Providing Integrated Development Environments for Multiagent Systems

Simon Lynch[1] and Keerthi Rajendran[1]

[1] University of Teesside, Middlesbrough, United Kingdom
{s.c.lynch, k.rajendran,}@tees.ac.uk

Abstract. The computing industry has yet to take up agent technology as a new approach to software development. While other paradigms are supported by various tools, including generic IDEs, these are not well developed for agent-ware. Many agent platforms provide some form of IDE but these are platform-specific and are typically so tightly coupled to their agent platforms that they offer little re-use. There has been too little discussion about which tools an IDE should contain and few attempts to produce a generic IDE. In this paper, we identify two levels of abstraction requiring IDE tool support and draw on current research to categorise a set of generic tools for each level. We describe the reasons why existing MAS IDEs are coupled to their platforms and present an extendible software architecture which avoids this coupling. We build an IDE using this architecture and demonstrate its decoupling and extensibility by experimentation.

Keywords: MAS, IDE, debugging, deployment, software architectures.

1 Introduction

Writing agent-based software requires a varied set of skills and imposes a level of complexity beyond that experienced with other approaches to software construction [5, 6, 12]. One reason for this is that agent-based code is reactive - agents respond to messages and changes in their environment. This can make debugging agents more complex than debugging traditional systems. Other problems are introduced by the distribution and concurrency which is present in systems containing multiple agents [12] where processing is spread across agents and performance is as much a result of agent-agent interaction as it is a result of the data manipulations from individual blocks of program code.

Research into Multiagent Systems (MAS) has produced a choice of methodologies, design tools and platforms for deploying agents, yet the development of MAS in industry is still limited to a few research-supported implementations like those conducted by Agentis Software [2]. While these industrial implementations have demonstrated some benefits in using agents they have also highlighted a need for appropriate programmer support. Currently, tool support for general MAS technology is not sufficient for widespread adoption [3, 5] and practitioners suggest that the consolidation of a suitable set of tools and technologies for Agent Oriented Software

Engineering (AOSE) is one of the most important challenges for transferring the results of agent-based research into industry [3, 5, 13]. A complete set of AOSE tools would include those used for deployment and monitoring as well as those used early in the design phases of agent-based solutions.

As with other paradigms (like object-orientation) an IDE for MAS development is an important tool [2] but, while there has been some publication of ideas [6, 10, 11] and different MAS platforms provide varying levels of support [9, 12, 15, 16], there has not been enough discussion within the research community about the features a MAS IDE should contain and IDEs have, to date, only been developed as adjuncts to existing MAS languages and platforms. This is understandable: the first priority for researchers is to build the run-time platform itself since there is little point in having an IDE without a platform. IDE development becomes important only after the platform is operational and often only once a user group becomes established. The result of this trend is that existing IDEs are so tightly coupled to their target agent-languages and/or agent-platforms that they offer little opportunity for re-use.

Our initial work in IDE construction followed this approach. Furthermore, many of the tools in our original IDE were aimed at programming and debugging individual agents rather than higher level tasks like system-wide debugging and deployment. Providing tools for individual agents is less demanding because these tools can often use technologies borrowed from other paradigms (like objects). We were not alone in this, the lack of tool support at the MAS system level, mainly for post-implementation activities, has been recognised [12, 5, 6] and we have more recently focused on this. We have also acknowledged the need, discussed by Bordini *et al*. [5], to decouple agent tools from any specific agent framework and language. For example, the JACK Development Environment (JDE) is used with the JACK agent platform (www.agent-software.com) while the JADE toolset is used with the JADE framework [1] and these cannot be interchanged. We have taken a different approach to IDE development and believe we have achieved cross-platform reuse by constructing our IDE as a MAS in its own right. In this paper we describe our approach by considering two questions that underpin this work:

(i) are there common tools, across MAS platforms which can be included in a generic IDE and is it possible to classify these in any way?

(ii) how is it possible to build a generic IDE for cross-platform use so it may be modified/extended to take account of other platform-specific features?

2 Classification of Tools

This section considers a means of classifying tools for MAS development and identifies tools/features important for inclusion in a MAS IDE. Some of these are generic across applications while others are agent paradigm specific.

There are a wide range of agent types and associated concepts of agency: some research communities are mostly concerned with web-based software components with communication ability, other communities concentrate on platforms like Jade [1] and Midas [8] which implement agents but have an open view of the nature of these agents, others focus on specific types of agents using languages/platforms like 3APL

[7] and Jason [4] which support BDI agents. Typically, specialised agents are built on and supported by a more general agent-layer which provides facilities for message transmission, etc. (3APL and Jason are supported by Jade for example). The IDE presented in this paper integrates with platforms operating at the level of Jade but in doing so supports any languages/platforms deployed on top of them. As discussed in later sections, the IDE is itself developed as a MAS and the agents making up this MAS are also mid-level agents.

With respect to the reference model proposed by Modi *et al*. [17], the IDE will typically receive information from the framework layer but this information will describe activity within the agent layer. While we recognise the importance of the Reference Model, we are concerned with the *tasks* performed by agent developers rather than levels of agent implementation. This leads us to sub-divide activity in the Reference Model's agent layer since it involves both individual agents as well as interacting MAS. In this paper we consider a classification of tools from two different levels of abstraction: the system-level and the agent-level. Tools in each of these levels can be further classified into build-time or run-time tools according to their use,

(i) the system-level considers the semantic content of agent-agent messages but represents the agents themselves as black box entities. It is concerned with building MAS as opposed to building individual agents. System level tasks are similar to the high level tasks in more traditional paradigms (system wide testing, interoperability, deployment, etc). This level also captures additional MAS concepts like emergent behaviour, groups and teamwork. At run time, during monitoring and debugging phases, activities at this level are concerned with the emergent properties of a MAS which include the nature of agent-agent interactions and messaging, the system's architecture, etc.;

(ii) the agent-level, in contrast is concerned primarily with the functionality of individual agents and considers, for example, how they react to given messages. Internal workings of agents have focus and activities are those associated with building single agents. Agent-level tasks map on to the programming tasks performed in other paradigms (editing source files, testing individual classes, etc).

Our analysis reveals that system level tools are often generic while agent-level tools tend to match the agent language used and the type of agency involved. Categorising tools into these levels has allowed us to determine which tools can be provided by plugging in existing (programming language based) development and debugging environments (which operate at the agent-level) and which are the direct responsibility of a system-level IDE.

Previous research acknowledges that debugging agents is difficult and has recognised the need for tools like the Tracing method [12] but still debugging, especially at the system-level, has not received sufficient attention [9, 19] and needs further investigation. System-level debugging where individual agents are black boxes, involves activities like analysing scenarios of activity involving groups of agents and solving errors in the functioning of the system as a whole. Tools are also needed for system-level phases of deployment and post-implementation monitoring but many of these tasks are inadequately supported [3, 6].

Since system-level tools are often neglected, the term "MAS IDE" can be misleading with most current MAS IDEs only providing the kind of tools associated with object-oriented IDEs, i.e.: focusing on programming issues and serving primarily to "*automate tedious coding tasks*" [5]. In this paper we use the term MAS IDE to refer to a toolkit with provision for both the agent-level and the system-level.

2.1 Generic Systems Support

This section discusses generic tool requirements at the system level. They have been derived from MAS literature and through our experience with MAS development.

2.1.1 Representing MAS Structure

A simple MAS may be composed of a homogeneous collection of interacting agents in which all elements of the system can be presented as black box components with a messaging interface. With larger scale systems there may be benefits in viewing a MAS as a hierarchical organisation where collections of agents are grouped into higher level forms which can be viewed as single-entity black-boxes. Agents can be sensibly grouped for various reasons: because the agents they contain represent a single holon, because they are deployed on a single physical network, because they form well established societies or because the model of agency considers group relationships of some kind (the Aalaadin AGR model for example). Allowing users to expand and collapse agent collections into single entities allows them to view a MAS at different levels of abstraction. This aids system comprehension as well as tasks like debugging.

2.1.2 Messaging

MAS are conceptualised and designed as interacting entities; these interactions are a key aspect of the MAS workings. Developers of message oriented systems have traditionally used tools that aggregate all message transmissions for later examination. A survey of MAS IDEs shows that many do little more than this. Some of the more developed IDEs also provide useful functions like conversation tracking [10], verification of scenarios by matching messages against design [19] and the ability to restrict logs to messages relating to particular agents [1]. Some simply produce text files of data while others provide a more readable display.

Since analysing messages is so fundamental to MAS debugging, a general purpose MAS IDE should provide extensive and specialised support for viewing messages. We suggest the following as a minimal set of facilities:

- separation of agent messages from system messages (e.g.: error messages);
- collation of messages under different search criteria like the identities of sending/receiving agents, conversation id, time frames, physical distribution, etc;
- offline storage of all messages with flexible filtering mechanisms so that the user may inspect different subsets of messages by applying different filtering criteria;
- synchronisation with other functions like logging and playback (described next).

2.1.3 Logging and Playback

For analysis and debugging it is difficult to observe any system, including MAS, in real-time because system activity occurs too fast for human comprehension. Some traditional programming tools use features like breakpoints to pause execution so that current system states can be inspected. These facilities are important for troubleshooting but are impossible to use with distributed, concurrent systems. One solution is to capture MAS activity (e.g.: agents joining the MAS, messages and errors) and allow users to replay it at slower speeds and apply breakpoints on the replay mechanism. This can be supported by filtering mechanisms that allow the user to focus on particular agents, agent groups, parts of the agent network and interaction scenarios. These kinds of activity are currently not well supported although their importance has been highlighted for some time, for example Ndumu *et al*. [18] explain the importance of offline replay of MAS activity from different perspectives. In addition, offline playback allows developers some opportunity to visualise the emergent properties of a system as they occur since an IDE can show the changing system architecture, message transmissions, system errors, etc. slowing down and pausing the replay as necessary.

2.1.4 Testing

It is necessary to test either single agents or sets of agents without the presence of other agents that send or receive messages from them. This may be because related agents have not yet been developed or because the agent(s) are being developed as plug and play agents. An IDE that allows agent interactions to be driven manually (or through scripts) therefore offers some advantages. Some systems provide messaging agents for this (e.g.: JADE's "Dummy Agent", Mock Agents in Agile PASSI). These are useful for testing agents' internal response to messages, the interactions between agents and to identify emergent behaviour in an agent sub-group.

2.1.5 Deployment

MAS consist of independently executing entities and do not have a single starting point like other types of applications so launching them is more complex but this is only addressed by few platforms [6]. Toolkits that allow MAS launching information to be specified and also automate the launching process reduce the likelihood of errors and aid system reconfiguration. Information to support launching may include details such as agent instances (type and number of agents), locations of agent executable files, their dependencies (constraints on the order in which they are started up) and structure (hierarchical groupings).

There is also a need to remove agents or add new agents to a running MAS. Administrators may be required to monitor running MAS for conditions that need correction [6], this is more important for dynamic MAS whose composition and structure change frequently at runtime. Platforms produce messages indicating error and abnormal conditions. Some messages indicate failures that require agent repair or redundancy; others may indicate system-level conditions like high message traffic which may need correction by reconfiguring MAS structure. Administrators require facilities to change MAS structure at run-time by adding/removing agents or

relocating agents to other parts of the MAS network. Support for these tasks is also required during the testing and debugging of MAS.

3 Decoupling and Reuse

Choosing to use MAS as a paradigm for development and selecting a platform comes with the constraints of the agent language, philosophy and toolset associated with the platform. Such a "platform package" will tend to have leanings towards certain types of applications and types of agency rather than be generic/tailorable. If a developer selects 3APL they commit to using BDI agents and the 3APL IDE/toolkit. If they need to build mobile agents on portable devices for some other work they will change agent languages but in doing so they will be forced to discard all those tools they used with 3APL including the IDE. IDEs used in object oriented development are no longer like this, they can be readily reconfigured for different languages and linked to specific tools for those languages. This is not only true for well featured packages like Eclipse but is also the case with tools that are little more than editors. WinEdit for example (www.winedit.com) can be configured to color program code according to simple syntax rules and link to specific compilers. This kind of cross-platform reuse is not offered by IDEs for MAS.

Bordini et al. [5] identify a number of priorities to enable wider development of agent based software. In relation to practical MAS construction they highlight the need to integrate MAS development environments with existing object oriented IDEs and imply that a key challenge is also to develop a MAS IDE that can integrate across different MAS platforms but acknowledge that this is difficult currently since there is "unavoidabl[ly] tight coupling of agent IDEs and agent platforms" [ibid. p.40].

3.1 The Causes of Coupling

There are various factors which tend to increase the levels of coupling between components in software systems. These affect agent based software in similar ways to other types of software. In considering those coupling dependencies which occur between an agent platform and an IDE we identify three issues. The first relates to the nature of communication and information exchange between agent platform and IDE. If they use some unique mechanism for interaction or pass data which is structured in complex ways then their relationship will exhibit close coupling. In the worse case, if they communicate through a series of method calls and share internal data structures, they will effectively be using some common API and it will only be possible to reuse the IDE (or replace it) with other software built on the same API.

A second factor is the extent to which the IDE provides specific support for features unique to the platform (or perhaps handled in a unique way by the platform). This can be overcome by restricting the IDE so that it only supports those generic features which form part of all agent platforms but this would be a poor solution since the IDE would then implement only the small subset of features and fail to provide many of the tools that developers need. When working with mobile agents, for example, it is highly desirable for an IDE to manage aspects of mobility yet mobility

tools would probably not be included in a generic set of features. This apparently presents a conflict of interests since an IDE can only implement generic features if it is to offer cross-platform reuse and yet, if it is going to offer a comprehensive set of tools for developers, it must also provide platform specific tools.

Finally, a third factor which increases coupling dependency is the extent to which the design of the IDE is influenced by the design (or agent-paradigm) of any platform/agent language. This relates in part to the previous two factors (if the IDE shares an API with the platform or is influenced by platform specific features then coupling will be increased) but it may also be caused by less explicit dependencies. If, for example, it is assumed that agents are BDI and the IDE is built around this premise then the reporting of agent behavior may be in terms of "plans". This approach would limit reuse since the IDE would be less suitable for non-BDI agents.

3.2 Achieving Decoupling

The first requirement to limit coupling is that the IDE and the platform avoid communicating by uniquely defined method calls and avoid passing complex data structures which may not be appropriate for other platforms. It would be possible to achieve this with existing OO techniques (such as using a command pattern) but we also want an IDE which can be modified at run-time without a need for recompilation or rebuild (to provide enhanced reporting of agent mobility for example) and can be used, simultaneously, with multiple frameworks which may each provide different information (structure and content) to the IDE. After results of initial experimentation, in preference to using an OO approach, we have constructed the IDE in the form of its own, independent MAS. Communication to and from this *IDE-MAS* is sent textually in the form of inter-agent messages by following an agreed protocol for message structuring. In keeping with the principle that agents written in different languages, using different paradigms are able to communicate as long as they do so using some agreed protocol, an IDE deployed in the form of a MAS is less tightly coupled than one relying on some other means of communication. In this case the coupling is defined only in terms of the message protocol required by the IDE. Any platform wishing to use the IDE need only send the IDE messages about the platform's events (agents joining / leaving the system, agent-agent message passing, errors, etc).

This approach overcomes the first point discussed in the section above but does not address the paradox of how an IDE can provide platform-specific features and still be suitable for cross-platform reuse. We have addressed this by following the model used with object oriented IDEs which provide a generic set of OO tools but then allow specific language tools (compilers, etc) to be plugged in to them. In our case, since the MAS IDE is now in the form of its own MAS, these tools are plugged in by adding new agents to the *IDE-MAS*.

Initially then, the IDE provides only a generic set of tools independent of any agent and not specific to any particular notion of agency. Further tools are then freely added in the form of additional agents which are incorporated into the IDE. This approach is made possible by tightly encapsulating the IDE so that its internal architecture and agent composition is not visible externally and by using a flexible protocol for messaging between its agents.

3.3 Architecture and Message Protocol

The IDE is arranged as an organisation of agents who's internal structure is invisible to external systems and who's agents present a shared interface. In addition the internal agents are arranged so that all messages received from external systems are received by a single internal message-dispatch agent. In practice these externally generated messages are sent by some external MAS (or, more likely its supporting MAS platform) to report on events occurring in the external system. It is by virtue of these messages that the IDE is able to monitor the structures and behaviours occurring in the external system.

The IDE-internal message-dispatch agent forwards messages to other agents within the IDE according to the *message type* which is a facet of the IDE message protocol. All agents inside the IDE register their interests with the message-dispatcher by telling it which types of message they wish to receive. For example: the internal agent which dynamically shows the architecture of an external system will register an interest in messages containing information about agents joining and leaving the external system. There are a number of advantages to using a dispatch agent:

(i) it hides the internal agents to such an extent that, even though the composition of agents in the IDE may change, external agents remain unaffected by any changes. External agents are unaffected by changes even if the structure of the IDE changes at run-time, this allows IDE users to switch on/off agents while monitoring a live system;

(ii) new agents can be added to the IDE simply by registering them with the dispatcher and noting what types of messages they wish to be copied into. There is no requirement for further configuration;

(iii) it is possible to dispatch the same message to multiple intra-IDE agents.

The flexibility of the IDE is further improved by using an extendible message protocol which is only weakly specified. Developers will typically incorporate additional agents in the IDE to monitor or manipulate some specific feature of the external agent platform they are using. These additional IDE agents will often need different types of information from that required by those standardised, generic tools provided by the IDE by default. By implication they will need this additional information transmitted to them within the IDE messages.

The IDE message protocol is based on a simple slot-filler notation but allows extra information to be included in additional slots. The IDE recognises specific types of MAS event (agents joining/leaving the systems, messages sent between agents, etc) indicated by the use of tags in the message protocol. For example, a message indicating that an agent named "Sue" has joined the system will use a register-agent tag and appear something like...

```
((from  external-sender)
 (to    IDE-dispatch)
 (type  register-agent)
 (body  (name Sue))
```

The tags *from*, *to* and *type* identify the agents involved in the information transfer and the type of information sent. The *body* tag contains information specific to the

message type. As outlined above IDE dispatcher routes this message to its internal registry agent on the basis of the message type.

The protocol allows extra slots to be included in any message. For example, one platform tested in this study allows agents to have a scope indicating their visibility. "Global" agents are visible across an entire MAS, "local" and "internal" agents have restricted visibility. Messages indicating agents have joined the system have an extra slot when used with this platform, e.g.:

```
((from  external-sender)
 (to    IDE-dispatch)
 (type  register-agent)
 (body  (name   Sue)
        (scope  global))
```

The IDE provides generic tools to monitor the architecture of an external MAS, its messages and report on any errors. The generic tool to monitor agent messaging is agnostic about the ACL used by the external system, it simply displays the content of external messages in a textual form. Alternative IDE agents, oriented to specific ACLs can be substituted, these will provide better information by making more effective use of the data contained in the body of IDE messages.

As well as introducing new tags to pre-existing message types, The IDE message protocol can be extended by introducing new message types. If the IDE is being used with a platform which supports mobility for example, the protocol could be extended to allow the following format of message which states that the agent mobile3 has moved from node7 to node11.

```
((from  external-sender)
 (to    IDE-dispatch)
 (type  mobile-relocate)
 (body  (name   mobile3)
        (source node7)
        (dest   node11))
```

No changes to the IDE or its existing agents are needed in order to accommodate this new message type. All that is required is that the external system generates a message of type *mobile-relocate* when appropriate and sends them to IDE-dispatch.

So far our discussions have focused on system level tools, these are provided by including generic tools in the IDE which can be added to and replaced by more specific tools as required. It is also necessary to supply agent level tools. However, the nature of tools at the agent level is different to those at the system level and the ways that they can be provided by the IDE are also different. There are a range of approaches to agent implementation provided by various platforms. Broadly we consider two different categories:

(i) platforms where agents are based on extensions of existing languages (Java for example) and,

(ii) those based on specialised agent definition languages.

In the first case neither system-level concepts nor many aspects of agency are explicitly visible in the program code. Agent-level activities like debugging and

editing the code that defines an agent are similar to those carried out with code not involving agents and existing IDEs, editors, etc. provided for the base language are suitable. In the second case, where agents are defined in a specialised language, the language/framework has an obligation to its developer community to provide appropriate agent-level tools if it intends widespread use. Currently the tendency is either to plug-in to an existing IDE e.g.: The Living Systems Developer which uses Eclipse (www.eclipse.org) or to provide a specialised IDE (these typically follow the model set out by the object-oriented IDEs [5]).

Consequently, whether platform specific or not, agent-level tools can be provided in the form of a conventional IDE. The requirement for a platform-independent MAS-toolkit to provide agent-level tools is then only that it must allow a range of IDEs to be registered so agents of different types can be inspected, traced and edited using appropriate tools.

Building the IDE as a MAS makes it possible to link agent-level tools (editors, object-IDEs, etc) by simply adding an agent to the IDE-MAS which calls up the tool. Alteration to the internal structure of the IDE is invisible to any other sub-systems so does not require them to be modified.

4 Evaluation

Our primary interest is the extent to which the design approach of the IDE allows it to be decoupled from any particular agent platform thereby allowing it to be reused. To evaluate the design we have constructed an IDE based on the principles described above. We have examined two aspects in judging the level of decoupling achieved:

(i) the decoupling of the IDE from any specific agent platform;

(ii) the decoupling from other components.

The first allows the IDE to be used with different platforms, the second allows further tools to be integrated with the IDE. The IDE is implemented as a MAS which defines a text-based message protocol and can link to any other software capable of socket-based communication. The IDE itself is not dependent on any platform for gathering system information. This suggests that the level of coupling is low but we have also demonstrated this experimentally in the following ways:

(i) the IDE has been used successfully with a MAS platform supporting agents written in Java and in Lisp [14];

(ii) while Galaxy Communicator (http://communicator.sourceforge.net/) should perhaps not be considered a true MAS, the IDE has been successfully used with Galaxy;

(iii) the IDE has been successfully used as a link between a MAS framework and Galaxy (readily achieved since the IDE can communicate with both systems);

(iv) the IDE has been used with a virtual MAS – a shell masquerading as a running MAS which, through the use of scripts, generates agent architectures and messages which are passed to non-virtual agents.

Since the IDE is deployed in the form of a MAS, providing new MAS-level tools is readily achieved by adding new agents to the IDE. This has been verified experimentally. Similarly, we have demonstrated by experimentation that the IDE can integrate with agent-level tools. In practice this is achieved by allowing agents to be inspected by different object-IDEs, according to their type. This capability is provided in the same way that a general purpose editor can call up appropriate compilers for the programs it is editing.

The IDE has also been tested for usability by following small user-groups of students involved in MAS development. Observations confirmed the importance of MAS tools in general (also noted by other authors [11, 12, 16]) and supported our proposition that the two levels of development and debugging for MAS, the agent-level and the system-level, involve different types of activity, the first where the focus of attention relates more to issues concerning program code, the second where the focus is systems architecture and messaging. Users reported a perceived reduction in the learning curve associated with moving to a new MAS platform while retaining the same IDE and highlighted the benefit of adopting their own personal preference of agent-level tools (eg: Eclipse).

5 Conclusion

This paper acknowledges that the lack of a complete set of engineering tools is a contributing factor to the slow uptake of agent-based software development in industry. In particular we have focused on the provision of IDEs.

We have categorised suitable tools for MAS IDEs into two levels of abstraction the agent-level and the system-level. We suggest that tools for use at the agent-level are generally available but there is a greater problem in providing tools at the system-level. Furthermore, we note that for tools to be of general use they must be decoupled from any particular agent platform.

We have presented a generalised set of requirements for system wide tools irrespective of the agent platform used. This minimal set of tools comprises facilities to inspect, monitor and debug a running MAS, it is intentionally neutral on framework specific aspects like types of agency, the structure of messages, mobility, etc.

We have tested an approach to IDE construction in which the IDE is built in the form of its own MAS. This approach decouples the IDE from any particular agent platform i.e.: the agents it monitors can be built on different agent platforms. We have evaluated this resulting IDE by experimentation and found that it can be extended by adding new agents to provide additional system-level tools and easily linked to existing agent-level tools like editors and inspectors. In addition we have succeeded in using the IDE as a bridge between two, otherwise incompatible, agent frameworks.

While more discussion is needed within the research community to determine which system-level tools are most appropriate and how they should be presented, we believe that it is possible to deploy these as an extendible, generic and platform-independent IDE.

References

1. Bellifemine, F. L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley. (2007)
2. Benfield, S. S., Hendrickson, J., Galanti, D.: Making a Strong Business Case for Multiagent Technology. In: AAMAS (Hakodate, Japan), ACM Press, New York. (2006)
3. Bernon, C., Cossentino, M., Pavon, J.: An Overview of Current Trends in European AOSE Research. Informatica, 29, 379—390. (2005)
4. Bordini, R., Hübner, J.F., Vieira.: R. Jason and the golden fleece of agent-oriented programming. In Multi-Agent Programming: Languages, Platforms and Applications, Kluwer. (2005)
5. Bordini, R., Braubach, L., Dastani, M., Seghrouchni, A.E.F., Gomez-Sanz, J.J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A Survey of Programming Languages and Platforms for Multi-Agent Systems. Informatica, 30(1), 33--44. (2006)
6. Braubach, L., Pokahr, A., Bade, D., Krempels, K., Lamersdorf, W.: Deployment of Distributed Multi-Agent Systems. In: M.-P. Gleizes, A.Omicini, F.Zambonelli (eds.) ESAW 2004, LNAI, vol. 3451, pp. 261--276. Springer, Heidelberg. (2005)
7. Dastani, M., Meyer, J.Ch.: A Practical Agent Programming Language. ProMAS, Hawaii. (2007)
8. Filho, A., H., Antonio do Prado, H., Pereira de Lucena, C. J.: A WSA-Based Architecture for Building Multi-Agent Systems. AAMAS, Hawaii. (2007)
9. Flater, D.: Debugging agent interactions: a case study. ACM Symposium on Applied Computing (Las Vegas, Nevada, United States). ACM Press New York, NY. (2001)
10. Fonseca, S. P., Griss, M. L., Letsinger, R.: Agent Behavior Architectures: A MAS Framework Comparison. AAMAS, Bologna, Italy, ACM Press, New York, NY. (2002)
11. Gutknecht, O., Ferber, J., Michel, F.: Integrating Tools and Infrastructures for Generic Multi-Agent Systems. Fifth International Joint Conference on Autonomous Agents Montreal, Canada, ACM Press, New York, NY. (2001)
12. Lam, D. N., Barber, K. S.: Verifying and Explaining Agent Behaviour in an Implemented Agent System. AAMAS, New York, USA, ACM Press, New York, NY. (2004)
13. Luck, M., McBurney, P., Shehory, O., Willmott, S.: Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing), AgentLink, ISBN 0854328459. (2005)
14. Lynch, S.C., Rajendran, K.: Boris-A Framework for Developing Multi-Agent Systems in Lisp and Java, International Lisp User Group Meeting, New York, USA. (2003)
15. Lynch, S.C., Rajendran, K.: Breaking Into Industry: Tool Support for Multiagent Systems. AAMAS, Hawaii. (2007)
16. Massonet, P., Deville, Y., Neve, C.: From AOSE Methodology to Agent Implementation. AAMAS, Bologna, Italy, ACM Press, New York, NY. (2002)
17. Modi, P. J., Mancoridis, S., Mongan, W., M., Regli, W., Mayk, I.: Towards a reference model for agent-based systems. AAMAS, Japan, ACM Press, New York, NY. (2006)
18. Ndumu, D. T., Nwana, H.S., Lee, L.C., Collis, J.C.: Visualising and debugging distributed multi-agent systems. Conference on Autonomous Agents, Seattle, USA, ACM Press, New York, NY. (1999)
19. Poutakidis, D., Padgham, L., Winikoff, M.: Debugging Multi-Agent Systems Using Design Artifacts: The Case of Interaction Protocols. AAMAS, Bologna, Italy, ACM Press, New York, NY. (2002)