recursion, concurrency, and non-deterministic operators. Instead of finding a sequence of actions to achieve a desired state from an initial state, the problem is to find a sequence of actions that constitute a legal execution of the high-level program. When there is no non-determinism in the agent program, then the problem is the straight forward execution of actions in the agent program. On the other hand, when the agent program consists of actions that are composed only by non-deterministic operators, then the problem is identical to the planning problem.

Concurrent MetateM, as proposed by Fisher (2005), is based on the direct execution of an extension of propositional temporal logic specifications. A multi-agent system in Concurrent MetateM consists of a set of concurrently executing agents with the ability to communicate asynchronously. Each agent is programmed by means of a temporal logic specification of the behaviour that the agent have to generate. In particular, it consists of rules that can be fired when their antecedents are satisfied with respect to the execution history. The consequent of a fired rule, which can be a temporal formula, forms the commitment of the agent that needs to be satisfied. The execution of an agent builds iteratively a logical model for the temporal agent specification. In Concurrent MetateM, the beliefs of agents are propositions extended with modal belief operators (allowing agents to reason about each others' beliefs), goals are temporal eventualities, and plans are primitive actions.

CLAIM, as proposed by El Fallah Seghrouchni and Suna (2005), is a declarative multi-agent programming language focusing on mobile agents. It comes with a distributed platform called SyMPA that enables the execution of multi-agent programs. A multi-agent system in CLAIM is a set of hierarchies of agents distributed over a network. An agent in CLAIM can be a subagent of another one such that the hierarchies determine the parent–child relation between agents. Agents in CLAIM are BDI based and can be programmed in terms of knowledge, goals, capabilities, messages, parents and children. Agents can migrate within a hierarchy as well as between hierarchies by means of the move operation. The migration of agents in CLAIM is a strong migration, that is, the state of the agent just before the migration is saved, encrypted, and transferred to the destination. At the destination, the agent's state is restored and processes are resumed from their interruption point.

4.1.3 Hybrid style programming frameworks

3APL (An Abstract Agent Programming Language), as originally proposed by Hindriks *et al.* (1999), is a programming language for single agents. The state of an agent in 3APL consists of (declarative) beliefs and plans, where plans consist of belief update, test, and abstract actions composed of sequence, conditional choice, and iteration operators. This version of 3APL provides only plan revision rules that are applied to revise an agent's plan. The execution of a 3APL agent program is a cyclic process. At each cycle a plan revision rule is selected and applied after which a plan from the plan base is selected and executed. The execution of a plan modifies the belief base of the executed agent program. This original version of 3APL was an abstract programming language that lacked a development and execution platform. This version is extended by Dastani *et al.* (2005b) with declarative goals and a variety of action types. In addition, an execution platform is developed for the extended version of 3APL.

2APL (A Practical Agent Programming Language), as proposed by Dastani (2008), is developed to implement multi-agent systems. It provides two sets of programming constructs to implement multi-agent and individual agent concepts. The multi-agent programming constructs are provided to create individual agents, external environments, and to specify the agent's access relations to the external environments. In 2APL, an environment (Java object) has a state and can execute a set of actions (method calls) to change its state. At the individual agent level, 2APL agents are implemented in terms of beliefs, goals, actions, plans, events, and three different types of rules. The beliefs and goals of 2APL agents are implemented in a declarative way, while plans and (interfaces to) external environments are implemented in an imperative style. The declarative part of the programming language supports the implementation of an agent's reasoning and update mechanisms. The imperative part of the programming language facilitates the implementation of plans, control flow, and mechanisms such as procedure call, recursion, and interfacing with legacy code. 2APL agents can perform different types of actions such as belief update actions, belief and goal test actions, external actions (including sense actions), actions to manage the dynamics of goals, and communication actions. Three types of rules are used to generate plans. The first type of rule is designed to generate plans to achieve goals, the second to process (internal and external) events/messages, and the third to repair failed plans. Finally, 2APL comes with a development environment with tools to log and monitor the execution of multi-agent programs. These tools are similar as those provided by the Jade framework.

GOAL, as proposed by Hindriks (2009), is a BDI-based programming language developed to implement autonomous agents. It provides programming constructs to implement an agent's knowledge, beliefs, and goals declaratively. It also provides programming constructs to implement action selection rules that can be used to select actions based on the agent's current knowledge, beliefs, and goals. A characteristic feature of GOAL is the distinction between knowledge and beliefs. Knowledge represents an agent's general information that are not the subject of modification, for example, the agent's domain knowledge, while beliefs represents an agent's current information that can be modified during the agent execution, for example, by sensing the environment or performing mental actions. Another characteristic feature of GOAL is the absence of plans. The action selection rules generate only atomic actions when they are applied. GOAL provides different types of actions such as user-defined actions, built-in actions, and the communication actions. The execution of a GOAL agent is a cyclic process where at each cycle the agent senses the environment, applies action selection rules, and performs the generated actions. The development environment of GOAL can be used to log and monitor the execution of multi-agent programs.

Jason, as proposed by Bordini *et al.* (2007), is introduced as an interpreter of an extension of AgentSpeak, which is originally proposed by Rao (1996). Jason distinguishes multi-agent system concerns from individual agent concerns An individual agent in Jason is characterised by its beliefs, plans, and the events that are either received from the environment or generated internally. A plan in Jason is designed for a specific event and belief context. The execution of individual agents in Jason is controlled by means of a cycle of operations encoded in its operational semantics. In each cycle, events from the environment are collected, an event is selected, a plan is generated for the selected event and added to the intention base, and finally a plan is selected from the intention base and executed. A plan rule in Jason indicates that a plan should be generated by an agent if an event is received/generated and the agent has certain beliefs. Jason is based on first-order representation for beliefs, events, and plans. Jason has no explicit programming construct to implement declarative goals, though goals can be simulated indirectly by means of a pattern of plans. Moreover, the beliefs and plans in Jason can be annotated with additional information that can be used in belief queries and plan selection process. Finally, plan failure in Jason can be modelled by means of plans that react to the so-called deletion events. The development environment of Jason provides tools to log and monitor the execution of multi-agent programs.

IMPACT, as proposed by Dix and Zhang (2005), is a project that aims at developing a multi-agent system platform. This project is based on the idea of agentisation, that is, agents are built around given legacy code. The multi-agent system platform comes with a programming language and its formal semantics. An agent is built around a legacy code by abstracting from the legacy code and describing its main features. In particular, an agent is specified in terms of the set of all datatypes managed by the legacy code, a set of functions over the datatypes allowing external processes to access the datatypes, and a set of composition operators that are defined on the datatypes and generate new composed datatypes. The state of an agent is determined by the state of the data in terms of which the agent is defined. Each agent has a set of actions that it can perform in its environment. An action can have different status such as permitted, obliged, or forbidden. The execution of an agent follows a cycle where messages from other agents are processed (which may in turn change the data and thus its state), the status of each action is determined, the actions that can be executed are determined, and the state is updated accordingly.

4.2 Programming frameworks for multi-agent organisations

In the literature on multi-agent systems, there have been many proposals for specification languages and logics to specify and reason about normative multi-agent systems, virtual organisations, and electronic institutions (see e.g. Jones & Sergot, 1993; Prakken & Sergot, 1996; Ågotnes *et al.*, 2008; Boella & van der Torre, 2008). How to develop, program, and execute such normative systems was one of the central themes that were discussed and promoted during the AgentLink technical fora on programming multi-agent systems (see Dastani & Gomez-Sanz, 2005 and Dastani & Gomez-Sanz, 2006 for the general report of these technical fora). In this section, we discuss some proposals for specifying and implementing normative multi-agent systems.

One of the early modelling languages for specifying institutions in terms of institutional rules and norms is ISLANDER proposed by Esteva *et al.* (2002). In order to interpret institution specifications and execute them, an execution platform, called AMELI, has been developed by Esteva *et al.* (2004). This platform implements an infrastructure that, on the one hand, facilitates agent participation within the institutional environment and supports their communication and, on the other, enforces the institutional rules and norms as specified in the institutional specification. The key aspect of ISLANDER/AMELI is that norms can never be violated by the agents. In other words, systems programmed via ISLANDER/AMELI make only use of regimentation in order to guarantee the norms to be actually followed. The norms in Esteva *et al.* (2004), Garcia-Camino *et al.* (2005), Silva (2008) are related to actions that the agents should or should not perform. In these approaches, the issue of expressing more high-level norms concerning a state of the system that should be brought about is ignored. Such high-level norms can be used to represent *what* the agents should establish—in terms of a declarative description of a system state—rather than specifying *how* they should establish it.

Another approach concerning specification of normative multi-agent systems by means of social and organisational concepts is $\mathscr{M}$OISE$^+$, proposed by Hübner *et al.* (2007). This modelling language can be used to specify multi-agent systems through three organisational dimensions: structural (e.g. specifying roles, groups, and links within organisations, subgroup relation, number of agents that can play a role), functional (e.g. goals, missions, and social schemes specifying structured sets of goals), and deontic (e.g. norms, obligations, and prohibitions within organisations). In a series of papers, different computational frameworks have been proposed to implement and execute $\mathscr{M}$OISE$^+$ specifications. Examples of such frameworks are $\mathcal{S}-\mathscr{M}$OISE$^+$ as proposed by Hübner *et al.* (2006) and its artifact-based version ORG4MAS as proposed by Hübner *et al.* (2010). These frameworks are concerned with norms that are about declarative descriptions of a state that should be achieved. Following the $\mathscr{M}$OISE$^+$ specification language, $\mathcal{S}-\mathcal{M}$OISE$^+$ is an organisational middleware that provides agents access to the communication layer and the current state of the specified organisation. Moreover, this middleware allows agents to change the organisation and its specification, as long as such changes do not violate organisational constraints. In the artifact version of this framework, ORG4MAS, various organisational artifacts are used to implement specific components of an organisation such as group and goal schema. In this framework, a special artifact, called reputation artifact, is introduced to manage the enforcement of the norms.

To summarize, in the work on electronic institutions ISLANDER/AMELI norms pertain to low-level procedures that directly refer to actions, whereas $\mathcal{M}$OISE$^+$/$\mathcal{S}-\mathcal{M}$OISE$^+$ are concerned with more high-level norms pertaining to declarative descriptions of the system. However, $\mathcal{S}-\mathcal{M}$OISE$^+$ does not allow agents to violate organisational rules and norms by ensuring that they respect the organisational specification. This suggests that norms in $\mathcal{S}-\mathcal{M}$OISE$^+$ are regimented rather than being enforced by means of sanctions. In the artifact version of this framework, ORG4MAS, the enforcement of norms is assumed to be managed indirectly through a reputation mechanism, but it remains unclear how such a reputation system realises sanctioning. Another important issue is that AMELI and $\mathcal{S}-\mathcal{M}$OISE$^+$ lack a complete operational semantics that capture all aspects of normative systems, including the enforcement of norms. An explicit formal and operational treatment of norm enforcement is essential for a thorough understanding and analysis of computational frameworks of normative multi-agent systems. In addition, the computational frameworks related to $\mathcal{M}$OISE$^+$ are not grounded in a logical system such that the soundness and properties of the programmed systems cannot be analysed through formal analyses and verification mechanisms. Finally, it should be noted that ISLANDER/AMELI and $\mathcal{M}$OISE$^+$/$\mathcal{S}-\mathcal{M}$OISE$^+$ provide a variety of social and organisational concepts.

`powerJava`, as proposed by Baldoni *et al.* (2005), and `powerJade` as proposed by Baldoni *et al.* (2008), are developed to implement institutions in terms of roles. While powerJava extends Java with programming constructs to implement institutions, `powerJade` proposes similar extensions to the Jade framework. In these frameworks, an institution is considered as an exogenous coordination mechanism that manages the interactions between participating computational entities (objects in `powerJava` and agents in `powerJade`) by means of roles. A role is defined in the context of an institution (e.g. a student role is defined in the context of a school) and encapsulates capabilities, also called powers, that its players can use to interact with the institution and with other roles in the institution (e.g. a student can participate in an exam). For an object or an agent to play a role in an institution in order to gain specific abilities, they should satisfy specific requirements as well. In `powerJava` roles and organisations are implemented as Java classes. In particular, a role within an institution is implemented as an inner class of the class that implements the organisation. Moreover, the powers that a player of a role gains and the requirements that the player of the role should satisfy are implemented as methods of the class that implements the role. In `powerJade`, organisations, roles, and players are implemented as subclasses of the Jade agent class. The powers that the player of a role gains and the requirements that a player of a role should satisfy are implemented as Jade behaviours (associated to the role).

Finally, a recent programming language that is developed to support the implementation of multi-agent organisations is 2OPL (Organisation Oriented Programming), as proposed by Dastani *et al.* (2009b) and Tinnemeier *et al.* (2009b). This is a rule-based programming language that facilitates the implementation of norm-based organisations. In this approach, an organisation is considered as a software entity that exogenously coordinates the interaction between agents and their shared environment. In particular, the organisation is a software entity that manages the interaction between the agents themselves and between agents and the shared environment. 2OPL provides programming constructs to specify (1) the initial state of an organisation, (2) the effects of agent's actions in the shared environment, and (3) the applicable norms and sanctions. In 2OPL, norms can be either enforced by means of sanctions or regimented. In the first case, agents are allowed to violate norms after which sanctions are imposed. In the second case, norms are considered as constraints that cannot be violated. The enforcement of norms by sanctions is a way to guarantee higher autonomy for agents and higher flexibility for multi-agent systems. The interpreter of 2OPL is based on a cyclic control process. At each cycle, the observable actions of the individual agents (i.e. communication and environment actions) are monitored, the effects of the actions are determined, and norms and sanction are imposed if necessary. An advantage of 2OPL approach is its complete operational semantics such that normative organisation programs can be formally analysed by means of verification techniques (see e.g. Astefanoaei *et al.*, 2009). This organisation-oriented programming language is extended with programming constructs that support the implementation of concepts such as obligation, permission, prohibition, deadline, norm change, and conditional norm; see Tinnemeier *et al.* (2009a, 2009b, 2010).

4.3 Programming frameworks for multi-agent environments

A framework for the development of multi-agent environments is Cartago (Common artifact infrastructure for agent open environment), which is proposed by Ricci *et al.* (2006). This framework is based on the A&A model that proposes a working environment to be used by agents for supporting their activities. A working environment is considered as consisting of a set of artifacts organised in workspaces (containers of artifacts). The artifacts are meant to encapsulate specific functionalities and can be added, removed, and organised in the workspaces by agents at run time. Artifacts can be used by agents through their usage interfaces that allow agents to trigger and control the execution of artifacts' operations and perceiving events from them. Different operations are supported by artifact interfaces. An agent can, for example, create, remove, or search for artifacts and workspaces. Agents can also execute operations of artifacts, for example, sense the events generated by an artifact or inspect an artifact by retrieving its description. This framework can be distributed in the sense that a working environment can consist of one or more workspaces that can be mapped onto a different nodes of a network. Cartago is implemented in Java and has been connected to various agent executions platforms such as the execution platforms for 2APL and Jason.

Beside this generic architecture and framework for the development of environments, there have been many interesting environments implemented using existing programming languages such as Java or C++. These environments are initially developed in an *ad hoc* manner either for an existing agent platform (e.g. the platforms for 2APL, GOAL, Jadex, and Jason) or as a simulation environment. The availability of these implemented environments raises the question how they can be (re)used and applied to arbitrary agent platforms. In practice, agent developers rebuild similar environments from scratch. Apart from these duplicating works, the interaction between agents and environments are managed in an *ad hoc* manner making the reuse of the environments a dedicated task that depends on the specific agent platform and the environment at hand. This problem has lead to an initiative for creating a generic environment interface that provides the required functionalities for connecting agents to environments (see e.g. Behrens *et al.*, 2010). This initiative wants to become a *de facto* standard. If environments were developed using such a standard, they could be exchanged freely between agent platforms that support the standard and thus would make already existing environments widely available. In order to develop a generic environment interface standard various issues should be addressed. An important issue is the right level of abstraction for modelling the interaction between agents and environments. This generic environment interface standard supports the interaction between agents and environment in two ways. On the one hand, agents can perform actions, including sense action, in the environment (the environment is assumed to realise the effect of the actions).

On the other hand, the environment can send events to individual agents. This interface provides constructs to establish and manage the relation between agents with entities (agent bodies) in the environment, the registration of agents by the interface, adding and removing entities from the environment, and performing actions and retrieving percepts from the environment. Several agent platforms such as 2APL, GOAL, and Jason have already integrated the environment interface standard.

5 Current trends

Existing multi-agent programming languages and development frameworks are the result of continuous developments. Despite their characteristic differences, these developments and extensions have been quite similar causing the programming languages to converge in the sense of providing programming constructs for the same set of concepts and abstractions. For example, 2APL, GOAL, Jason, and Jadex provide similar types of actions such as actions to modify an agent's state, communication actions, and external actions allowing individual agents to interact with a shared environment. In the logic-based programming languages for BDI architectures such as 2APL, GOAL, Jason, and KGP, an agent's beliefs and goals are often programmed declaratively (e.g. in Prolog) allowing the programmed agent to reason with its beliefs and goals. Existing agent programming languages and development frameworks such as 2APL, GOAL, Jason, Jade, JACK, and Jadex provide constructs to process various types of events by means of generating and executing plans. In order to respect programming principles such as reuse and encapsulation, agent programming languages such as 2APL, GOAL, and JACK provide constructs to support implementation of modules. The similarity between these languages is not only due to similar programming constructs, the underlying semantics of these languages converge as well. For example, programming languages with declarative beliefs and goals (e.g. 2APL, GOAL, and KGP) establish rational constraints in their underlying semantics by, for example, requiring that agents should have consistent beliefs and that agents cannot aim at achieving goals that are believed to be achieved. Finally, the execution platforms corresponding to agent programming languages such as 2APL, GOAL, Jason, and Jadex also converge in the sense that they provide similar functionalities and development tools. Most multi-agent development framework such as 2APL, Jason, GOAL, and Jadex provide editors that support the syntax of their corresponding programming languages, different tools to monitor and control the execution of agents, and different services such as agent management and directory facilitator. In the following, we focus on two main ongoing trends in the development of multi-agent programming languages and development frameworks.

5.1 Goal types

An advance in the field of agent programming languages concerns the concept of goals. Goals are essential for agents with pro-active behaviour (see e.g. Wooldridge, 2009). The initial focus of agent-oriented programming languages was on achievement goals, which represent a desired state that the agent aims at achieving. In due course other goal types have been studied by Braubach and Pokahr (2009), Dastani *et al.* (2006), Duff *et al.* (2006), Hindriks and van Riemsdijk (2008). Examples of goal types are perform goal (the goal to execute certain actions), test goal (the goal to test agent's state), and maintain goal (the goal to maintain a state). In order to allow the implementation of various goal types existing agent programming languages provide a variety of constructs to represent and reason with the goal types. For example, JACK, as presented by Winikoff (2005), provides programming constructs to implement, among others, test, achieve, insist, and maintain goals. In addition, Jadex, as presented by Pokahr *et al.* (2005), covers achieve, query, perform, and maintain goals. The way in which goals are treated by these programming languages differs. In Jadex goals are represented in XML in terms of a label/name and a number of other parameters, while JACK goals are particular types of events. Moreover, neither JACK nor Jadex provide the formal semantics of their goal types. Winikoff *et al.* (2002) provides a survey of existing literature on goal types. A more recent (theoretical) trend in this direction is to go beyond these goal types and to introduce more expressive goal types or even a language for expressing goal types. For example, Dastani *et al.* (2011) proposes six types of multiple state goals (goals expressing a property that should hold over a number of states), while other approaches propose to take arbitrary linear temporal logic (LTL) formulae as goals (see e.g. Bacchus & Kabanza, 1998; Baral & Zhao, 2007; Shapiro & Brewka, 2007; Hindriks *et al.*, 2009; Khan & Lespoerance, 2009). The advantage of the approach proposed by Dastani *et al.* is their computational setting where the six multiple state goal types are defined in terms of achieve and maintain goals. This makes it possible to implement these goal types in the agent programming frameworks that already have an operationalisation of achieve and maintain goals.

5.2 Modular programming

From the software development point of view, the ultimate aim of multi-agent programming languages and development frameworks is to support practitioners to develop multi-agent systems for industrial applications. To this aim it is important that programming languages and development frameworks satisfy essential principles in structured programming such as modularity. Of course, the separation of concerns at the level of individual agents, organisation, and environment support modularity in multi-agent programming. However, programming languages and development frameworks for agents, organisations, and environments need to satisfy modularity as well. There have been some proposals for supporting modules in BDI-based programming languages. Examples of these proposals are Braubach *et al.* (2005), Busetta *et al.* (2000), Hindriks (2008), van Riemsdijk *et al.* (2006), Dastani and Steunebrink (2010), and Madden and Logan (2009). In these proposals, modularisation is considered as a mechanism to structure an individual agent's program in separate modules, each encapsulating cognitive components such as beliefs, goals, and plans that together model a specific functionality and can be used to handle specific situations or tasks. However, the way the modules are used in these programming approaches are different. For example, in JACK (Busetta *et al.*, 2000) and Jadex (Braubach *et al.*, 2005), modules (which are also called capabilities) are used for information hiding and reusability by encapsulating different cognitive components that together implement a specific capability/ functionality of the agent. In these approaches, the encapsulated components are used during an agent's execution to process events that are received by the agent. In other approaches (e.g. van Riemsdijk *et al.*, 2006; Hindriks, 2008), modules are used to realise a specific policy or mechanism in order to control an agent execution. More specifically, in GOAL, as proposed by Hindriks (2008), modules are considered as the 'focus of execution', which can be used to disambiguate the application and execution of plans. This is done by assigning a mental state condition (beliefs and/or goals) to each module. The modules whose conditions are satisfied form the focus of an agent's execution such that only plans from these modules are applied and executed. In 3APL, as proposed by van Riemsdijk *et al.* (2006), a module is a set of planning rules that is associated with a specific goal indicating which planning rules can be applied to achieve the goal. In other words, a module implements specific means for achieving specific goals. In 2APL, as proposed by Dastani and Steunebrink (2010), modules are introduced for encapsulation of different cognitive components that together implement a specific agent functionality. The significant difference with other approaches is that a programmer can perform a wide range of operations on modules. These module-related operations enable a programmer to directly and explicitly control when and how modules are used. For instance, a programmer can create an instance of the module specification, query and update its internals, and execute the updated module instance. An agent that executes a module instance, stops deliberating on its current cognitive state and starts deliberating on a new cognitive state that is encapsulated by the executed module instance. The proposed notion of module can be used to implement a variety of agent concepts such as agent role and agent profile. Recently, a modularisation idea for Jason is proposed by Madden and Logan (2009). In this proposal, a module encapsulates a subset of an agent's functionalities and consists of cognitive ingredients such as belief, goal, and event bases, a plan library, and a list of exported belief and goal predicates. An agent is then defined as a composition of modules (modules cannot be nested), together with a slightly modified version of the Jason's original interpreter. Finally, it should be noted that the concept of module as used by Novak and Dix (2006) is different from other approaches. In this approach, a module is considered as one specific cognitive component (e.g. an agent's beliefs) and not as a functionality modelled by different cognitive components. Note also that behaviours in Jade, which can be used to implement an agent's functionality, can also be seen as a kind of module.

6 Current challenges

There are many challenges to meet in the multi-agent programming research field. Examples of these challenges are scalability and automatic code generation from system specifications. In this overview, we focus on two challenges that require both theoretical and practical investigations. The first challenge is a principle integration of programming languages for individual agents, organisations, and environments and the second challenge is the debugging and testing of multi-agent programs.

6.1 Integration of programming languages

Respecting the separation of concerns principle advocates separate programming languages and development frameworks for the implementation of individual agents, environments, and organisations.

The development of multi-agent systems therefore requires a systematic integration of the corresponding programming languages and development frameworks. Ideally, one should be able to write programs for different components of a multi-agent system separately and integrate these programs either by means of another program that indicates how the programs of different components should interact and executed, or through a platform that facilitates an integrated execution of all involved programs. It should be noted that this challenge is only relevant when different components of multi-agent systems need to be programmed separately using dedicated programming languages and development frameworks.

The first proposal for integrating programming languages for various components of multi-agent systems is based on the integration of 2APL and 2OPL, as mentioned by Tinnemeier (2011) and realised by Adal (2010). In this approach, a multi-agent program is implemented by specifying a number of agents programmed in 2APL, one or more environments programmed in Java, and an organisation programmed in 2OPL. An execution of such a multi-agent program is a concurrent execution of the specified individual agents programs, the environment program, and the organisation program. The execution of individual agents programmed in 2APL may cause agents to interact with each other and with the environment. The resulting actions are not directly effectuated in the environment, but passed to the organisation implemented in 2OPL. The organisation decides the effects of those actions in the environment based on the specified organisational norms and sanctions. In particular, the performance of actions in the environment by individual agents is effectuated by the organisation program, which allows/disallows actions and realises the effect of actions in the environment. The organisation will also evaluate the updated state of the environment with respect to the specified organisational norms and impose sanctions when violations are detected. Imposing sanctions is seen as specific updates of the environment state according to the sanctions specified by the organisation. This integration approach views an organisation as an exogenous coordination mechanism that controls and regulates the interaction among agents as well as between agents and environment.

Another integration proposal is JaCaMo as proposed by Boissier *et al.* (2013). This approach aims at integrating Jason, Cartago, and MOISE to program individual agents, environments, and organisations, respectively. The idea is to integrate on the one hand MOISE and Cartago, and on the other Jason and Cartago. The integration of MOISE and Cartago is by means of organisational artifacts and based on the earlier work on ORA4MAS as presented by Kitio *et al.* (2008). The integration of Jason and Cartago is through artifact operations performed by the Jason agents. In this integrated approach, the organisational infrastructure of a multi-agent system consists of organisational artifacts and organisational agents that together are responsible for functionalities concerning the management and enacting of the organisation. Organisational agents manage organisational activities such as observing and reasoning about organisation dynamics. The violation of norms is detected by organisational artifacts after which organisational agents have to deal with those violations. The organisational artifacts and agents are intended to implement norm regimentation and enforcement by means of sanctions, as originally proposed by Dastani *et al.* (2009b). A characteristic of this integration is that the management of organisational activities such as norm enforcement is the responsibility of the so-called organisational agents. It is, however, not clear why such activities should be performed by an agent rather than, for example, by the organisation itself. After all, an agent has by definition its own objectives which is used to motivate its actions and which may not be compatible or even in conflict with the organisation objectives.

6.2 Debugging and testing multi-agent programs

Debugging is the art of finding and resolving errors or possible defects in a computer program. In general, there are various types of bugs such as syntax bugs, semantic bugs (logical and concurrent bugs), or design bugs. Design bugs arise before the actual programming and are based on erroneous design of software programs (see e.g. Dastani *et al.*, 2009a). In contrast to design bugs, both syntax and semantic bugs arise during programming and are related to the actual code of the programs. Although syntax bugs depend on the specification of programming languages and are (most of the time) simple typos, which can easily be detected by the program parser (compiler), semantic bugs are mistakes at the semantic level. Because they often depend on the intention of the developer they can rarely be detected automatically by the program parsers. Therefore, special tools are needed to detect semantic bugs. The ease of the debugging experience is largely dependent on the quality of these debugging tools and the ability of developers to work with these tools.

The main challenge with respect to debugging multi-agent programs are the semantic bugs caused by the execution of autonomous agent programs, and those caused by the interaction between agents, environments, and organisations. The bugs causing by the interaction between agents are often dealt with by means of different types of visualisation tools such as sniffer and causality graphs (see e.g. Bellifemine *et al.*, 2005; Pokahr *et al.*, 2005; Botia *et al.*, 2006; Bordini *et al.*, 2007; Dastani, 2008; Vigueras & Botia, 2008). The visualisation tools allow the developer to browse through exchanged messages, inspect the messages, and present them using different visualisation techniques. Debugging the interaction between agents, environments, and organisations are still an unexplored research area. The semantic bugs caused by the execution of individual agent programs are dealt with by a variety of techniques such as breakpoints, assertions, and execution tracers (see e.g. Bellifemine *et al.*, 2005; Lam & Suzanne Barber, 2005; Bordini *et al.*, 2007; Collier, 2007; Sudeikat *et al.*, 2007; Dastani, 2008). A breakpoint is a marker that can be placed in the program's code to control the program's execution. When the marker is reached the program execution is halted. Assertions are statements that can be annotated to specific elements of the programming language. When an assertion is evaluated to false, a warning is generated to inform the developer about the agent and the element where the assertion is evaluated to false. The execution tracer is a standard tool that is present in most multi-agent development frameworks. This is a window that enables a developer to view, inspect, and trace an agent's internal state, and to start, stop, and step through the execution of the agent program.

Debugging agent programs that are based on BDI abstractions requires additional tools to monitor and control temporal and cognitive properties of the agent program executions. In Dastani *et al.* (2009a), a debugging approach is proposed which is based on a specification language to express temporal and cognitive execution properties of multi-agent programs and a set of debugging tools. The expressions of the specification language are related to the proposed debugging tools such that the debugging tools are activated as soon as their associated properties hold for the multi-agent program execution thus far. The specification language is based on LTL extended with the BDI operators. Given an execution of a multi-agent program, one can check if an agent drops a specific goal when it is achieved, when two or more agents have the same beliefs, whether a protocol is suited for a given task, whether important beliefs are communicated and if they are adopted/rejected once they are communicated.

Recent developments in multi-agent programming languages have proposed specific programming constructs enabling the implementation of social concepts such as norms, roles, obligations, and sanctions (see e.g. Esteva *et al.*, 2004; Hiibner *et al.*, 2006; Dastani *et al.*, 2009b; Tinnemeier *et al.*, 2009b). Debugging such multi-agent programs requires specific debugging constructs to specify properties related to the social aspects and to find and resolve defects involved in such programs. The presented debugging approaches assume all agents are developed on one single platform such that their executions for debugging purposes are not distributed on different platforms. One important challenge and a future work on debugging multi-agent systems remains the debugging of multi-agent programs that run concurrently on different platforms. It is, for example, not clear how the distributed executions of a multi-agent program can be monitored and controlled or how the distributed program codes can be accessed and modified.

The techniques mentioned above are helpful when errors manifest themselves directly to the developers or users. However, errors in a program do not always manifest themselves directly. For industrial applications it may be necessary to extensively test programs before deploying them. The testing should remove as many bugs (and possible defects) as possible. However, it is often infeasible to test every single situation in which the program could run. A testing approach for multi-agent programs is proposed by Poutakidis *et al.* (2002, 2003) and Winikoff (2010). Testing is an indispensable part of evaluating multi-agent programs and should therefore be integrated in the existing debugging approaches. This allows the generation of a set of critical test traces which will be the subject of debugging in *post mortem* mode. Finally, we would like to emphasise that for mission and industrial critical systems, it is often necessary to formally verify the programs by means of theorem proving or model checking techniques. Unfortunately, these verification techniques are computationally expensive and cannot be applied to large-scale software systems. Some existing work on formal verification of multi-agent systems are presented in Dastani *et al.* (2010).

6.3 Towards industry-strength technology

The field of multi-agent programming can be approached from two different perspectives: academic and industry. From the academic perspective, this research field contributes to the integration of theories and techniques from artificial intelligence in the development and implementation of intelligent systems. In particular, it provides computational and executable models for high-level (social and cognitive) concepts, and proposes corresponding programming constructs that can be used to develop and implement intelligent systems. For example, computational models for beliefs and goals with their corresponding programming constructs are used to develop and implement proactive systems, and event processing mechanisms are used to develop and implement reactive systems. Moreover, the provided models and programming languages are extended and used to build prototypes of intelligent systems. For example, programming languages for BDI agents are extended with emotion models (e.g. Dastani & Meyer, 2010) to prototype emotional agents, and multi-agent programming languages are extended with social concepts to prototype or simulate agent societies. The academic perspective on this research field has shown to be attractive to the researchers from the logic community. In particular, researchers from computational logics and knowledge representation and reasoning have proposed formal and computational models that can be used in multi-agent programming languages and their interpreters to allow the representation and reasoning with an agent's (cognitive) state. In addition, researchers with specification and verification background have proposed various frameworks to specify and verify properties of individual agents and multi-agent programs.

Although the research in the multi-agent programming community has been dominated by the academic perspective and attracted more academic researchers than industrial practitioners, an important aim of this community has been the development of industry strength tools and technologies to build industrial applications. There are some unverified explanations for why the existing tools and technologies from multi-agent programming research community have not been adopted and deployed for industrial applications. For example, it is often claimed that the industry requires tools and technologies with predictable behaviours, while agents and multi-agent systems are generally seen by the industry as intelligent and adaptive systems with unpredictable behaviours. Moreover, the developers from the industry seem to prefer having full control over the execution of the developed software systems, while most of the existing multi-agent programming languages are based on interpreters that make complex decisions (e.g., deliberation and control processes) on behalf of the software developers. It is also claimed that the industry tends to use standard software technologies and is not eager to adopt paradigmatic change in software technologies and methodologies.

The adoption of multi-agent programming tools and technologies by the industry is a major challenge that still needs to be met by the multi-agent programming community. One possible way to meet this challenge is by transferring multi-agent programming tools and technologies to the standard software technologies. An idea is to start with the high-level concepts and abstractions for which multi-agent programming research field have provided computational models and programming constructs, and propose either corresponding language-level supports in the standard programming languages (e.g. C++ or Java), or alternatively propose corresponding design patterns, that is, general reusable solutions to problems such as proactivity, reactivity, adaptivity, control, and monitoring. The language-level support can either be realised by standard programming approaches such as meta-programming or aspect-oriented programming where concepts such as deliberation and control can be considered as different concerns that can be programmed either by meta-programs or aspects. Although these suggestions are not mature and need to be work out both in details and in practice, attempts along these lines can bring multi-agent community closer to the industry.

7 Conclusion

The maturation of multi-agent programming languages and development frameworks is still a main issue in the multi-agent programming community. From the software development perspective, one of the objectives of multi-agent programming community is to propose programming languages that support direct and effective implementations of large-scale multi-agent systems. This is realised by proposing programming constructs to facilitate the implementation of concepts and abstractions used in the analysis and design of multi-agent systems. Up until now many multi-agent programming languages and development frameworks have been proposed. They differ from each other in the set of abstractions, programming constructs, and principles they convey. Although these programming languages and development frameworks are evolving towards a certain level of maturity in the sense that their programming concepts and operations are well motivated and have profound semantics, a majority of them are still not being used in the development of large-scale industrial applications.

Currently, multi-agent programming languages and development frameworks, in particular those that are based on social and cognitive constructs, are mainly considered as research works. The incorporation of social and cognitive concepts in multi-agent programming languages and development frameworks requires their semantic and computational analyses. Therefore, the multi-agent programming research field has been attractive to researchers from various scientific disciplines such as logic, artificial intelligence, philosophy, cognitive science, and social science.

From this interdisciplinary perspective, multi-agent programming languages and development frameworks are primarily seen as computational architectures of multi-agent systems and are not designed to satisfy or support (functional or non-functional) requirements for engineering large-scale industrial applications. The maturation of multi-agent programming languages and development frameworks regarding such requirements remains a future research direction in multi-agent programming community (see Cysneiros & Yu, 2003; Silva *et al.*, 2003 for a discussion on multi-agent system requirements).

The state of the art in the field of multi-agent programming shows the emergence of specialised programming languages and development frameworks for individual agents, organisations, and environments. The main focus of multi-agent programming community has been on the development of programming languages and development frameworks for individual agents. Although the research on (formal) models of multi-agent organisations and environments has a long history, the emergence of programming languages and development frameworks for supporting their implementations is a recent phenomenon. The evolution of programming languages and development frameworks for individual agents show a convergence in the sense that they propose programming constructs for an established set of concepts and abstractions. These languages and frameworks differ from each other as they use different programming styles (declarative, imperative, or both), support different programming principles such as modularity, abstraction, recursion, exception handling, support for legacy code, and as their corresponding integrated development environments provide different sets of functionalities such as editing, debugging, and automatic generation of codes.

One of the current challenges in the multi-agent programming research field is the integration of programming languages and development frameworks for individual agents, multi-agent organisations, and multi-agent environments. Although there have been several attempts to integrate specific programming languages, the ultimate goal is a mechanism to facilitate the integration of arbitrary programming languages and development frameworks for individual agents, multi-agent organisations, and multi-agent environments. A possible solution to realise this goal is to develop standard interfaces that can manage the interactions between individual agent programs, multi-agent organisation programs, and multi-agent environment programs. There have already been some initiatives to establish standard interfaces for managing the interaction of individual agent programs and environment programs, but the research in this direction is still in a preliminary phase and needs support and collaboration from the community. Another issue currently challenging the multi-agent programming community is the debugging and testing of multi-agent programs. There is a need for powerful debugging facilities and testing tools that can cope with the distributed nature of multi-agent systems, the autonomy of individual agents, and the interactions between individual agent, multi-agent organisation, and multi-agent environment programs. There have been some initial attempts for enriching debugging tools with expressive specification languages such that tools can be initialised and activated when the execution of multi-agent programs satisfy certain properties, but such attempts ignore multi-agent organisation and environment programs.

As noticed before, this overview is by no means complete. There are still many issues related to multi-agent programming that need to be investigated. Among these issues are mechanisms to deal with plan failure, goal types, reasoning about organisations and environments from an agent's point of view, the integration of concepts such as sensing, planning, acting, learning, and emotions in the agent's deliberation process, the adaptivity of organisation and environment based on the executions of individual agent programs, and formal verification of multi-agent programs.

Acknowledgement

The author would like to offer special thanks to anonymous reviewers for their valuable and constructive comments. They have helped to improve the quality of this paper.

References

Adal, A. 2010. *An Interpreter for Organization Oriented Programming Language*. Master's thesis, Utrecht University.

Ågotnes, T., van der Hoek, W. & Wooldridge, M. 2008. Robust normative systems. In *Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, L., Parkes, D., Mueller, J. P. & Parsons, S. (eds), 747–754, May. IFAMAAS/ACM DL.

Arbab, F. 1998. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, 11–22.

Arbab, F., Astefanoaei, L., de Boer, F., Dastani, M., Meyer, J.-J. C. & Tinnermeier, N. 2009. Reo connectors as coordination artifacts in 2APL systems. In *Proceedings of the 11th Pacific Rim International Conference on Multi-Agents (PRIMA 2008)*, LNCS, **5357**, 42–53. Springer.

Astefanoaei, L., Dastani, M., Meyer, J.-J. C. & Boer, F. 2009. On the semantics and verification of normative multi-agent systems. *International Journal of Universal Computer Science* **15**(13), 2629–2652.

Bacchus, F. & Kabanza, F. 1998. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence* **22**(1–2), 5–27.

Baldoni, M., Boella, G., Dorni, M., Grenna, R. & Mugnaini, A. 2008. powerJADE: organizations and roles as primitives in the JADE framework. In *WOA 2008: Dagli oggetti agli agenti, Evoluzione dell'agent development: metodologie, tool, piattaforme e linguaggi*, 84–92.

Baldoni, M., Boella, G. & Van Der Torre, L. 2005. Roles as a coordination construct: introducing powerJava. In *Proceedings of the 1st International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems*, Electronic Notes in Theoretical Computer Science **150**, 9–29.

Baral, C. & Zhao, J. 2007. Non-monotonic temporal logics for goal specification. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 236–242.

Behrens, T., Dix, J., Hindriks, K., Dastani, M., Bordini, R., Hubner, J., Pokahr, A. & Braubach., L. 2010. An interface for agent-environment interaction. In *Proceedings of the Eighth International Workshop on Programming Multi-Agent Systems (ProMAS'10)*, Bordini, R.H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds), 125–147. Springer.

Bellifemine, F., Bergenti, F., Caire, G. & Poggi, A. 2005. JADE—a Java agent development framework. In *Multi-Agent Programming: Languages, Platforms and Applications.* Kluwer.

Bergenti, F., Gleizes, M.-P. & Zambonelli, F. (eds) 2004. *Methodologies and Software Engineering for Agent Systems*. Multiagent Systems, Artificial Societies, and Simulated Organizations **11**, Kluwer Academic Publisher.

Boella, G. & van der Torre, L. 2008. Substantive and procedural norms in normative multiagent systems. *Journal of Applied Logic* **6**, 152–171.

Boissier, O., Bordini, R., Hbner, J. F., Ricci, A. & Santi, A. 2013. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* **78**, 747–761.

Bordini, R., Huubner, J. & Wooldridge, M. 2007. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley Series in Agent Technology, John Wiley & Sons.

Bordini, R. H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds) 2005. *Multi-Agent Programming: Languages, Platforms and Applications*. Multiagent Systems, Artificial Societies, and Simulated Organizations **15**. Springer, ISBN: 978-0-387-24568-3.

Bordini, R. H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds) 2009. *Multi-Agent Programming: Languages, Tools and Applications*, Springer, ISBN: 978-0-387-89298-6.

Botia, J. A., Hernansaez, J. M. & Gomez-Skarmeta, A. F. 2006. On the application of clustering techniques to support debugging large-scale multi-agent systems. In *Proceedings of the Fourth International Workshop on Programming Multi-Agent Systems (ProMAS'06)*, 217–227.

Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Mancarella, P., Sadri, F., Stathis, K., Terreni, G. & Toni, F. 2004. The KGP model of agency for global computing: computational model and prototype implementation. In *Global Computing*, LNCS, **3267**, 340–367. Springer.

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

Braubach, L. & Pokahr, A. 2009. Representing long-term and interest BDI goals. In *Proceedings of the Seventh International Workshop on Programming Multi-Agent Systems (ProMAS'09).*

Braubach, L., Pokahr, A. & Lamersdorf, W. 2005. Extending the capability concept for flexible BDI agent modularization. In *Proceedings of the Third International Workshop on Programming Multi-Agent Systems (ProMAS'05)*, 139–155.

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. & Perini, A. 2003. TROPOS: an agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems* **8**(3), 203–236.

Busetta, P., Howden, N., Ronnquist, R. & Hodgson, A. 2000. Structuring BDI agents in functional clusters. In *Intelligent Agents VI: Theories, Architectures and Languages*, Jennings N. & Lesperance Y. (eds), Lecture Notes in Computer Science, **1757**, 277–289. Springer.

Cohen, P. R. & Levesque, H. J. 1990. Intention is choice with commitment. *Artificial Intelligence* **42**(2–3), 213–261.

Collier, R. 2007. Debugging agents in agent factory. In *Proceedings of the Fourth International Workshop on Programming Multi-Agent Systems (ProMAS'06)*, 229–248.

Cysneiros, L. M. & Yu, E. S. K. 2003. Requirements engineering for large-scale multi-agent systems. In *Software Engineering for Large-Scale Multi-Agent Systems, Research Issues and Practical Applications (SELMAS)*, Garcia, A. F., de Lucena, C. J. P., Zambonelli, F., Omicini, A. & Castro, J. (eds), LNCS, **2603**, 39–56. Springer.

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

Dastani, M., Arbab, F. & de Boer, F. S. 2005a. Coordination and composition in multi-agnet systems. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, 439–446.

Dastani, M., Brandsema, J., Dubel, A. & Meyer, J.-J. C. 2009a. Debugging BDI-based multi-agent programs. In *Proceedings of the Seventh International Workshop on Programming Multi-Agent Systems (ProMAS'09)*, LNCS, **5919**, 151–169.

Dastani, M., Grossi, D., Meyer, J.-J. C. & Tinnemeier, N. 2009b. Normative multi-agent programs and their logics. In *Post Proceedings of the International Workshop on Knowledge Representation for Agents and Multi-Agent Systems (KRAMAS'08)*, LNAI, **5605**, 16–31. Springer.

Dastani, M. & Gomez-Sanz, J. 2006. Programming multi-agent systems. *The Knowledge Engineering Review* **20**(2), 151–164.

Dastani, M. & Gomez-Sanz, J. J. 2005. AgentLink III Technical Forum Group, programming multiagent systems. `http://people.cs.uu.nl/mehdi/al3promas.html`.

Dastani, M., Hindriks, K. & Meyer, J.-J. C. 2010. *Specification and Verification of Multi-Agent Systems*, Springer, ISBN 978-1-4419-6983-5.

Dastani, M. & Meyer, J.-J. C. 2010. Agents with emotions. *International Journal of Intelligent Systems* **25**(7), 636–654.

Dastani, M. & Steunebrink, B. R. 2010. Operational semantics for BDI modules in multi-agent programming. In *Proceedings of the 10th International Conference on Computational Logic in Multi-Agent Systems (CLIMA'09)*, 83–101. Springer-Verlag.

Dastani, M., van Riemsdijk, M. B. & Meyer, J.-J. C. 2005b. Programming multi-agent systems in 3APL. In *Multi-Agent Programming: Languages, Platforms and Applications*, Bordini, R.H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds), 39–67. Kluwer.

Dastani, M., van Riemsdijk, M. B. & Meyer, J.-J. C. 2006. Goal types in agent programming. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06).*

Dastani, M., van Riemsdijk, B. & Winikoff, M. 2011. Rich goal types in agent programming. In *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011).*

De Giacomo, G., Lesperance, Y. & Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* **121**(1–2), 109–169.

Dignum, V. 2004. *A Model for Organizational Interaction*. PhD thesis, Utrecht University, SIKS.

Dix, J. & Zhang, Y. 2005. IMPACT: a multi-agent framework with declarative semantics. In *Multi-Agent Programming: Languages, Platforms and Applications*, Bordini, R.H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds), 69–94. Kluwer.

Duff, S., Harland, J. & Thangarajah, J. 2006. On proactivity and maintenance goals. In *Proceedings of the Fifth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 06),* 1033–1040.

El Fallah Seghrouchni, A. & Suna, A. 2005. CLAIM and SyMPA: a programming environment for intelligent and mobile agents. In *Multi-Agent Programming: Languages, Platforms and Applications*, Bordini, R.H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds), 95–122, Kluwer.

Esteva, M., de la Cruz, D. & Sierra, C. 2002. ISLANDER: an electronic institutions editor. In *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2002)*, 1045–1052.

Esteva, M., Rodriguez-Aguilar, J. A., Rosell, B. & Arcos, J. L. 2004. AMELI: an agent-based middleware for electronic institutions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, 236–243, July.

Ferber, J. 1999. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, Addison-Wesley Longman Publishing, ISBN: 0201360489.

Fisher, M. 2005. METATEM: the story so far. In *Proceedings of the First International Workshop on Programming Multi-Agent Systems (ProMAS'03)*, LNAI, **3862**, 3–22. Springer Verlag.

Garcia-Camino, A., Noriega, P. & Rodriguez-Aguilar, J. A. 2005. Implementing norms in electronic institutions. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 05)*, 667–673.

Gomez-Sanz, J. & Pavon, J. 2003. Agent oriented software engineering with INGENIAS. In LNCS, **2691**, 394–403. Springer.

Grossi, D. 2007. *Designing Invisible Handcuffs*. PhD thesis, Utrecht University, SIKS.

Hindriks, K. 2008. Modules as policy-based intentions: modular agent programming in GOAL. In *Proceedings of the Fifth International Workshop on Programming Multi-Agent Systems (ProMAS'07)*, **4908**. Springer.

Hindriks, K. 2009. Programming rational agents in GOAL. In *Multi-Agent Programming: Languages and Tools and Applications*, Bordini, R.H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds), 119–157. Springer.

Hindriks, K., De Boer, F., 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., van der Hoek, W. & van Riemsdijk, M. B. 2009. Agent programming with temporally extended goals. In *Proceedings of the Eight International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*, 137–144. IFAAMAS.

Hindriks, K. & van Riemsdijk, M. B. 2008. Satisfying maintenance goals. In *Declarative Agent Languages and Technologies (DALT 07)*, LNAI, **4897**, 86–103. Springer.

Hübner, J., Sichman, J. S. & Boissier, O. 2006. S-MOISE+: a middleware for developing organised multi-agent systems. In *Proceedings of the International Workshop on Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems*, LNCS, **3913**, 64–78. Springer.

Hübner, J., Sichman, J. S. & Boissier, O. 2007. Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering* **1**(3/4), 370–395.

Hübner, J. F., Boissier, O., Kitio, R. & Ricci, A. 2010. Instrumenting multi-agent organisations with organisational artifacts and agents: giving the organisational power back to the agents. *International Journal of Autonomous Agents and Multi-Agent Systems* **20**, 369–400.

Jones, A. J. I. & Sergot, M. 1993. On the characterization of law and computer systems. In *Deontic Logic in Computer Science: Normative System Specification*, Meyer J.-J. C. & Wieringa R. J. (eds). John Wiley & Sons, 275–307.

Kakas, A., Mancarella, P., Sadri, F., Stathis, K. & Toni, F. 2004. The KGP model of agency. In *The 16th European Conference on Artificial Intelligence (ECAI'04)*, 33–37.

Khan, S. M. & Lespoerance, Y. 2009. A logical account of prioritized goals and their dynamics. In *Proceedings of the Ninth International Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense-09)*, Lakemeyer G., Morgenstern L. & Williams M. A. (eds). Open Publications of UTS Scholars, 85–90.

Kitio, R., Boissier, O., Hubner, J. & Ricci, A. 2008. Organisational artifacts and agents for open multi-agent organisations: 'giving the power back to the agents'. In *Proceedings of the 2007 International Conference on Coordination, Organizations, Institutions, and Norms in Agent Systems III, COIN'07*, 171–186. Springer-Verlag. ISBN: 3-540-79002-0, 978-3-540-79002-0.

Lam, D. N. & Suzanne Barber, K. 2005. Debugging agent behavior in an implemented agent system. In *Proceedings of the Second International Workshop on Programming Multi-agent Systems (ProMAS'04)*, 104–125.

Leite, J., Alferes, J. & Pereira, L. M. 2001. Minerva—a dynamic logic programming agent architecture. In the proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001). Meyer, J.-J. C. & Tambe, M. (eds), LNAI **2333**, 141–157, Springer.

Madden, N. & Logan, B. 2009. Modularity and compositionality in Jason. In *Proceedings of the Seventh International Workshop on Programming Multi-Agent Systems (ProMAS'09)*, Braubach, L., Briot, J.-P. & Thangarajah, J. (eds), LNAI, **5919**, 237–253. Springer. ISBN: 978-3-642-14842-2.

Meyer, J.-J. C., van der Hoek, W. & van Linder, B. 1999. A logical approach to the dynamics of commitments. *Arificial Intelligence* **113**, 1–40.

Miiller, J. P. 1996. *The Design of Autonomous Agents A Layered Approach*, LNAI, **1177**. Springer-Verlag.

Muldoon, C., O'Hare, G. M. P., Collier, R. W. & O'Grady, M. J. 2009. Towards pervasive intelligence: reflections on the evolution of the agent factory framework. In *Multi-Agent Programming: Languages and Tools and Applications*, Bordini R. H., Dastani M., Dix J. & El Fallah Seghrouchni A. (eds). Springer, 187–212.

Nair, R. & Tambe, M. 2005. Hybrid BDI-POMDP framework for multiagent teaming. *Journal of Artificial Intelligence Research* **23**(1), 367–420.

Novaik, P. & Dix, J. 2006. Modular BDI architecture. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06).*

Omicini, A. 2007. Formal ReSpecT in the A&A perspective. *Electronic Notes Theoretical Computer Science* **175**(2), 97–117.

Padgham, L. & Winikoff, M. 2003. Prometheus: a methodology for developing intelligent agents. In Lecture Notes in Artificial Intelligence, **2585**, 174–185. Springer.

Pokahr, A., Braubach, L. & Lamersdorf, W. 2005. Jadex: a BDI reasoning engine. In *Multi-Agent Programming: Languages, Platforms and Applications*, Bordini, R.H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds), 149–174. Kluwer.

Poutakidis, D., Padgham, L. & Winikoff, M. 2002. Debugging multi-agent systems using design artifacts: the case of interaction protocols. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, 960–967.

Poutakidis, D., Padgham, L. & Winikoff, M. 2003. An exploration of bugs and debugging in multi-agent systems. In *Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, 628–632. ACM Press.

Prakken, H. & Sergot, M. 1996. Contrary-to-duty obligations. *Studia Logica* **57**, 91–115.

Rao, A. S. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, Van de Velde, W. & Perram, John W. (eds). Agents Breaking Away, LNCS 1038, 42–55. Springer.

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

Rao, A. S. & Georgeff, M. P. 1995. BDI agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS 95)*, Allen, J., Fikes, R. & Sandewall, E. (eds), 312–319. MIT Press.

Rens, G., Ferrein, A. & van der Poel, E. 2009. A BDI agent architecture for a POMDP planner. In *Nineth International Symposium on Logical Formalizations of Commonsense Reasoning*.

Ricci, A., Viroli, M. & Omicini, A. 2006. Cartago: a framework for prototyping artifact-based environments in MAS. In *E4MAS*, 67–86.

Sadri, F. 2005. Using the KGP model of agency to design applications. In *Proceedings of the 6th International Conference on Computational Logic in Multi-Agent Systems (CLIMA 05)*, **3900**, 165–185. Springer.

Sardina, S., De Giacomo, G., Lesperance, Y. & Levesque, H. J. 2004. On the semantics of deliberation in IndiGolog—from theory to implementation. *Annals of Mathematics and Artificial Intelligence* **41**(2–4), 259–299.

Shapiro, S. & Brewka, G. 2007. Dynamic interactions between goals and beliefs. In *International Joint Conference on Artificial Intelligence (IJCAI'07)*, 2625–2630.

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

Silva, C. T. L. L., Castro, J. & Tedesco, P. A. 2003. Requirements for multi-agent systems. In *Workshop em Engenharia de Requisitos (WER)*, Galvao Martins, L. E. & Franch, X. (eds), 198–212.

Silva, V. T. 2008. From the specification to the implementation of norms: an automatic approach to generate rules from norms to govern the behavior of agents. *International Journal of Autonomous Agents and Multiagent Systems (JAAMAS)* **17**(1), 113–155.

Sudeikat, J., Braubach, L., Pokahr, A., Lamersdorf, W. & Renz, W. 2007. Validation of BDI agents. In *Proceedings of the Fourth International Workshop on Programming Multi-Agent Systems (ProMAS'06)*, 185–200.

Tasaki, M., Yabu, Y., Iwanari, Y., Yokoo, M., Tambe, M., Marecki, J. & Varakantham, P. 2008. Introducing communication in Dis-POMDPs with locality of interaction. *International Conference on Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM* **2**, 169–175.

Tinnemeier, N. 2011. *Organizing Agent Organizations: Syntax and Operational Semantics of an Organization-Oriented Programming Language*. PhD thesis, Dutch Research School for Information and Knowledge Systems (SIKS).

Tinnemeier, N., Dastani, M. & Meyer, J.-J. C. 2009a. Roles and norms for programming agent organizations. In *Proceedings of the Eight International Conference on Autonomous Agents and Multiagent Systems (AAMAS 09)*, Decker, K. S., Sichman, J. S., Sierra, C. & Castelfranchi, C. (eds), 121–128. IFAMAAS/ACM DL.

Tinnemeier, N., Dastani, M. & Meyer, J.-J. C. 2010. Programming norm change. In *Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*, van der Hoek, W., Kaminka, G. A., Lesperance, Y., Luck, M. & Sen, S. (eds), 957–964. IFAMAAS/ACM DL.

Tinnemeier, N., Dastani, M., Meyer, J.-J. C. & van der Torre, L. 2009b. Programming normative artifacts with declarative obligations and prohibitions. In *Proceedings of IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology,* 145–152. IEEE Computer Society.

Van Dyke Parunak, H. & Weyns, D. (eds) 2007. Introduction, special issue on environments for multi-agent systems. *Autonomous Agents and Multi-Agent Systems* **14**(1), 1–4.

van Riemsdijk, M. B., Dastani, M., Meyer, J.-J. C. & de Boer, F. S. 2006. Goal-oriented modularity in agent programming. In *Proceedings of the Fifth International Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, 1271–1278.

van Riemsdijk, M. B., van der Hoek, W. & Meyer, J.-J. C. 2003. Agent programming in Dribble: from beliefs to goals using plans. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03),* 393–400. ACM.

Vigueras, G. & Botia, J. A. 2008. Tracking causality by visualization of multi-agent interactions using causality graphs. In *Proceedings of the Fifth International Workshop on Programming Multi-Agent Systems (ProMAS'07)*, 190–204.

Weiss, G. 1999. *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*, The MIT Press, ISBN: 0-262-23203-0.

Weyns, D., Van Dyke Parunak, H., Michel, F., Holvoet, T. & Ferber, J. (eds) 2005. *Environments for Multiagent Systems State-of-the-Art and Research Challenges*, LNCS, **3374**, 1–47. Springer.

Winikoff, M. 2005. JACK™ intelligent agents: an industrial strength platform. In *Multi-Agent Programming: Languages, Platforms and Applications*, 175–193. Kluwer.

Winikoff, M. 2010. Assurance of agent systems: what role should formal verification play? In *Specification and Verification of Multi-Agent Systems* Dastani M., Hindriks K. V. & Meyer J.-J. C. (eds). ACM Press, 353–383, ISBN: 978-1-4419-6983-5.

Winikoff, M., Padgham, L., Harland, J. & Thangarajah, J. 2002. Declarative and procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Respresentation and Reasoning (KR'02).*

Woolridge, M. 2002. *Introduction to Multiagent Systems*, John Wiley & Sons, Inc.

Wooldridge, M. 2009. *An Introduction to MultiAgent Systems*, 2nd edition. Wiley, ISBN: 978-0-470-51946-2.

Zambonelli, F., Jennings, N. R. & Wooldridge, M. 2003. Developing multiagent systems: the Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **12**(3), 317–370.