Languages for Programming BDI-style Agents: an Overview

Viviana Mascardi, Daniela Demergasso, Davide Ancona
DISI, Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy
mascardi@disi.unige.it, 1996s165@educ.disi.unige.it, davide@disi.unige.it

Abstract— **The notion of an intelligent agent as an entity which appears to be the subject of mental attitudes like beliefs, desires and intentions (hence, the BDI acronym) is well known and accepted by many researchers. Besides the definition of various BDI logics, many languages and integrated environments for programming BDI-style agents have been proposed since the early nineties. In this reasoned bibliography, nine languages and implemented systems, namely PRS, dMARS, JACK, JAM, Jadex, AgentSpeak(L), 3APL, Dribble, and Coo-BDI, are discussed and compared. References to other systems and languages based on the BDI model are also provided, as well as pointers to surveys dealing with related topics.**

I. Introduction

The notion of an intelligent agent as an entity which appears to be the subject of beliefs, desires, commitments, and other mental attitudes, is well known and accepted by many researchers [1]. The philosopher Dennett has coined the term *intentional system* to denote systems of this kind [2].

In order to formalise intentional systems, different logics have been developed, among which Cohen and Levesque's theory of intentions [3], and Rao and Georgeff's Belief-Desire-Intention logic [4], [5], [6]. However, intelligent software agents cannot just be formalised using ad-hoc logical languages: they must be programmed using executable languages, as any other piece of software. Hence, there is a pressing need of programming languages which can fill the gap between the logical theory and the practical issues concerned with software agents' development.

One of the computational models that gained more consensus as a candidate to fill this gap is the Belief-Desire-Intention (BDI) one [7], which, as the acronym itself suggests, is characterized by the following concepts:

– *Beliefs*: the agent's knowledge about the world.

– *Desires*: the objectives to be accomplished.

– *Intentions*: the courses of actions currently under execution to achieve the agent's desires.

Besides these components, the BDI model includes a *plan library*, namely a set of "recipes" representing the procedural knowledge of the agent, and an *event queue* where both events (either perceived from the environment or generated by the agent itself to notify an update of its belief base) and internal subgoals (generated by the agent itself while trying to achieve a desire) are stored.

The typical BDI execution cycle is characterised by the following steps:

1. observe the world and the agent's internal state, and update the *event queue* consequently;

2. generate possible new plan instances whose trigger event matches an event in the event queue (*relevant* plan instances) and whose precondition is satisfied (*applicable* plan instances);

3. select for execution one instance from the set of applicable plan instances;

4. push the selected instance onto an existing or new *intention stack*, according to whether or not the event is a (sub)goal;

5. select an intention stack, take the topmost plan instance and execute the next step of this current instance: if the step is an action, perform it, otherwise, if it is a subgoal, insert it on the event queue.

Usually, BDI-style agents do no adopt first principles planning at all, as all plans must be generated by the agent programmer at design time. The planning done by agents consists entirely of context-sensitive subgoal expansion, which is deferred until a point in time at which the subgoal is selected for execution.

This paper provides an overview of languages and implemented systems for programming BDI-style agents. In Section II, nine systems based on the BDI model are surveyed, and in Section III they are compared along seven dimensions. Our knowledge about these nine systems, comes both from our own readings and experience, and from talks with most of the authors of the systems themselves, in particular with R. Bordini (AgentSpeak), M. Dastani (3APL), M. J. Huber (JAM), A. Pokahr (Jadex), M. B. van Riemsdijk (Dribble), and M. Winikoff (AgentTalk). All of them have been asked to check the content of Section III before submission[1]. Section III also contains pointers to other existing systems, together with references to related work.

II. BDI-style languages and systems

The choice of the nine languages and systems that we briefly introduce in this section is motivated either by their historical relevance (PRS, dMARS, AgentSpeak(L)) or by their current significance and with adoption (the remaining six ones). Clearly, there are many more BDI-based languages besides these ones, most of which are cited in Section III.

[1]Despite to the precious advices given by these researchers, the paper might contain inaccuracies, whose responsibility is only ours!

Although we could not treat all the existing languages and systems in depth, our choice does not mean, in any way, that the languages surveyed in this section are "better" (according to whatever criterion) than those cited in Section III.

A. PRS

The SRI's procedural reasoning system, PRS [8], [9], was developed for representing and using an expert's procedural knowledge for accomplishing goals and tasks, based on the research of procedural reasoning carried out at the Artificial Intelligence Center, SRI International. It can be considered the ancestor of all the languages and architectures for practical reasoning discussed in this paper. Procedural knowledge amounts to descriptions of collections of structured actions for use in specific situations. PRS supports the definition of real-time, continuously-active, intelligent systems that make use of procedural knowledge, such as diagnostic programs and system controllers.

Main components of the language: PRS architecture consists of (1) a *database* containing current facts and beliefs, (2) a set of *goals* to be achieved, (3) a set of plans, called *Acts*, describing how sequences of conditional tests and actions may be performed to achieve certain goals or to react to certain situations, and (4) an *interpreter* that manipulates these components to select and execute appropriate plans for achieving the system's goals.

Agent operation: The PRS interpreter runs the entire system. At any particular time, certain goals are established and certain events occur that alter the beliefs held in the system database. These changes in the system's goals and beliefs trigger (invoke) various Acts. One or more of these applicable Acts will then be chosen and placed on the intention graph. Finally, PRS selects a task (intention) from the root of the intention graph and executes one step of that task. This will result either in the performance of a primitive action in the world, the establishment of a new subgoal or the conclusion of some new belief, or a modification to the intention graph itself.

Semantics: We were not able to find documents describing the formal semantics of the original PRS system; our understanding is that the work on giving a formal semantics to PRS started only with the research on dMARS (see Section II-B).

Implementation: A list of implemented PRS systems can be retrieved from M. Wooldridge's page on BDI software, `http://www.csc.liv.ac.uk/~mjw/pubs/rara/resources.html`. The list includes for example PRS-CL (`http://www.ai.sri.com/~prs/`) and UMPRS (`http://ai.eecs.umich.edu/people/durfee/UMPRS.html`).

Industrial-strength applications: The PRS has been evaluated in a simulation of maintenance procedures for the space shuttle, as well as other domains [10].

B. dMARS

dMARS was implemented at the Australian AI Institute, under the direction of M. Georgeff. It was a kind of "second generation PRS", implemented in C++ and used for commercial agent development projects.

Main components of the language: An agent in dMars is characterised by a plan library, three main selection functions which select the intention to execute, the plan to adopt, and the event to manage respectively, and two auxiliary selection functions that are used during the agent's functioning.

A plan is in turn constituted by an invocation condition, an optional context and a mandatory maintainance condition, a body – that is a tree representing the possible flows of actions; arcs are labelled with either an internal or an external action, or a subgoal, while nodes are labelled with states – two sequences of internal actions (updates to the belief base), to be executed if the plan succeeds or fails.

The state of an agent includes the current belief base, the set of current intentions (namely, plan instances which contain information about the current state of execution of the plans they originate from), and the event queue.

Agent operation: The dMARS operation cycle respects the basic cycle depicted in the introduction:
– If the event queue is not empty, an event is selected from it and relevant plans and, in turn, applicable plans are determined. An applicable plan is selected and used to generate a plan instance. With an external event, a new intention containing just the plan instance as a singleton sequence is created. With an internal event, the plan instance is pushed onto the intention stack which generated that (subgoal) event.
– If the event queue is empty, an intention is chosen and the action labelling the current branch in the body of its topmost plan is executed.

A plan fails if all its branches have been attempted, and all of them failed. In this case, the failure actions must be executed. Otherwise, a plan succeeds and the success actions must be executed.

Implementation: The first implementation of dMARS has been developed by the Australian Artificial Intelligence Institute (AAII) - Melbourne, Australia - in 1995. AAII implemented the dMARS platform for distributed reasoning agents consisting of graphical editors, a compiler and an interpreter for a logic goal-oriented programming language, a number of run-time libraries (including an in-memory knowledge database, a multi-threading package and a communication subsystem). dMARS was developed in C++ and ran on a variety of Unix platforms. At the end of 1997, dMARS was being ported to Windows/NT.

Semantics: In [11], an operational semantics of dMARS described using Z [12].

Industrial-strength applications: The dMARS system has been used for both research and production in factory automation, simulation, business and air traffic control systems. Among the customers of dMARS, there are NASA (space shuttle malfunction handling), AirServices, Thomson Airsys (air traffic control), Daimler Chrysler (supply chain management, resource & logistics management), Hazelwood Power (process control). A survey of the applications developed with dMARS can be found in [13].

C. JACK

JACK Intelligent Agents [14], [15] incorporates the BDI model and allows developers to create new reasoning models to suit their customers particular requirements. It is implemented in Java, and the JACK Agent Language extends Java with constructs for agent characteristics such as plans and events. JACK has been built by a team of experts who have worked on PRS and dMARS, and is a commercial product.

Main components of the language: The JACK Agent Language extends the regular Java syntax. It allows the programmers to develop the components that are necessary to define BDI agents and their behaviour, namely:

- Agents - which have methods and data members just like objects, but also contain capabilities that an agent has, namely belief bases (*beliefsets*), descriptions of events that they can handle, and plans that they can use to handle them.
- Capabilities - which serve to encapsulate and aggregate functional components of the JACK Agent Language for use by agents.
- Beliefsets - which are used to store beliefs and data that the agent has acquired.
- Views - which provide a way of modelling any data in a way easily manipulated by JACK.
- Events - which identify the circumstances and messages that it can respond to.
- Plans - which are executed in response to these events.

Agent operation: When an agent is instantiated in a system, it will wait until it is given a goal or it experiences an event to which it must respond. When it receives an event (or goal), the agent initiates activity to handle the event. If it does not believe that the goal or event has already been handled, it will look for the appropriate plan(s) to handle it. The agent then executes the plan or plans depending on the event type. The handling of the event may be synchronous or asynchronous relative to the posting. The plan execution may involve interaction with an agent's beliefset relations or other Java data structures. The plan being executed can in turn initiate other subtasks, which may in turn initiate further subtasks (and so on). Plans can succeed or fail. Under certain circumstances, if the plan fails, the agent may try another plan.

Implementation: JACK is implemented entirely in Java and should run on any Java-based platform. JACK stores all program files and data as normal text, allowing standard configuration management and versioning tools to be used.

Semantics: We did not find any formal semantics of JACK.

Industrial-strength applications: One of the most significant applications of JACK was an unmanned aerial vehicle (UAV) that was guided by an on-board JACK intelligent software agent that directed the aircraft's autopilot during the course of the mission.

D. JAM

JAM is an intelligent agent architecture that grew out of academic research and extended during the last five years of use, development, and application. JAM combines ideas drawn from the BDI theories, the PRS system and its UMPRS and PRS-CL implementations, the SRI International's ACT plan interlingua [16], and the Structured Circuit Semantics (SCS) representation [17]. It also addresses mobility aspects from Agent Tcl [18], Agents for Remote Action (ARA) [19], Aglets [20] and others.

Main components of the language: Each JAM agent is composed of five primary components: a world model, a plan library, an interpreter, an intention structure, and an observer. The world model is a database that represents the beliefs of the agent. The plan library, interpreter, and intention structure has the same purpose of the corresponding components in dMARS. The observer is a user-specified lightweight declarative procedure that the agent interleaves between plan steps (in addition to the reasoning performed by the JAM interpreter) in order to perform functionality outside of the scope of JAM's normal goal/plan-based reasoning (e.g., to buffer incoming messages).

Note that, like in Coo-BDI (discussed in Section II-I), the set of plans available to the agent in the plan library can be augmented during execution through communication with other agents, generated from internal reasoning or by many other means.

Agent operation: The JAM interpreter is responsible for selecting and executing plans based upon the intentions, plans, goals, and beliefs about the current situation. The agent checks all the plans that can be applied to a goal to make sure they are relevant to the current situation. Those plans that are applicable are collected into what is called the Applicable Plan List (or APL). An utility value is determined for each instantiated plan in the APL and, if no meta-level plans are available to select between the APL elements, the JAM interpreter selects the highest utility instantiated plan (called an intention) and intends it to the goal. Note that neither the original PRS specification nor prior PRS-based implementations (such as PRC-CL) support utility-based reasoning.

Implementation: JAM is distributed freely for non-commercial use and can be downloaded from `http://www.marcush.net/IRS/download_jam.html`

Semantics: We are not aware of any formal semantics of JAM.

Industrial-strength applications: We could not find any information on industrial-strength applications developed using JAM.

E. Jadex

The Jadex research project is conducted by the Distributed Systems and Information Systems Group at the University of Hamburg. The developed software framework is currently in a beta-stage. A basic set of features already supports the development of rational agents on top of the the FIPA-compliant JADE platform [21]. The main purposes of Jadex are both to bring together BDI-style reasoning and FIPA-compliant communication [22], and to extend the traditional BDI-model (e.g. with explicit goals).

Main components of the language: Jadex agents have beliefs, which can be any kind of Java object and are stored in a belief base, goals, that are implicit or explicit descriptions of states to be achieved, and plans, that are procedural recipes coded in Java.

Agent operation: After initialisation, the Jadex runtime engine executes the agent by keeping track of its goals while continuously selecting and executing plan steps, based on internal events and messages from other agents. Jadex is supplied with some predefined functionalities and can integrate third party tools like the "beangenerator" plug-in for the ontology design tool Protégé [23].

Implementation: Jadex is implemented on top of JADE. To easily integrate the Jadex engine (implemented in Java) into JADE agents, a wrapper agent class is provided, which creates and initialises an instance of the Jadex engine with the beliefs, goals and plans from an agent definition file.

Semantics: For the basic operation of the Jadex interpreter, as well as for some specific aspects such as goal deliberation, an operational semantics has been sketched in [24].

Industrial-strength applications: From the Jadex home page (`http://vsis-www.informatik.uni-hamburg.de/projects/jadex/`), the pointers to three applications developed using Jadex, namely MedPAge, Dynatech, and Blackjack, can be found.

F. AgentSpeak(L)

AgentSpeak(L) [25] takes as its starting point PRS and dMARS and formalizes its operational semantics. It can be viewed as a simplified, textual language of PRS or dMARS.

Main components of the language: AgentSpeak(L) is based on a restricted first-order language with events and actions. The beliefs, desires and intentions of the agent are not represented as modal formulas, but they are ascribed to agents, in an implicit way, at design time. The current state of the agent can be viewed as its current belief base; states that the agent wants to bring about can be viewed as desires; and the adoption of programs to satisfy such stimuli can be viewed as intentions.

Agent operation: Like in PRS and dMARS, at every interpretation cycle of an agent program, AgentSpeak(L) updates a list of events, which may be generated from perception of the environment, or from the execution of intentions (when subgoals are specified in the body of plan). In [26], R. Machado and R. H. Bordini have introduced a Belief Revision Function (BRF) in the architecture which is implicit in Rao's interpreter, and in [27] R. H. Bordini, et al., enhance the interpreter with an efficient intention selection in BDI agents via decision-theoretic task scheduling.

Implementation: There are many implementations of the AgentSpeak(L) language, among which:

- SIM_Speak [26] (the first working AgentSpeak(L) interpreter), which runs on Sloman's SIM_AGENT toolkit, a testbed for cognitively rich agent architectures [28], and
- Jason [29], which provides an interpreter for a version of AgentSpeak(L) extended with speech-acts [30]; Jason supports the distribution of the agents by means of SACI [31].
- AgentTalk [32], an interpreter for a simplified version of AgentSpeak(L) implemented by M. Winikoff.

Semantics: The agent operation described above is formalised in [25], [33], and [34].

Industrial-strength applications: AgentSpeak(L) has been used to program animated embodied agents in virtual environments.

G. 3APL

3APL [35] supports the design and construction of intelligent agents for the development of complex systems through a set of intuitive concepts like beliefs, goals and plans.

Although the 3APL architecture has many similarities with other cognitive architectures such as PRS, it departs from them in many ways. For example, the PRS architecture is designed to plan agents' goals (desires) while the 3APL architecture is designed to control and revise agents' to-do-goals. Moreover, there is no ingredient in the PRS architecture that corresponds to the practical reasoning rules, which are a powerful mechanism to revise mental attitudes. Finally, the deliberation cycle in 3APL is supposed to be a programmable component while the deliberation cycle in PRS is integrated.

Main components of the language: An agent in 3APL is characterised by two sets: the expertise of the agent, which is a set of actions, and the agent's rule base, which is a set of rules.

A rule is formed by:

- an optional head and an optional body (both of which are goals, namely either basic actions, or queries, or achievement, or sequences of goals, or nondeterministic choices of goals, or goal variables);
- a guard, that is a belief;
- a type, that may be either `reactive`, or `failure`, or `plan`, or `optimisation`.

In 3APL goals represent both the target of the agent and the way to achieve this target, thus 3APL goals are similar both to dMARS achieve goals and to dMARS plans.

The state of a 3APL agent is constituted by its belief base and its goal base.

Agent operation: The architecture for 3APL is based on the think-act cycle, which is divided into two parts. The first part corresponds to a phase of practical reasoning by using practical reasoning rules, and the second corresponds to an execution phase in which the agent performs some action.

Think stage. The application of a rule to a goal results in the replacement of a subgoal which matches with the head of the rule by the body of the rule in case the head of the rule is non-empty. If the body of the rule is empty, the subgoal is simply dropped. In case the head of a rule is empty only the guard of the rule needs to be derivable from the beliefs of the agent, and a new goal (the body of the rule) is added to the goal base of the agent.

Act stage. The execution of a goal is specified through the computation steps an agent can perform on a goal. A computation step corresponds to a simple action of the agent, which is either a basic action or else a query on the beliefs of the agent.

Implementation: Both a Java version and an Haskell version of 3APL can be downloaded from `http://www.cs.uu.nl/3apl/download.html`.

Semantics: Originally, the operational semantics of 3APL was specified by means of Plotkin-style transition semantics [36], while in [37] 3APL has been re-specified in Z. In [38], the specification of a programming language for implementing the deliberation cycle of cognitive agents is shown, and 3APL has been used as the object language.

Industrial-strength applications: We are not aware of real applications developed using 3APL.

H. Dribble

Dribble [39] is a propositional language that constitutes a synthesis between the declarative features of the language GOAL [40], and the procedural features of 3APL.

Some attention should be devoted to the terminology used. In the original paper on 3APL [35], 3APL is defined to have beliefs and goals (no plans). These goals are however procedural (basically sequences of actions) and are actually the same as the plans of Dribble (modulo some details). The important feature of Dribble compared with the original version of 3APL, is the addition of declarative goals (based on GOAL). In the Dribble paper, the term "goal" has been used for declarative goals (in the sense of propositional formulas describing a situation that is to be achieved), and "plans" for the procedural part of the agent (which was termed "goals" in [35]). Further, the ideas of Dribble have been incorporated in the latest version of 3APL, as discussed in [41]. That paper presents a first order version of Dribble, with some minor extensions. It uses the Dribble terminology of "goals" for declarative goals and "plans" for the procedural part.

Main components of the language: The language Dribble incorporates beliefs, declarative goals, and plans (i.e., procedural goals, following 3APL terminology).

Agent operation: Dribble basically adopts a Think-Act cycle like 3APL.

Implementation: We were not able to find documents describing a working implementation of the Dribble language.

Semantics: In [39], an operational semantics of the possible mental state changes is defined using transition systems. A dynamic logic is also sketched in which one can reason about actions defined in that logic. These actions transform the mental state of the agent. In [42], a demonstration that mental state transitions defined by actions in the logic, correspond to the mental state transitions defined by the transition system is provided.

Industrial-strength applications: We are not aware of applications developed using Dribble.

I. Coo-BDI

Coo-BDI (Cooperative BDI) [43] is based on the dMARS specification and extends it by introducing *cooperations* among agents to retrieve external plans for achieving desires.

Main components of the language: The cooperation strategy of an agent *A* includes the set of agents with which is expected to cooperate (a set of agent names), the plan retrieval policy (*always*, *noLocal*) and the plan acquisition policy (*discard*, *add*, *replace*).

Coo-BDI plans are classified in *specific* and *default* ones; besides the standard components, they also have an *access specifier* which determines the set of agents the plan can be shared with (*private*, *public* and *only(TrustedAgents)*).

Coo-BDI intentions are characterized by "standard" components plus components introduced to manage the external plan retrieval mechanism.

Agent operation: The operation of a Coo-BDI agent is based on a three steps cycle:

1) process the event queue;
2) process suspended intentions;
3) process active intentions.

The mechanism for retrieving relevant plans involves cooperation with the trusted agents, in order to retrieve external plans, besides the local ones.

Implementation: An integration of the ideas underlying Coo-BDI into the Jason programming language has been designed [44], and its implementation is under way [45].

Semantics: No formal semantics of Coo-BDI has been defined. In [43] the Coo-BDI interpreter is fully described in Prolog, which gives an operational specification of its operation.

Industrial-strength applications: No applications have been developed using Coo-BDI.

III. Comparison and related work

In Tables 1 and 2, we summarise the analysis of the nine surveyed systems along seven dimensions. In particular, in Table 2 we take the ability of the agents to easily integrate ontologies (**Ont**) and to update the plan library at runtime (**Dyn**) into account. In Table 2, references are given if the analysed feature is not supported by the original system, but by some of its extensions. Instead, a "Yes" in the cell means that the original version of the system natively supports the corresponding feature.

Many resources on BDI-style languages are available to the research community, although, to the best of our knowledge, no exhaustive roadmap on this topic exists. An introduction to the BDI logics, architecture, and to some languages based on BDI concepts can be found both in [54] and in [1]. The paper [55] discusses and compares five MAS development toolkits, namely AgentBuilder [56], CaseLP [57], DESIRE [58], IMPACT [59], and ZEUS [60], that support the definition of agents in terms of mental attitudes.

Among the on-line resources, `http://www.csc.liv.ac.uk/~mjw/pubs/rara/resources.html` provides pointers to some

	Implementations	Formal semantics	Industrial-strength applic.
PRS	UMPRS [46], PRS-CL [47], others [48]	No	[10]
dMARS	In 1995, AAII implemented a C++ platform running on Unix; in 1997 dMARS was ported to Windows/NT	Operational [11]	[13]
JACK	Java [49]	No	Unmanned vehicle
JAM	Java [50]	No	No
Jadex	Java [51]	Operational (sketched in [24])	[51]
AS(L)	SIM_Speak [26], AgentTalk [32], Jason [29]	Operational [33], [25], [34]	Virtual environments
3APL	Java and Prolog [52]	Operational [37], [36]; meta-level [38]	No
Dribble	No	Operational [39], dynamic logic-based [42]	No
Coo-BDI	Coo-AgentSpeak [44], [45]	Operational [43]	No

TABLE I

IMPLEMENTATIONS, SEMANTICS, APPLICATIONS OF THE SURVEYED SYSTEMS

	Basic components	Operation cycle	Ont	Dyn
PRS	Standard	Standard	No	No
dMARS	Standard	Standard	No	No
JACK	Standard + capabilities (that aggregate functional components) + views (to easily model data)	Standard	No	No
JAM	Standard + observer (user-specified declarative procedure that the agent interleaves between plan steps) + utility of plans	Utility-based	No	Yes
Jadex	Beliefs + goals + plans + capabilities (that aggregate functional components)	Standard	Yes	No
AS(L)	Standard	Standard; efficient [27]	Yes [53]	Yes [44]
3APL	Beliefs, plans, practical reasoning rules, basic action specifications	Think-act	No	Yes
Dribble	Beliefs, plans, declarative goals, practical reasoning rules, goal rules, basic action specifications	Think-act	No	Yes
Coo-BDI	Standard + cooperation strategy (trusted agents + plan retrieval and acquisition policies) + plans' access specifiers	Perceive-cooperate-act	No	Yes

TABLE II

OTHER FEATURES OF THE SURVEYED LANGUAGES AND SYSTEMS

implemented BDI systems, while `http://www.cs.rmit.edu.au/agents/SAC/survey.html` surveys systems based on the concepts of action, event, plan, belief, goal, decision and choice.

Besides the BDI-based languages and integrated environments that we have not discussed in Section II, we can cite: MYWORLD [61], in which agents are directly programmed in terms of beliefs and intentions; ViP [62], a visual programming language for plan execution systems with a formal semantics based upon an agent process algebra; CAN [63], a conceptual notation for agents with procedural and declarative goals; NUIN [64], a Java framework for building BDI agents, with strong emphasis on Semantic Web aspects; SPARK [65], that builds on PRS and supports the construction of large-scale, practical agent systems; and Jason [29], that supports many extensions to the AgentSpeak language and is a BDI system in the spirit of Jadex, JAM, JACK, but with much better formal basis.

When we move from BDI-style languages to the more general class of agent programming languages based on computational logics (which, however, includes the BDI-style languages), we can find two surveys that complement each other in many ways. The first one, [66], discusses different formalisms with the aim of putting in evidence the contribution of logic to knowledge representation formalisms and to basic mechanisms and languages for agents and MAS modeling. The second one, [67], analyses a subset of logic-based executable languages whose main features are their suitability for specifying agents and MASs and their possible integration into an existing conceptual framework for agent-oriented software engineering based on computational logic.

ACKNOWLEDGEMENTS

The authors acknowledge Rafael Bordini, Mehdi Dastani, Marcus J. Huber, Alexander Pokahr, M. Birna van Riemsdijk, and Michael Winikoff for their precious advices.

This work was partially funded by the MIUR project "Sviluppo e verifica di sistemi multi-agente basati sulla logica", 2004-2005, coordinated by A. Martelli.

REFERENCES

[1] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[2] D. C. Dennett, *The Intentional Stance*. The MIT Press, 1987.

[3] P. R. Cohen and H. J. Levesque, "Intention is choice with commitment," *Artificial Intelligence*, vol. 42, 1990.

[4] A. S. Rao and M. P. Georgeff, "Modeling rational agents within a BDI–architecture," in *Proc. of KR'91*, 1991, pp. 473–484.

[5] ——, "Asymmetry thesis and side-effect problems in linear-time and branching-time intention logics." in *Proc. of IJCAI'91*, 1991, pp. 498–504.

[6] ——, "A model-theoretic approach to the verification of situated reasoning systems," in *Proc. of IJCAI'93*, 1993, pp. 318–324.

[7] ——, "BDI agents: from theory to practice," in *Proc. of ICMAS'95*, 1995, pp. 312–319.

[8] M. P. Georgeff and A. L. Lansky, "Reactive reasoning and planning," in *Proc. of AAAI'87*, 1987, pp. 677–682.

[9] K. L. Myers, "User guide for the procedural reasoning system," Artificial Intelligence Center, SRI International, Menlo Park, CA, Tech. Rep., 1997.

[10] M. P. Georgeff and F. F. Ingrand, "Decision-making in an embedded reasoning system," in *Proc. of IJCAI'89*, 1989, pp. 972–978.

[11] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge, "A formal specification of dMARS," in *Proc. of ATAL'97*, 1997, pp. 155–176.

[12] M. Spivey, *The Z Notation: A Reference Manual, 2nd edition*. Prentice Hall International Series in Computer Science, 1992.

[13] M. P. Georgeff and A. S. Rao, "A profile of the Australian AI institute," *IEEE Expert*, vol. 11, no. 6, pp. 89–92, 1996.

[14] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas, "JACK intelligent agents – components for intelligent agents in Java," *AgentLink News Letter*, vol. 2, 1999.

[15] Agent Oriented Software Group, "What is JACK?" http://www.agent-software.com/shared/products/index.html.

[16] K. L. Myers and D. E. Wilkins, "The Act Formalism, Version 2.2," SRI International AI Center Technical Report, SRI International, Menlo Park, CA, Tech. Rep., 1997.

[17] J. Lee and E. H. Durfee, "Structured circuit semantics for reactive plan execution systems," in *Proc. of AAAI'94*, 1994, pp. 1232–1237.

[18] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus, "Agent Tcl," in *Mobile Agents: Explanations and Examples*. Manning Publishing, 1997.

[19] H. Peine, "ARA - Agents for Remote Action," in *Mobile Agents*. Manning Publishing, 1997.

[20] D. Lange and O. Mitsuru, *Programming and Deploying Java Mobile Agents with Aglets*, 1998.

[21] JADE Home Page, http://jade.tilab.com/.

[22] FIPA Home Page, http://www.fipa.org/.

[23] Protégé Home Page, http://protege.stanford.edu/.

[24] A. Pokahr, L. Braubach, and W. Lamersdorf, "A flexible BDI architecture supporting extensibility," in *Proc. of IAT-2005*, 2005, pp. 379–385.

[25] A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *Proc. of MAAMAW'96*, 1996, pp. 42–55.

[26] R. Machado and R. H. Bordini, "Running AgentSpeak(L) agents on SIM_AGENT," in *Proc. of ATAL'01*, 2001, pp. 158–174.

[27] R. H. Bordini, A. L. C. Bazzan, R. de O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser, "AgentSpeak(XL): efficient intention selection in BDI agents via decision-theoretic task scheduling," in *Proc. of AAMAS'02*, 2002, pp. 1294–1302.

[28] A. Sloman and R. Poli, "SIM_AGENT: A toolkit for exploring agent design," in *Proc. of ATAL'95*. Springer-Verlag, 1995, pp. 392–407.

[29] R. H. Bordini, J. F. Hübner, *et al.*, *Jason: A Java-based AgentSpeak interpreter used with SACI for multi-agent distribution over the net*, Manual, release 0.5 ed., 2004.

[30] A. F. Moreira, R. Vieira, and R. H. Bordini, "Extending the Operational Semantics of a BDI Agent-Oriented Programming Language for Introducing Speech-Act Based Communication," in *Proc. of DALT'03*. Springer-Verlag, 2003, pp. 135–154.

[31] J. F. Hübner and J. S. Sichman, *SACI — Simple Agent Communication Infrastructure*, 2003, sACI Home Page: `http://www.lti.pcs.usp.br/saci/`.

[32] M. Winikoff, "The AgentTalk home page," http://goanna.cs.rmit.edu.au/~winikoff/agenttalk/.

[33] M. d'Inverno and M. Luck, "Engineering AgentSpeak(L): A formal computational model," *Logic and Computation Journal*, vol. 8, no. 3, pp. 1–27, 1998.

[34] R. H. Bordini and A. F. Moreira, "Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L)," *Annals of Math. and AI*, vol. 42, no. 1–3, pp. 197–226, 2004.

[35] K. V. Hindriks, F. S. D. Boer, W. V. der Hoek, and J.-J. C. Meyer, "Agent programming in 3APL," *AAMAS Journal*, vol. 2, no. 4, pp. 357–401, 1999.

[36] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer, "Formal semantics for an abstract agent programming language," in *Proc. of ATAL'97*, 1997, pp. 215–229.

[37] M. d'Inverno, K. V. Hindriks, and M. Luck, "A formal architecture for the 3APL agent programming language," in *Proc. of ZB'00*, 2000, pp. 168–187.

[38] M. Dastani, F. S. de Boer, F. Dignum, and J.-J. C. Meyer, "Programming agent deliberation – an approach illustrated using the 3APL language," in *Proc. of AAMAS'03*, 2003.

[39] B. van Riemsdijk, W. van der Hoek, and J.-J. C. Meyer, "Agent programming in Dribble: from beliefs to goals using plans," in *Proc. of AAMAS'03*, 2003, pp. 393–400.

[40] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer, "Agent programming with declarative goals," in *Proc. of ATAL'00*, 2000, pp. 228–243.

[41] M. Dastani, M. B. van Riemsdijk, F. Dignum, and J.-J. C. Meyer, "A programming language for cognitive agents: goal directed 3APL," in *Proc. of ProMAS'03*, ser. LNAI. Springer, 2004, vol. 3067, pp. 111–130.

[42] M. B. van Riemsdijk, "Agent programming in Dribble: from beliefs to goals with plans," Master's thesis, Utrecht University, 2002.

[43] D. Ancona and V. Mascardi, "Coo-BDI: Extending the BDI model with cooperativity," in *Post-proc. of DALT'03*, 2004, pp. 109–134.

[44] D. Ancona, V. Mascardi, J. F. Hübner, and R. H. Bordini, "Coo-AgentSpeak: Cooperation in AgentSpeak through Plan Exchange," in *Proc. of AAMAS'04*, 2004, pp. 698–705.

[45] D. Demergasso, "Coo-AgentSpeak: un linguaggio per agenti deliberativi e cooperativi," Master's thesis, DISI – Università di Genova, 2005, in Italian.

[46] J. Lee, M. J. Huber, P. G. Kenny, and E. H. Durfee, "UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications," in *Proc. of CIRFFSS'94*, 1994, pp. 842–849.

[47] PRS-CL Home Page, http://www.ai.sri.com/~prs/.

[48] M. Wooldridge's List of PRS Implementations, http://www.csc.liv.ac.uk/~mjw/pubs/rara/resources.html.

[49] JACK Home Page, http://www.agent-software.com/shared/products/index.html.

[50] JAM Home Page, http://www.marcush.net/IRS/irs_downloads.html.

[51] Jadex Home Page, http://vsis-www.informatik.uni-hamburg.de/projects/jadex/.

[52] 3APL Home Page, http://www.cs.uu.nl/3apl/.

[53] A. F. Moreira, R. Vieira, R. H. Bordini, and J. Hübner, "Agent-oriented programming with underlying ontological reasoning," in *Proc. of DALT'05*, 2005.

[54] M. Wooldridge and N. R. Jennings, "Agent theories, architectures, and languages: A survey," in *Proc of. ECAI ATAL Workshop*, 1994, pp. 1–39.

[55] T. Eiter and V. Mascardi, "Comparing Environments for Developing Software Agents," *AI Communications*, vol. 15, no. 4, pp. 169–197, 2002.

[56] AgentBuilder Home Page, http://www.agentbuilder.com/.

[57] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini, "Multi-agent systems development as a software engineering enterprise," in *Proc. of PADL'99*, 1999, pp. 46–60.

[58] DESIRE Home Page, http://www.cs.vu.nl/vakgroepen/ai/projects/desire/desire.html.

[59] T. Eiter, V. S. Subrahmanian, and G. Pick, "Heterogeneous active agents, I: Semantics," *Artificial Intelligence*, vol. 108, no. 1-2, pp. 179–255, 1999.

[60] H. S. Nwana, D. T. Ndumu, L. C. Lee, and J. C. Collis, "ZEUS: A tool-kit for building distributed multi-agent systems," *Applied Artifical Intelligence Journal*, vol. 13, no. 1, pp. 129–185, 1999.

[61] M. Wooldridge, "This is MYWORLD: The logic of an agent-oriented testbed for DAI," in *Proc. of ECAI ATAL Workshop*, 1994, pp. 160–178.

[62] D. Kinny, "ViP: a visual programming language for plan execution systems," in *Proc. of AAMAS'02*, 2002, pp. 721–728.

[63] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah, "Declarative & procedural goals in intelligent agent systems," in *Proc. of KR'02*, 2002, pp. 470–481.

[64] I. Dickinson and M. Wooldridge, "Towards practical reasoning agents for the semantic web," in *Proc. of AAMAS'03*, 2003, pp. 827–834.

[65] D. Morley and K. Myers, "The SPARK agent framework," in *Proc. of AAMAS'04*, 2004, pp. 714–721.

[66] F. Sadri and F. Toni, "Computational Logic and Multi-Agent Systems: a Roadmap," Department of Computing, Imperial College, London, Tech. Rep., 1999.

[67] V. Mascardi, M. Martelli, and L. Sterling, "Logic-based specification languages for intelligent software agents," *TPLP Journal*, vol. 4, no. 4, pp. 429–494, 2004.