A Survey of Programming Languages and Platforms for Multi-Agent Systems

Rafael H. Bordini
University of Durham, UK
E-mail: R.Bordini@durham.ac.uk, http://www.dur.ac.uk/r.bordini

Lars Braubach
Universität Hamburg, Germany
E-mail: braubach@informatik.uni-hamburg.de, http://vsis-www.informatik.uni-hamburg.de

Mehdi Dastani
Utrecht University, The Netherlands
E-mail: mehdi@cs.uu.nl, http://www.cs.uu.nl/~mehdi

Amal El Fallah Seghrouchni
University of Paris 6, France
E-mail: Amal.Elfallah@lip6.fr, http://www-poleia.lip6.fr/~elfallah

Jorge J. Gomez-Sanz
Universidad Complutense de Madrid, Spain
E-mail: jjgomez@sip.ucm.es, http://grasia.fdi.ucm.es/jorge

João Leite
Universidade Nova de Lisboa, Portugal
E-mail: jleite@di.fct.unl.pt, http://centria.di.fct.unl.pt/~jleite

Gregory O'Hare
University College Dublin, Ireland
E-mail: Gregory.OHare@ucd.ie, http://www.cs.ucd.ie/staff/gohare

Alexander Pokahr
Universität Hamburg, Germany
E-mail: pokahr@informatik.uni-hamburg.de, http://vsis-www.informatik.uni-hamburg.de

Alessandro Ricci
Università di Bologna, Italy
E-mail: aricci@deis.unibo.it, http://lia.deis.unibo.it/~ari

Keywords: Multi-Agent Systems, Programming Languages, Platforms

Received: April 1, 2005

This paper surveys recent research on programming languages and development tools for Multi-Agent Systems. It starts by addressing programming languages (declarative, imperative, and hybrid), followed by integrated development environments, and finally platforms and frameworks. To illustrate each of these categories, some systems were chosen based on the extent to which European researchers have contributed to their development. The current state of these systems is described and, in some cases, indications of future directions of research are given.

Povzetek: Podan je pregled jezikov in orodij za MAS.

1 Introduction

Research in Multi-Agent Systems (MAS) has recently led to the development of practical *programming languages and tools* that are appropriate for the implementation of such systems. Putting together this new programming paradigm is fast becoming one of the most important topics of research in multi-agent systems, in particular because this is an essential requirement for an eventual technology transfer.

Surveying the MAS literature will reveal a large number of different proposals for agent-oriented languages, ranging from purely declarative, to purely imperative, and various hybrid approaches. Some are designed from scratch,

directly encoding some theory of agency, while others extend existing languages to suit the peculiarities of this new paradigm. Using these languages, instead of more conventional ones, proves useful when the problem is modelled as a *multi-agent* system, and understood in terms of cognitive and social concepts such as beliefs, goals, plans, roles, and norms.

Most agent programming languages have some underlying platform which implements its semantics. However, agent frameworks exist that are not tightly coupled with one specific programming language. Instead, they are concerned with providing general techniques for relevant aspects such as agent communication and coordination. The most mature languages will be accompanied by some Integrated Development Environment (IDE), intended to enhance the productivity of programmers by automating tedious coding tasks. Typically these will provide functionalities such as project management, creating and editing source files, refactoring, build and run process, and testing.

Despite the large number of languages, frameworks, development environments, and platforms recently proposed, implementing MAS is still an often daunting task. To address the problem of managing the inherent complexity of MAS and helping the structuring of their development, the research community has produced a number of methodologies [4]. Nevertheless, even if MAS practitioners follow such methodologies during the design phase, they still find great difficulties in the implementation phase, partly due to the lack of maturity of both methodologies and programming tools. Among others, such difficulties can be traced to the lack of specialised debugging tools; to the lack skills that are necessary in mapping analysis/design concepts to programming languages constructs; to the lack of proficiency in dealing with the specific characteristics of different agent platforms; and also to the lack of understanding of the very foundations as well as practical characteristics of the agent-oriented approach to programming.

Even though most of the languages and tools developed so far have not been tried yet in large-scale, industrial-strength applications, much progress has been achieved in the last few years. This is, therefore, an appropriate time for a *reality check* and a bird's eye view of the field, helping to consolidate existing achievements and guide future developments. To this end, this paper surveys some of the existing approaches situated in the MAS Programming area of research, from programming languages to development infrastructures, chosen in part according to the extent to which European researchers have contributed to their development.

The first part of the paper is devoted to the presentation of agent-oriented programming languages, structured according to the existing paradigm on which they build. In Section 2, we present declarative agent-oriented languages, while Section 3 covers the imperative languages and Section 4 some hybrid languages. The second part will cover various implementations of software infrastructure for agents. These will be structured according to whether they are development environments for MAS, in Section 5, or MAS platforms and frameworks, in Section 6. The paper ends with some reference to further readings on this subject in Section 7, and some final remarks in Section 8.

2 Declarative Languages

Declarative languages are partially characterised by their strong formal nature, normally grounded on logic. This is the case with most of the declarative languages described here: FLUX, Minerva, Dali, and ResPect. Other declarative languages are also grounded on other formalisms, such as CLAIM which finds parts of its roots in the ambient calculus. Declarative languages that allow for easy integration with imperative code will be reviewed in Section 4 below.

CLAIM (**C**omputational **L**anguage for **A**utonomous, **I**ntelligent and **M**obile Agents [23]) is a high-level declarative agent-oriented programming language. It is part of an unified framework called Himalaya [25] (**H**ierarchical **I**ntelligent **M**obile **A**gents for building **L**arge-scale and **A**daptive s**Y**stems based on **A**mbients). It combines the main advantages of agent-oriented programming languages, for representing cognitive aspects and reasoning, with those of concurrent languages based on process algebra, for representing concurrency and agent mobility.

The CLAIM language is inspired by *ambient calculus* [11] and agents are hierarchically organised, thus supporting the design of Mobile Multi-Agent Systems (MMAS) – a set of connected hierarchies of agents – to be deployed on a network of computers. Every agent (i.e., a node of a hierarchy) contains cognitive elements (e.g., knowledge, goals, capabilities), processes, and sub-agents and is also mobile as it can move within its hierarchy or to a remote one. In addition, an agent can dynamically acquire intelligent and computational components from its sub-agents, which can be seen as some sort of inheritance. The mobility and the inheritance as defined in Himalaya framework favour a dynamic adaptability and reconfiguration of systems [50] for coping with the increasing complexity of distributed and cooperative applications. The main elements of CLAIM agents are cognitive, interaction, mobility, and reconfiguration primitives.

The formal semantics of CLAIM is based on Plotkin's [41] structural operational approach consisting of a transition relation, from an initial state of a program to another state resulting from the execution of an atomic operation. At each step of an agent execution, either a message is dealt with, a running process executed, or a goal processed. For a detailed presentation of the semantics, we refer the reader to [24].

As an MMAS within Himalaya is deployed on a set of connected computers, the language CLAIM is supported by a distributed platform called SyMPA [51], which offers all the necessary mechanisms for management of agents, communication, mobility, security, fault-tolerance, and load balancing [30]. SyMPA is implemented in Java and

compliant with the specifications of the MASIF [37] standard from the OMG (Object Management Group). There is a central system providing management functions. An agent system is deployed on each computer connected to the platform.

The Himalaya environment has been used for developing several complex applications that showed the expressiveness of the language and the robustness and strength of the platform, such as: an application for information search on the Web [22], several electronic commerce applications [23, 52], a load balancing and resource sharing application using mobile agents [30], and an application for a network of digital libraries.

FLUX [53] is a high-level programming system for cognitive agents, which can be downloaded from `http://www.fluxagent.org`. It consists of an implementation of the Fluent Calculus, an action representation formalism that provides a basic solution to the classical frame problem using the concept of state update axioms, while addressing a variety of aspects in reasoning about actions (hence the relevance for agents), such as ramifications (i.e., indirect effects of actions), qualifications (i.e., unexpected action failure), nondeterministic actions, concurrent actions, continuous change, and noisy sensors and effectors.

An agent program in FLUX is a logic program consisting of three parts: the kernel providing the agent with the cognitive ability to reason about its actions and acquired sensor data, a background theory providing an internal model of its environment, and a strategy which specifies the task-oriented behaviour in accordance with which the agent reasons, plans, and acts. The full expressive power of logic programming can be used to design strategies while facilitating formal proofs of the correctness of strategies with respect to a problem-dependent specification.

The use of progression, where a (possibly incomplete) initial world model is updated upon the performance of an action, is one of the main characteristics of FLUX. This allows for a computationally efficient solution to the frame problem and, consequently, an efficient agent implementation based on the Fluent Calculus. Further information regarding FLUX can be obtained in [54].

MINERVA [32, 33] is an agent system designed to provide a common agent framework based on the strengths of Logic Programming, to allow for the combination of several existing non-monotonic knowledge representation and reasoning mechanisms. It uses MDLP and KABUL to specify agents and their behaviour. A MINERVA agent consists of several specialised, possibly concurrent, subagents performing various tasks, whose behaviour is specified in KABUL, while reading and manipulating a common knowledge base specified in MDLP.

MDLP (**M**ulti-Dimensional **D**ynamic **L**ogic **P**rogramming) is the basic knowledge representation mechanism of an agent in MINERVA. MDLP is an extension of Answer Set Programming (ASP) where knowledge is represented by logic programs arranged in an acyclic digraph. In this digraph, vertices are sets of logic programs, and edges represent the relations between program. MDLP enjoys the merits of ASP such as default negation. Default negation allows the definition of non-monotonic behaviour thus facilitating the representation of, and reasoning about, incomplete knowledge. MDLP also allows for the simultaneous representation of several aspects such as hierarchies and preferences, as well as the evolution of the represented knowledge.

KABUL (**K**nowledge **A**nd **B**ehavior **U**pdate **L**anguage), as its recent evolution EVOLP [1], is a logic-programming style language that allows the specification of updates to a knowledge base and to itself. A program in KABUL is a set of *statements*, each statement being a type of condition-action rule that can be seen as encoding an agent behaviour. The epistemic effects of actions can be either an update to the knowledge base of the agent, represented by an MDLP program, or a self update to the KABUL program, thus changing the behaviour of the agent over time. Conditions range from external observations, epistemic state of the agent, as well as concurrent execution of other actions. This allows for a combination of reactive and proactive behaviour, in the sense that no external stimuli are needed to trigger the behaviour of the agent, while these can be combined with the rational features provided by the underlying MDLP knowledge representation framework and its formal and precise ASP-based semantics. More information regarding MDLP, KABUL, and MINERVA can be found in [32].

DALI [14] is an Active Logic Programming language designed for executable specification of logical agents. It uses plain Horn Clauses and its semantics is based on Least Herbrand Models. It intends to provide constructs to represent reactivity and proactivity in an agent by means of rules. A DALI agent is a logic program that contains reactive rules, events, and actions aimed at interacting with an external environment. The reactive and proactive behaviour of a DALI agent is triggered by several kinds of events: external, internal, present, and past events. All the events and actions are time stamped so as to record when they occurred. The new syntactic entities, i.e., predicates related to events and proactivity, are indicated with special postfixes. When an event occurs in the agent's "external world", the agent can perceive it and decide to react. The reaction is defined by a reactive rule which has in its head that external event. The internal events define the behaviour of a DALI agent, making it proactive independently of the environment and allowing it to manipulate and revise its knowledge.

ReSpecT [38] is a logic-based language, with a well-defined formal semantics, allowing for the definition of reactions, expressed in terms of rules. A rule in ReSpecT consists of a head specifying the communication event that triggers the reaction and a body specifying which actions (tuples from the tuple centre) are atomically executed when the reaction is triggered. When a basic action fails, the reaction atomically fails and all its effects on the tuple centre state are rolled back. The coordinating behaviour of

tuple centres can be changed and adapted at runtime by dynamically changing the reactions defined in ReSpecT. Such a feature is typically exploited to deal with dynamism and openness of MAS applications. The tuple centre programmed with these reactions acts as a basic scheduler, encapsulating the policy adopted to coordinate the various (autonomous) agent tasks. By changing the reactions, the overall coordinating behaviour of the system changes, without the need to change the agent's behaviour. This language is used within the TuCSoN framework (discussed below in Section 6).

3 Imperative Languages

Purely imperative approaches to agent-oriented programming are less common, mainly due to the fact that most abstractions related to agent-oriented design are, typically, declarative in nature. There are however many programmers who still use conventional, i.e. non-agent oriented, imperative languages for developing multi-agent systems; as a result, in practice agent notions are often implemented in an *ad-hoc* manner. An example of an agent-oriented language which is still essentially imperative, while incorporating agent-specific abstractions, is the language available with the development environment JACK [57, 26].

The **JACK Agent Language (JAL)** has been developed by a company called Agent Oriented Software. JAL is based on ideas of reactive planning systems resulting from the work on the BDI agent architecture and is, in this respect, similar to the hybrid languages Jason, 3APL, and Jadex (discussed below in Section 4). However, instead of providing a logic-based language, JAL is an extension of Java (implementing some features of logic languages such as logical variables). A number of syntactic constructs is added to Java, allowing programmers to create plans and belief bases, all in a graphical manner as JAL has a sophisticated IDE which provides a tool for such purpose. In JAL, plans can be composed of *reasoning methods* and grouped into *capabilities* which, together, compose a specific ability an agent is supposed to have, thus supporting a good degree of modularisation. Another structuring mechanism present in JAL is the ability to use *teams* of agents, or agent organisations, a notion that is increasingly important both in agent-oriented design [4] and because of recent developments in self-organising systems [47]. Although JAL has no formal semantics, as a commercial platform, JACK has extensive documentation and supporting tools. It has been used in a variety of industrial applications as well as for research. For evaluation purposes, a free trial license for JAL can be obtained; more information is available at `http://www.agent-software.com`.

4 Hybrid Approaches

Various well-known agent languages combine declarative and imperative features. In this section we describe agent programming languages which are declarative while at the same time providing some specific constructs allowing for the use of code implemented in some external imperative language. These constructs serve as a means for the use of legacy code. The languages chosen to illustrate the hybrid approach are: 3APL, *Jason*, IMPACT, Go!, and AF-APL.

3APL (**A**n **A**bstract **A**gent **P**rogramming **L**anguage "**triple-a-p-l**") is a programming language for implementing cognitive agents that have beliefs, goals, and plans as mental attitudes, can generate and revise their plans to achieve their goals, and are able to interact with each other and with the environment they share with other agents. The first version of 3APL was designed by Hindriks et al. at Utrecht University [28]. Since its initial design, the 3APL programming language has been subject to continuous development [17, 16].

One of the main features of 3APL consists of programming constructs to implement mental attitudes of an agent as well as the deliberation process which manipulates them [15]. In particular, 3APL allows direct specification of mental attitudes such as beliefs, goals, plans, actions and reasoning rules. Actions form the basic building blocks of plans and can be internal mental actions, external actions, or communication actions. The deliberation-related constructs allow the implementation of selection and execution of actions and plans through which an agent's belief base can be updated and through which the shared environment can be modified. It also allows the selection and application of reasoning rules through which the plan base can be modified.

The 3APL programming language is designed so as to respect a number of software engineering and programming principles such as separation of concerns, modularity, abstraction, and reusability. It also supports the integration of Prolog (declarative) and Java (imperative) programming languages. Interested readers will find in the 3APL user guide (`http://www.cs.uu.nl/3apl`) a number of illustrative toy-problem applications such as the "blocks world", Axelrod's tournament, an English auction system, and the Contract Net protocol. 3APL has also been applied to the implementation of the high-level control of mobile robots. In particular, 3APL is being used for controlling the behaviour of SONY AIBO robots and to implement small-device mobile applications.

Jason is an interpreter, implemented by R.Bordini and J.Hübner, for an extended version of AgentSpeak(L), a logic-based agent-oriented programming language introduced by A. Rao in [43]. The language is influenced by the work on the Beliefs-Desires-Intentions (BDI) architecture and BDI logics [44]. The semantics of the extended language (which we call simply AgentSpeak), given by Bordini and colleagues, was recently revised and appears in [55]. The core of the interpreter available with ***Jason*** is in fact an implementation of that operational semantics. ***Jason*** is available *Open Source* under GNU LGPL at `http://jason.sourceforge.net` [6]. Although the documentation is available at that URL, the best mate-

rial for an overview of the work on Jason is [7].

Some of the features available in ***Jason*** are: (i) speech-act based inter-agent communication (and belief annotation of information sources); (ii) annotations on plan labels, which can be used by elaborate (e.g., decision-theoretic) selection functions; (iii) fully customisable (in Java) selection functions, trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting); (iv) straightforward extensibility (and use of legacy code) by means of user-defined "internal actions"; (v) a clear notion of a *multi-agent environment*, which is implemented in Java (this can be a simulation of a real environment, e.g., for testing purposes before the system is actually deployed). ***Jason*** has a simple IDE which is discussed in Section 5.

IMPACT is a system developed by Subrahmanian et al. [49], with the main purpose of providing a framework to build agents on top of heterogeneous sources of knowledge, i.e., to transform legacy code into agents that can communicate and act. To "agentise" such legacy code, IMPACT provides the notion of an agent program written over a language of so-called *code-calls*. A code-call can be seen as an encapsulation of whatever the legacy code is, represented logically through conditions and queries on the results produced by such code. These are used in clauses, that form agent programs, determining constraints on the actions that are to be taken by agents. Actions in IMPACT use some deontic notions such as agent actions being, at a certain time, "obligatory", "permitted", "forbidden", etc. Such agent programs and their semantics resemble logic programs extended with deontic modalities. The semantics is given by the notion of a *rational status sets*, which are generalisations of the notion of stable models in logic programming.

The IMPACT platform provides a number of features, including agent deployment over a network, registration of available agent services and yellow-page facilities. Information on the IMPACT platform is available at `http://www.cs.umd.edu/projects/impact/`. The framework has been extended to support also temporal or probabilistic reasoning [20]. A recent overview of the IMPACT language and platform can be found in [21].

Go! [12] is a multi-paradigm agent programming language, with a declarative subset of function and relation definitions, an imperative subset comprising action procedure definitions, and rich program structuring mechanism. Based on the symbolic programming language April [36], Go! extends it with knowledge representation features of logic programming, yielding a multi-threaded, strongly typed and higher order (in the functional-programming sense) language.

Inherited from April, threads primarily communicate through asynchronous message passing. Threads, executing action rules, react to received messages using pattern matching and pattern-based message reaction rules. A communication daemon enables threads in different Go! processes to communicate transparently over a network. Typically, each agent will comprise several threads, each of which can directly communicate with threads in other agents. Threads within a single Go! process, hence in the same agent, can also communicate by manipulating shared cell or dynamic relation objects. As in Linda tuple stores, these elements are used to coordinate the activities of different threads within an agent. Go! is strongly typed, which can often reduce the programmer's burden, and compile-time type checking improves code safety. New types can be declared and thereby new data constructors can be introduced.

The design of Go! took into consideration critical issues such as security, transparency, and integrity, in regards to the adoption of logic programming technology. Features of Prolog that lack a transparent semantics, such as the cut ('`!`') were left out. In Prolog the same clause syntax is used both for defining relations, with a declarative semantics, and for defining procedures which only have an operational semantics. In Go!, behaviour is described using action rules that have a specialised syntax.

Agent Factory Agent Programming Language (AF-APL) is the core programming language that resides at the heart of Agent Factory, which will be reviewed in Section 5. AF-APL is originally based on Agent-Oriented Programming as first put forward by Y.Shoham [48], but was revised and extended with BDI concepts, such as beliefs and plans. The syntax and semantics of the AF-APL language have been derived from a logical model of how an agent commits itself to a course of action. Details of this model can be found in [13, 46]. Specifically, the model defines the mental state of an agent to be comprised of two primary mental attitudes: beliefs and commitments. In AF-APL, the belief set is comprised of a set of declarations about the current state of the environment. Agents are *situated*, given that an AF-APL programmer can declare explicitly, for each agent, a set of sensors referred to as perceptors and a set of effectors known as actuators. Perceptors are realized as instances of Java classes that define how to convert raw sensor data into beliefs that may be added to the belief set of the agent. Similarly, an actuator is realized as an instance of a Java class, which has two responsibilities: (1) to define the action identifier that should be used when referring to the action that is realized by the actuator, and (2) to contain code that implements the action. Collectively, these declarations are termed the embodiment configuration of the agent, and they are specified within the agent program.

5 Integrated Development Environments

Integrated Development Environments (IDEs), focus on the programming language level and intend to enhance the productivity by automating tedious coding tasks. Looking at current IDEs from the object-oriented domain it can be

seen that such IDEs tend to provide functionalities that can be classified into five categories: *project management*, e.g. organising the project structure according to developers' needs; *creating and editing source files*, e.g. providing structure views for quick and easy navigation, online error detection, auto-completion, and so on; *refactoring* to enable fast and reliable code restructuring operations; *build and run process* allowing the execution of applications from within the IDE; and *testing*, e.g. supported by unit testing with test cases.

In the agent world, the situation differs from conventional programming in that there is no common ground with respect to agent programming languages and agent architectures. Hence, current agent IDEs exist only for agent languages of specific agent frameworks. Additionally, we found that only a small proportion of available agent frameworks offer IDE support at all, considering AgentLink (`http://www.agentlink.org`) as a representative selection of existing agent-related software. From this small number, we selected some representative examples: 3APL IDE, Jason IDE, JDE, CAFnE, Visual Soar, AgentBuilder, AgentFactory, and the Living Systems Developer.

The **3APL IDE** allows developers to load/edit 3APL programs that implement individual agents, execute one or more agent programs in either a step-by-step or continuous fashion, implement and configure the environment that is shared by the agents, monitor the internal state of individual agents through an agent property window, monitor the exchange of messages through the sniffer tool, send an external-user message to an individual agent, and read the system messages. The 3APL IDE is built on top of the 3APL multi-agent platform that consists of a directory facilitator called agent management system, a message transport system which delivers agent messages, and a plugin interface that allows agents to execute actions in the shared environment. The 3APL platform thus allows the implementation and concurrent execution of a set of 3APL agents. The 3APL development environment, its user guide, and further documentation can be found at `http://www.cs.uu.nl/3apl`.

Jason [6] is distributed with an IDE which provides a graphical interface for editing a multi-agent system configuration file, as well as AgentSpeak code for the individual agents. Through the IDE, it is also possible to run and control the execution mode of a multi-agent system, and to distribute agents over a network in a very simple way. The IDE also provides another tool, called "Mind Inspector", which allows the user to inspect agents' internal states when the system is running in debugging mode. This is very useful for debugging AgentSpeak MAS, as it allows the programmer to inspect agents' mental attitudes across a distributed system.

The JACK Development Environment (JDE) is a full-featured commercial IDE for the JACK BDI agent platform [57] developed by Agent Oriented Software Ltd. It is based on the JACK Agent Language (JAL) which was presented in Section 3. JDE allows agent developers to organise their files into projects offering a semantically organised tree view with respect to the different kinds of contained elements. The editing of agent code is supported by a rudimentary integrated editor that, for example, provides syntax highlighting for JAL. More advanced features such as auto-completion and error-detection are not available. However, the IDE provides a graphical plan editor that allows the construction of a plan from visual components similar to statecharts. Once the code base for a project is complete, it is possible to compile and run an application directly from within the IDE.

The **CAFnE** (**C**omponent **A**gent **F**ramework for **n**on-**E**xperts) tool [29] does not represent an IDE in the classical sense. Its objective is to enable domain experts to modify an existing agent application. CAFnE has been conceived to support the development of BDI agents based on a rather platform-independent BDI component language adapted from SMART [34]. The rationale of CAFnE is to hide the agent code layer and provide interactive dialogues for the development. Transformer modules can then be used to generate platform-dependent code from the internal representation.

Visual Soar is a freely available IDE for the Soar agent architecture [31]. It supports basic project management capabilities and mainly facilitates Soar agent programming through syntax highlighting and some consistency checking functionalities. Additionally, the IDE provides a connection to a Soar runtime environment allowing Soar agents to be executed from the IDE.

AgentBuilder is an agent platform directly based Agent-Oriented Programming (AOP), as originally defined by Shoham [48], developed by Acronymics Inc. It relies on the Reticular Agent Language which is an extension of Shoham's Agent0. As the used agent language is not intended for direct programming, an agent developer has to use the AgentBuilder IDE, which consists of a variety of different tools supporting all aspects of building agent applications. The IDE is conceived to hide agent code as much as possible and offers graphical wizards and tools whenever possible. It provides simple project management functionalities and integrates with a compiler tool. Successfully built agent applications can directly be executed from the IDE.

The **Agent Factory** [13] Development Environment offers support for basic project management, editing, and assembling the different agent constituents. It contains a cohesive layered framework for the development and deployment of agent-oriented applications. At the centre of this framework is the Agent Factory Agent Programming Language (AF-APL) described above in Section 4. The AF-APL interpreter is embedded within the distributed FIPA-compliant Run-Time Environment (RTE) which can be seen as a collection of agent platforms. Besides the IDE, a tool named VIPER [45] allows the composition of Agent UML Sequence Diagrams that sit at the heart of the Protocol Model. In addition to the tools that have been provided to support the development of AF-APL agents, the

Agent Factory Development Environment also includes a suite of tools that facilitate the testing and debugging of agent-oriented applications.

The **Living Systems Developer** is a commercial IDE for the Living Systems Technology Suite developed by Whitestein (`http://www.whitestein.com`). The underlying agent platform supports Java-based agents, rather than supporting a specialised agent language. The IDE is designed as an Eclipse (`http://www.eclipse.org`) plug-in, hence providing sophisticated editing and refactoring functionalities for Java code. In addition, several agent related aspects such as project management in accordance to the agent features used have been added. To facilitate the development process of agent-based applications, the IDE has been extended to fully support all phases of ADEM, the Agent Development Methodology also created at Whitestein.

6 Agent Platforms and Frameworks

Most languages described in this paper have some underlying platform which implements the semantics of the agent programming language. However, some implemented frameworks exist that are not so strongly tied to a particular programming language. Instead, these frameworks are more concerned with providing support for aspects such as agent communication and coordination. In this Section we focus on such frameworks, having chosen TuCSoN, JADE, and DESIRE as illustrative examples.

TuCSoN (**Tu**ple **C**entre **S**pread **o**ver the **N**etwork) is a framework for MAS coordination, based on a model and a related infrastructure providing general-purpose, programmable services for supporting agent communication and coordination [39]. The model is based on *tuple centres* as runtime programmable abstractions whose coordinating behaviour can be dynamically specified with a logic-based language called ReSpecT. Tuple centres are an example of coordination artifacts (see the survey on Environment modelling for MAS [56]), i.e, first-class entities (tools) populating the agent cooperative working environment, shared and used collectively by the agents to support their coordination. Such abstractions are also used in the SODA methodology (see the survey on Agent Oriented Software Engineering [4]) as basic building blocks for designing the social level and the environment in a MAS.

The TuCSoN technology is available as an open source project (`http://tucson.sourceforge.net`). It is completely based on Java, and is composed of: a runtime platform to be installed on hosts to turn them into nodes of the infrastructure; a set of libraries (APIs) to enable agents access to the services; and a set of tools mainly to support the runtime inspection and control (monitoring, debugging) of tuple-centres' state and coordinating behaviour. At the heart of the TuCSoN technology is the tuProlog technology, a Prolog engine fully integrated with the Java environment, available also as a standalone library and environment (the tuProlog technology is available at `http://tuprolog.sourceforge.net` [19]). Besides being adopted in research projects (e.g., for distributed workflow management, logistics, and e-learning), TuCSoN is currently used as one of the reference platforms for building agent-based systems in academic projects and thesis developed at the Engineering Faculties in Cesena and Bologna.

JADE (**J**ava **A**gent **DE**velopment Framework) [2] is a Java framework for the development of distributed multi-agent applications. It represents an agent middleware providing a set of available and easy-to-use services and several graphical tools for debugging and testing. One of the main objectives of the platform is to support interoperability by strictly adhering to the FIPA specifications concerning the platform architecture as well as the communication infrastructure. Moreover, JADE is very flexible and can be adapted to be used on devices with limited resources such as PDAs and mobile phones.

JADE has been widely used over the last years by many academic and industrial organisations (see [2]) ranging from tutorials for teaching support in agent-related University courses to Industrial prototyping. As an example, Whitestein has used JADE to construct an agent-based system for decision-making support in organ transplant centres [10].

The JADE platform is open source software, distributed by TILAB (Telecom Italia LABoratories) under the terms of the LGPL license and can be obtained at `http://jade.tilab.com`. Since May 2003, the International JADE Board has been responsible for supervising the management of the project. Currently, the JADE Board consists of five members: TILAB, Motorola, Whitestein Technologies AG, Profactor, and France Telecom.

Jadex [42] is a software framework for the creation of goal-oriented agents following the belief-desire-intention (BDI) model. The framework is realized as a rational agent layer that sits on top of a middleware agent infrastructure such as JADE [2], and supports agent development with well established technologies such as Java and XML. The Jadex reasoning engine addresses traditional limitations of BDI systems by introducing new concepts such as explicit goals and goal deliberation mechanisms (see, e.g., [8]), making results from goal-oriented analysis and design methods (e.g., KAOS and Tropos) more easily transferable to the implementation phase.

Jadex has been used to build applications in different domains such as simulation, scheduling, and mobile computing. For example, Jadex was used to develop a multi-agent application for negotiation of treatment schedules in hospitals [40]. Jadex has also been successfully used in several software engineering courses at the University of Hamburg.

The Jadex system, developed at the Distributed Systems and Information Systems group at the University of Hamburg, is freely available under the LGPL license and can be downloaded from `http://jadex.sourceforge.net`. Besides the framework and additional development

tools, the distribution contains an introductory tutorial, a user guide, and several illustrative example applications with source code.

DESIRE (**DE**sign and **S**pecification of **I**nteracting **RE**asoning components) is a compositional development method for multi-agent systems, based on a notion of compositional architecture, and developed by Treur et al. [9] at the Vrije Universiteit Amsterdam. In this approach, agent design is based on the following main aspects: process composition, knowledge composition, and relations between knowledge and process composition. In this component-based agent approach, an agent's complex reasoning process is built up as an interaction between the components representing the subprocesses of the overall reasoning process [9]. The reasoning process is structured according to a number of reasoning components that interact with each other. Components may or may not be composed of other components, where components that are not further decomposed are called primitive components. The functioning of the overall agent system is based on the functionality of these primitive components plus the composition relation that coordinates their interaction. Specification of a composition relation may involve, for example, the possibilities of information exchange among components and the control structure that activates the components. The DESIRE approach has been used for applications such as load balancing of electricity distribution and diagnosis systems. Further information and documentation of the tools supporting the development and implementation of multi-agent systems based on DESIRE is available at `http://www.few.vu.nl/~wai/demas/tools2.html`.

7 Further Reading

This paper should be complemented with related literature. Besides the references spread throughout the text, pointing to more detailed explanations of the systems described, we recommend the survey on agent programming languages by Mascardi et al. [35], which provides a detailed view of ConGolog, Agent-0, IMPACT, DyLog, Concurrent MetateM, and $\mathcal{E}_{hhf}$. A reference book on programming languages for Multi-Agent Systems has been published recently [5]. It contains detailed description of a selection of practical programming languages and tools which support MAS programming and implement key MAS concepts in a unified framework. Another extensive overview of agent technology is available in [3], which includes a comprehensive collection of papers on technologies, methodologies, and current research trends in the MAS domain.

As we have mentioned before, the criteria in which we based our choice of systems was, in part, the extent to which European researchers have contributed to their development. Of course there are various other agent languages, platforms, and tools besides those referred here. A good collection of agent-related software can be found in the AgentLink III website (`www.agentlink.org`).

Overall, the systems described here focus on the implementation phase. However, current research trends include the attempt to make implementation easier by bridging the analysis and design phase directly to implementation [4]. Examples of such research efforts are INGENIAS and its Development Kit [27] (`http://ingenias.sourceforge.net`), and MaSE and its AgenTool [18] (`http://macr.cis.ksu.edu/projects/agentTool/agentool.htm`).

8 Final Remarks

Programming Multi-Agent Systems is rapidly turning into a new discipline of its own. Throughout the paper, we have described several examples of languages and systems currently being developed in this area. We now draw some conclusions on the three main topics of this survey, namely languages, IDEs, and platforms.

Languages. Most research in agent-oriented programming languages is based on declarative approaches. There are many declarative solutions, most of them logic based. Purely imperative languages are unusual in the Agents literature, as in essence they are inappropriate for expressing the high-level abstractions associated with agent systems design. On the other hand, as we saw above, agent-oriented programming languages tend to allow for easy integration with (legacy) code written in imperative languages. Interestingly, the characteristics of the underlying agent architectures determine that it is often more appropriate to use interpreters rather than compilers.

IDEs. The existing IDEs provide basic support for project management, creating/editing files, and building/running the systems, but fail to support sophisticated features within all these categories. In addition, none of the agent IDEs covers aspects of refactoring and testing of agent applications. One reason for this is that, except for the Living Systems Developer, all IDEs have been developed from scratch and thus do not rely on existing reliable technology. In general, IDE support for developing agent-based systems is rather weak and the existing agent tools do not offer the same level of usability as state-of-the-art object-oriented IDEs. One of the main reason for this is the currently unavoidable tight coupling of agent IDEs and agent platforms, which results from the lack of agreement on an unified programming language for multi-agent systems. Another trend (observable in some of the IDEs), which is in contrast to object-oriented IDEs, is that they partly try to abstract away from the underlying programming language in favour of using graphical means of programming, such as wizards and statecharts.

Platforms. Closed frameworks such as DESIRE, strongly based on a platform, provide more complete solutions than others such as Jadex or TuCSoN. They usually offer an agent architecture and a system model, very useful for novel developers, together with the communication infras-

tructure and a range of robust services, such as directory facilitators, agent management services, and monitoring facilities. As a drawback, closed frameworks limit the development. For example, the design approach of the framework may not fit certain domain problems. Perhaps that is the reason why most researchers tend to use more open solutions. Currently, the most popular solution is to use JADE as underlying agent infrastructure combined with some other (higher-level) approach to program the agents' behaviour. When dealing with more general frameworks (rather than tied to a platform), their use (i.e., defining the agents that will run within it, together with the required services and resources) should be automated as much as possible, in part to free the developer from low-level details (e.g. location of the configuration files, their concrete syntax, etc.). Despite this, few existing frameworks have IDE support. Concerning the paradigm of communication used, there are several on offer, often being an important issue when choosing which framework to adopt. TuCSoN is representative of tuple-centred communication, JADE of message passing, and DESIRE of data flow among processes.

The various approaches mentioned along this survey indicate that there is still much work to be done. Among the major challenges faced by this research community are:

- The conception and development of specialised debugging tools, in particular for cognitive agent languages;
- The integration of agent tools into existing IDEs, rather than starting from scratch;
- The separation of MAS frameworks from agent platforms, so that each framework can be used for deploying systems on a variety of platforms.
- The dissemination of the MAS programming paradigm, so that programmers have a better understanding of its foundations as well as practical characteristics.

We believe that the recent developments surveyed here show a lively interest in this area of research. Despite the large number of open issues and challenges, we expect that the experience gathered in developing MAS with these tools will take us closer to a more mature programming paradigm. Arguably, this is one of the few concrete ways for allowing wider audiences to use in practice, and in a systematic way, the various techniques that the MAS research community has developed over the last two decades.

Acknowledgements

We gratefully acknowledge the help and support of AgentLink III, in particular its Technical Fora, which not only motivated the authors to work together in producing this joint survey, but also provided the conditions for much of the discussion that we used in this paper and indeed that will guide future work in this area of research. We also acknowledge the valuable comments and suggestions provided by the anonymous referees.

References

[1] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 50–61. Springer, 2002.

[2] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE — a java agent development framework. In Bordini et al. [5], chapter 5, pages 125–148.

[3] F. Bergenti, M.-P. Gleizes, and F. Zambonelli, editors. *Methodologies and Software Engineering for Agent Systems*. Kluwer, 2004.

[4] C. Bernon, M. Cossentino, and J. Pavon. An overview of current trends in european aose research. *Journal of Informatica*, 2005. In this volume.

[5] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, 2005.

[6] R. H. Bordini, J. F. Hübner, et al. ***Jason***, manual, release 0.7 edition, Aug. 2005. `http://jason.sf.net/`.

[7] R. H. Bordini, J. F. Hübner, and R. Vieira. ***Jason*** and the Golden Fleece of agent-oriented programming. In Bordini et al. [5], chapter 1, pages 3–37.

[8] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal representation for BDI agent systems. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems, second Int. Workshop (ProMAS'04)*, volume 3346 of *LNAI*, pages 44–65. Springer Verlag, 2005.

[9] F. Brazier, C. Jonker, and J. Treur. Principles of compositional multi-agent system development. In *Proceedings of Conference on Information Technology and Knowledge Systems*, pages 347–360. Austrian Computer Society, 1998.

[10] M. Calisti, P. Funk, S. Biellman, and T. Bugnon. A multi-agent system for organ transplant management. In A. Moreno and J. Nealon, editors, *Applications of Software Agent Technology in the HealthCare Domain*, pages 199–212. Birkhäuser Verlag, 2004.

[11] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.

[12] K. Clark and F. McCabe. Go! — a multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):171–206, 2004.

[13] R. W. Collier. *Agent Factory: A Framework for the Engineering of Agent-Oriented Applications*. PhD thesis, University College Dublin, 2001.

[14] S. Costantini and A. Tocchio. A logic programming language for multi-agent systems. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 1–13. Springer, 2002.

[15] M. Dastani, F. de Boer, F. Dignum, and J.-J. Meyer. Programming agent deliberation: An approach illustrated using the 3APL language. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, pages 97–104. ACM, 2003.

[16] M. Dastani, M. B. van Riemsdijk, F. Dignum, and J.-J. C. Meyer. A programming language for cognitive agents: goal directed 3APL. In M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Programming multiagent systems, first international workshop (ProMAS'03)*, volume 3067 of *LNCS*, pages 111–130, Berlin, 2004. Springer Verlag.

[17] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [5], chapter 2, pages 39–67.

[18] S. DeLoach. Analysis and design using MaSE and agenTool. In *Proceedings of Midwest Artificial Intelligence and Cognitive Science*. Miami University Press, 2001.

[19] E. Denti, A. Omicini, and A. Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Science of Computer Programming*, 2005. In press.

[20] J. Dix, S. Kraus, and V. Subrahmanian. Agents dealing with time and uncertainty. In M. Gini, T. Ishida, C. Castelfranchi, and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 912–919. ACM Press, 2002.

[21] J. Dix and Y. Zhang. IMPACT: a multi-agent framework with declarative semantics. In Bordini et al. [5], chapter 3, pages 69–94.

[22] A. El Fallah Seghrouchni and A. Suna. An unified framework for programming autonomous, intelligent and mobile agents. In V. Marik, J. Müller, and M. Pechoucek, editors, *Proceedings of Third International Central and Eastern European Conference on Multi-Agent Systems*, volume 2691 of *LNAI*, pages 353–362. Springer Verlag, 2003.

[23] A. El Fallah Seghrouchni and A. Suna. CLAIM: A computational language for autonomous, intelligent and mobile agents. In M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Programming Multiagent Systems, first international workshop (ProMAS'03)*, volume 3067 of *LNCS*, pages 90–110. Springer Verlag, 2004.

[24] A. El Fallah Seghrouchni and A. Suna. Programming mobile intelligent agents:an operational semantics. In *Proceedings of the International Conference on Intelligent Agent Technology*, pages 65–71. IEEE Computer Society, 2004.

[25] A. El Fallah Seghrouchni and A. Suna. Himalaya Framework: Hierarchical Intelligent Mobile Agents for building Large-scale and Adaptive sYstems based on Ambients. In T. Ishida, L. Gasser, and H. Nakashima, editors, *Proceedings of Massive Multi-Agent Systems workshop*, number 3446 in LNAI, pages 202–216. Springer Verlag, 2005.

[26] R. Evertsz, M. Fletcher, R. Jones, J. Jarvis, J. Brusey, and S. Dance. Implementing industrial multi-agent systems using JACK^{TM}. In *Programming multiagent systems, first international workshop (ProMAS'03)*, volume 3067 of *LNAI*, pages 18–48. Springer Verlag, 2004.

[27] J. Gomez-Sanz and J. Pavon. Agent oriented software engineering with INGENIAS. In V. Marik, J. Müller, and M. Pechoucek, editors, *Proceedings of the Third International Central and Eastern European Conference on Multi-Agent Systems*, volume 2691 of *LNCS*, pages 394–403. Springer Verlag, 2003.

[28] K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[29] G. Jayatilleke, L. Padgham, and M. Winikoff. Component agent framework for non-experts (CAFnE) toolkit. In R. Unland, M. Klusch, and M. Calisti, editors, *Software Agent-Based Applications, Platforms and Development Kits*. Birkhäuser Publishing Company, 2005.

[30] G. Klein, A. Suna, and A. El Fallah Seghrouchni. Resource sharing and load balancing based on agent mobility. In *Proceedings of International Conference on Enterprise Information Systems*, pages 350–355. ICEIS Press, 2004.

[31] J. F. Lehman, J. E. Laird, and P. S. Rosenbloom. A gentle introduction to Soar, an architecture for human cognition. In S. Sternberg and D. Scarborough, editors, *An invitation to Cognitive Science, vol. 4*. MIT Press, 1996.

[32] J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.

[33] J. A. Leite, J. J. Alferes, and L. M. Pereira. MINERVA — a dynamic logic programming agent architecture. In J.-J. Meyer and M. Tambe, editors, *Intelligent Agents VIII — Agent Theories, Architectures, and Languages*, volume 2333 of *LNAI*, pages 141–157. Springer, 2002.

[34] M. Luck and M. d'Inverno. *Understanding Agent Systems*. Springer Series on Agent Technology. Springer, 2nd edition, 2004.

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

[36] F. McCabe and K. Clark. April — agent process interaction language. In M. Wooldridge and N. Jennings, editors, *Intelligent Agents, ECAI-94 Workshop on Agent Theories, Architectures, and Languages*, volume 890 of *LNAI*, pages 324–340. Springer, 1995.

[37] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF, the OMG mobile agent system interoperability facility. In *Proceedings of Mobile Agents'98*, volume 1477 of *LNAI*, pages 50–67. Springer, 1998.

[38] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.

[39] A. Omicini and F. Zambonelli. Coordination for Internet application development. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[40] T. O. Paulussen, A. Zöller, A. Heinzl, A. Pokahr, L. Braubach, and W. Lamersdorf. Dynamic patient scheduling in hospitals. In M. Bichler, C. Holtmann, S. Kirn, J. Müller, and C. Weinhardt, editors, *Coordination and Agent Technology in Value Networks*. GITO, Berlin, 2004.

[41] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN 19, Department of Computer Science, Arhaus University, 1981.

[42] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In Bordini et al. [5], chapter 6, pages 149–174.

[43] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of Modelling Autonomous Agents in a Multi-Agent World*, number 1038 in LNAI, pages 42–55. Springer Verlag, 1996.

[44] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proceedings of International Conference on Multi Agent Systems*, pages 312–319. AAAI Press / MIT Press, 1995.

[45] C. F. B. Rooney, R. W. Collier, and G. M. P. O'Hare. VIPER: A visual protocol editor. In R. D. Nicola, G. Ferrari, and G. Meredith, editors, *Proceedings of the International Conference on Coordination Models and Languages*, volume 2949 of *LNCS*, pages 279–293. Springer Verlag, 2004.

[46] R. Ross, R. Collier, and G. O'Hare. AF-APL: Bridging principles and practices in agent oriented languages. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems, second Int. Workshop (ProMAS'04)*, volume 3346 of *LNCS*, pages 66–88. Springer Verlag, 2005.

[47] G. D. M. Serugendo, M.-P. Gleizes, and A. Karageorgos. Self-organisation and emergence in mas: An overview. *Journal of Informatica*, 2005. In this volume.

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

[49] V. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross. *Heterogenous Active Agents*. MIT-Press, 2000.

[50] A. Suna and A. El Fallah Seghrouchni. Adaptative mobile multi-agent systems. In *Proceedings of International Central and Eastern European Conference on Multi-Agent Systems*, LNAI, 2005. To appear.

[51] A. Suna and A. El Fallah Seghrouchni. A mobile agents platform: architecture, mobility and security elements. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems, second Int. Workshop (ProMAS'04)*, volume 3346 of *LNAI*, pages 126–146, New-York, 2005. Springer Verlag.

[52] A. Suna, C. Lemaitre, and A. El Fallah Seghrouchni. E-commerce using an agent oriented approach. *Revista Iberoamericana de Inteligencia Artifical*, 9(25):89–98, 2005.

[53] M. Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 2005. To appear.

[54] M. Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*, volume 33 of *Applied Logic Series*. Springer, 2005.

[55] R. Vieira, A. Moreira, M. Wooldridge, and R. H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. *To appear*, 2005.

[56] D. Weyns and T. Holvoet. On the role of environments in multiagent systems. *Journal of Informatica*, 2005. In this volume.

[57] M. Winikoff. JACK^{TM} intelligent agents: An industrial strength platform. In Bordini et al. [5], chapter 7, pages 175–193.