JACK Intelligent Agents™ – Summary of an Agent Infrastructure

Nick Howden, Ralph Rönnquist, Andrew Hodgson, Andrew Lucas
Agent Oriented Software Pty. Ltd.
221 Bouverie St, Carlton
Victoria 3053, Australia
+61 3 9349 5055
http://www.agent-software.com
<firstname.surname>@agent-software.com

ABSTRACT

Intelligent Agents are being used for modelling simple rational behaviours in a wide range of distributed applications. In particular, multi-agent architectures based on the Belief-Desire-Intention model have been used successfully in situations where modelling of human reasoning and team behaviour are needed, such as simulating tactical decision-making in defence operations and command and control structures. Other applications include intelligent web applications, manufacturing control, telephone call centres, and air traffic management.

The JACK Intelligent Agents™ framework by Agent Oriented Software brings the concept of intelligent agents into the mainstream of commercial software engineering and Java. JACK Intelligent Agents™ is a third generation agent framework, designed as a set of lightweight components with high performance and strong data typing.

We present the design approach and major technical characteristics of JACK Intelligent Agents™, with a focus on some of the more recent developments – modular structuring elements called *Capabilities* and the team-based reasoning model called *SimpleTeam*. Two example applications, in decision support and defence simulation are described. Also, we discuss the benefits of the component-based approach, both for the software engineer developing sophisticated distributed applications, and for the researcher exploring agent models and architectures.

Keywords

Intelligent Agents, Infrastructure, JACK, BDI, Java, Multi-agent Systems.

1. INTRODUCTION

Intelligent Agents are being used for modelling simple rational behaviours in a wide range of distributed applications. Intelligent agents have received various, if not contradictory, definitions; by general consensus, they must show some degree of autonomy, social ability, and combine pro-active and reactive behaviour [1]. One of the better known and most successful architectures for agents is the so-called BDI (Belief-Desire-Intention) architecture, which has seen a number of academic and industrial applications.

Agent Oriented Software Pty. Ltd. (AOS), based in Melbourne, Australia, has built JACK Intelligent Agents™, a framework in Java for multi-agent system development. The company's aim is to provide a platform for commercial, industrial and research applications. To this end, its framework supplies a high performance, lightweight implementation of the BDI architecture, and can be easily extended to support different agent models or specific application requirements. For brevity, we will refer to JACK Intelligent Agents™ simply as 'JACK'.

This paper is organised as follows. Section 2 introduces JACK Intelligent Agents™, presenting the approach taken by AOS to its design, and outlining its major engineering characteristics. The BDI model is discussed briefly in Section 3. Sections 4 & 5 describe some of the more interesting components of the JACK infrastructure and Section 6 gives some example applications built with JACK Intelligent Agents™. The JACK components and tools and the future research and development direction is described in Sections 7 & 8. Finally, in Section 9 we discuss how the use of this framework can be beneficial to both engineers and researchers.

2. JACK Intelligent Agents™

In this section, we present JACK by first highlighting the goals set by its designers, then we provide an overview of the engineering characteristics of the framework.

2.1 Approach

Major design goals for JACK were:

- to provide developers with a robust, stable, light-weight product;
- to satisfy a variety of practical application needs;
- to ease technology transfer from research to industry; and
- to enable further applied research.

Whilst applications can be built from the ground up adopting an agent oriented methodology and an appropriate framework, most organisations already possess and depend upon large legacy software systems. Thus, JACK agents have been designed specifically for use as components of larger environments. Consequently, an agent must coexist and be visible as simply another object by non-agent software. Conversely, a JACK programmer must be allowed to easily access any other component of a system. Type safeness when accessing data, reliability, and support for proper engineering processes are then key requirements in this kind of environment.

For similar reasons, JACK agents are not bound to any specific agent communications language. Nothing prevents the adoption

of high-level symbolic protocol such as KQML or FIPA's Agent Communication Language (ACL); possibly by integrating software already existing in the public domain. However, JACK has been geared towards industrial object-oriented middleware (such as CORBA) and message passing infrastructures (for instance, HLA or DIS in simulation environments). In addition, JACK provides a native lightweight communications infrastructure for situations where high performance is required.

JACK itself has been designed for extension by properly trained engineers, familiar with agent concepts and with a sound understanding of concurrent object-oriented programming.

2.2 Overview of the framework

From an engineering perspective, JACK consists of architecture-independent facilities, plus a set of plug-in components that address the requirements of specific agent architectures. The plug-ins supplied with version 3.0, released in February 2001, include support for the BDI model, and the model for building teams of agents called *SimpleTeam*.

To an application programmer, JACK currently consists of three main extensions to Java. The first is a set of syntactical additions to its host language. These additions, in turn, can be divided as follows:

- a small number of keywords for the identification of the main components of an agent (such as *agent*, *plan* and *event*);
- a set of statements for the declaration of attributes and other characteristics of the components (for instance, the information contained in beliefs or carried by events). All attributes are strongly typed;
- a set of statements for the definition of static relationships (for instance, which plans can be adopted to react to a certain event);
- a set of statements for the manipulation of an agent's state (for instance, additions of new goals or sub-goals to be achieved, changes of beliefs, interaction with other agents).

Furthermore, the programmer can use Java statements within the components of an agent.

For the convenience of programmers, in particular those with a background in Artificial Intelligence, JACK also supports logical variables and cursors. These are particularly helpful when querying the state of an agent's beliefs. Their semantics is mid-way between logic programming languages (with the addition of type checking Java style) and embedded SQL.

The second extension to Java is a compiler that converts the syntactic additions described above into pure Java classes and statements that can be loaded with, and be called by, other Java code. The compiler also partially transforms the code of plans in order to obtain the correct semantics of the BDI architecture.

Finally, a set of classes (called the kernel) provides the required run-time support to the generated code. This includes:

- automatic management of concurrency among tasks being pursued in parallel (*Intentions* in the BDI terminology);
- default behaviour of the agent in reaction to events, failure of actions and tasks, and so on; and
- native lightweight, high performance communications infrastructure for multi-agent applications.

Importantly, the JACK kernel supports multiple agents within a single process, multiple agents across many processes, and any mix of the two. This is particularly convenient for saving system resources. For instance, agents that perform only short computations or share most of their code or data can be grouped together.

A JACK programmer can extend or change the architecture of an agent by providing new plug-ins. In most cases, this simply means overriding the default Java methods provided by the kernel or supplying new classes for run-time support. However, it is possible to add further syntactic extensions to be handled by the JACK compiler.

Similarly, a different communications infrastructure can be supplied by overriding the appropriate run-time methods.

Future versions of JACK will extend the base BDI model with new plug-ins and will add a number of development and monitoring tools.

3. Belief-Desire-Intention Agents

The BDI agent model supported by JACK has its roots in philosophy and cognitive science, and in particular in the work of Bratman on rational agents [2]. A rational agent has bounded resources, limited understanding and incomplete knowledge of what happens in the environment in which it lives. Such an agent has beliefs about the world and desires to satisfy, driving it to form intentions to act. An intention is a commitment to perform a plan. In general, a plan is only partially specified at the time of its formulation since the exact steps to be performed may depend on the state of the environment when they are eventually executed. The activity of a rational agent consists of performing the actions that it intended to execute without any further reasoning, until it is forced to a revision of its own intentions by changes to its beliefs or desires. Beliefs, Desires and Intentions are called the mental attitudes (or mental states) of an agent.

Observe that BDI agents depart from purely deductive systems and other traditional Artificial Intelligence models because of the concept of intentionality, which significantly reduces the extent of deliberation required. BDI has demonstrated itself to be well suited to modelling certain types of behaviour, such as the application of standard operational procedures by trained staff. It has been successfully adopted in fields as diverse as simulation of military tactics, applications of business rules in workflows, and diagnostics in telecoms networks.

Based on previous research and practical application, Rao and Georgeff [3] have described a computational model for a generic software system implementing a BDI agent. Such a system is an example of event-driven programs. In reaction to an event, for instance a change in the environment or its own beliefs, a BDI agent adopts a plan as one of its intentions. Plans are precompiled procedures that depend on a set of conditions for being applicable. The process of adopting a plan as one of the agent's intentions may require a selection among multiple candidates.

The agent executes the steps of the plans that it has adopted as intentions until further deliberation is required; this may happen because of new events or the failure or successful conclusion of existing intentions.

A step of a plan can consist of adding a goal (that is, a desire to achieve a certain objective) to the agent itself, changing its beliefs,

interacting with other agents, and any other atomic action on the agent's own state or the external world.

The abstract BDI architecture has been implemented in a number of systems. Of these, two are of particular relevance to JACK since they represent its immediate predecessors. The first generation is typified by the Procedural Reasoning System (PRS) [4], developed by SRI International in the mid-1980s. dMARS [5], built in the mid-1990s by the Australian Artificial Intelligence Institute in Melbourne, Australia, is a second generation system. dMARS has been used as a development platform for a number of technology demonstrator applications, including simulations of tactical decision-making in air operations and air traffic management.

4. Team Oriented Programming in JACK

In addition to the BDI extension, JACK provides an extension to support Team Oriented programming, called *SimpleTeam*. Team Oriented programming is a nuance of Agent Oriented programming wherein agent collaboration is specified from the abstract viewpoint of the group as a whole. The concept behind this approach is that coordinated behaviour is specified or programmed from a high-level ('bird's-eye') perspective and that the underlying machinery maps such specifications into the individual activities of the agents concerned.

Within Artificial Intelligence research, teamwork as an agent programming activity has been studied since the early 1990s [6], and is a rapidly developing field. Many different theories and types of teams (ranging from strictly hierarchically structured teams to collaborating teams without formal structure) have been proposed in the literature. Also, theories have been proposed regarding mutual beliefs and goals, where individual members of a team attempt to achieve what they believe the team as a whole is attempting to achieve.

The SimpleTeam extension is neutral to the nature of the structure of a team (i.e. hierarchical and imposed 'from the top', or resulting from spontaneous collaboration 'from the bottom'), and to how team formation is achieved. The only assumption is that it is possible to classify the members of a team in terms of abstract roles. The goal of SimpleTeam is to provide a software infrastructure for the specification of coordinated behaviour, which can then be used for pursuing applied studies on social organisation.

SimpleTeam supports the most valuable and practical aspect regarding teams: the centralised specification of coordinated behaviour, and its realisation through actual coordinated activity. To achieve this, it adds several concepts to the base JACK kernel, including *Team* & *Role*. It also adds several statements to the JACK language to allow the management and manipulation of these concepts by the programmer, including statements to handle parallelism, team plans, and team capabilities.

5. Modular Development & Software Reuse

Development of complex agent-based systems requires conceptual and software tools that allow modular development and software reuse. A substantial amount of experience on this topic has been acquired by the design and development team at Agent Oriented Software during the implementation and use of JACK in customer projects. The most important issues that have been faced include: the definition of a methodology that enables incremental development of agents and a high level of code reuse; and, what type of tools best support such a methodology.

Originating from that experience, the concept of *Capabilities* was developed and added to JACK. Capabilities represent a cluster of functional components that can be used as building blocks during the development of agent systems; they bring sound software engineering practices to agent development.

Capabilities are used as an integral part of the agent design methodology. They allow the manipulation of high level packages of agent functionality from the specification stage, through the design stages and onto direct support within the JACK code.

6. Applications using JACK

Below we describe two applications that have been developed with JACK; one in decision support and another in Defence simulation for analysis.

6.1 Decision Support

The Intelligence Preparation for the Battlefield (IPB) process in the Australian Army is similar to that of the US Army and UK forces — a universal approach to managing information on the battlefield. DSTO's Land Operations Division has a long-term program to provide tools to the Army to assist with the IPB process, and to provide a higher level of situation awareness to the field commander. A key step in this process is the allocation of surveillance and reconnaissance assets to provide timely, relevant information on the opposition's intentions and movements. Currently, the generation of Information Collection Plans relies on human planners working without computer assistance. Having received the Commander's Critical Information Requests (CCIRs), the staff planners must then decide on the options they have to conduct surveillance and reconnaissance. They must take into account a large number of factors, including:

- The number and priority of the CCIRs.
- The nature of the surveillance/reconnaissance necessary to satisfy a CCIR. For example, an UGS (unmanned ground sensor array able to detect movement) would be able to detect an intrusion, but could not confirm this without the need for further reconnaissance.
- The surveillance assets at the disposal of the planners, including their capabilities, mobility, current commitments, scarcity and vulnerability.
- The ongoing availability of assets to conduct surveillance, taking into account other priorities, maintenance requirements, etc.
- The developing tactical situation during the planning process. A plan that does not consider this is out of date before it can be considered.

The outcome is a number of optional plans that can be evaluated in collaboration with the commander, and a decision can then be taken to implement one. As the plan is then put in place and the assets deployed, the evolving awareness of changes in the situation will result in the plan being modified, often several times. As these changes occur, the planners have the difficult task of rapidly re-deploying assets as efficiently as the circumstances permit.

Consequently, the Collection Plan Management System (CPMS) [7] is being developed in Land Operations Division, as a tool to

improve situation awareness through better management of surveillance and reconnaissance assets. CPMS is comprised of the following major components.

- The Planning Module, (Figure 1) based on JACK agents.
- A visualisation module containing a Geographic Information System (GIS), to provide the necessary terrain analysis and a variety of custom Graphical User Interfaces (GUIs). These GUIs interact with the GIS map overlays to provide an intuitive user interface for planning and information dissemination.
- A database module containing terrain data, the Commander's Critical Information Requests (CCIRs) and the surveillance assets to be tasked, such as their capabilities, availabilities, etc.

The Planning Module takes a list of CCIRs, the Order of Battle (ORBAT) containing the Command and Control structure and available assets for tasking as input, and presents a set of alternative collection plans as output to the commander. Each alternative plan is a suggestion of how the available surveillance and reconnaissance assets may be used to service the input CCIRs in a collectively 'best' way, on the basis of a given range of evaluation dimensions. The design of the Planning Module recognises that there is rarely a single, optimal collection plan to be formed, and, in any case, a pre-programmed evaluation will always be incomplete with respect to an actual situation.

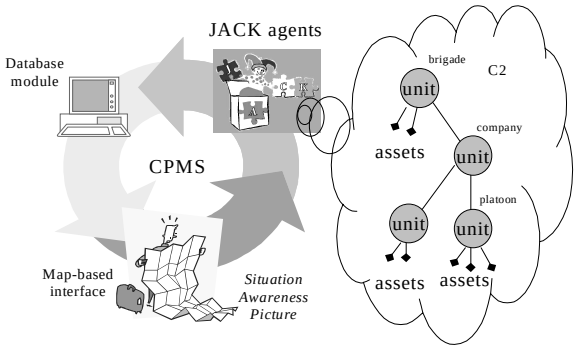

Figure 1: The CPMS Planning Module uses JACK Intelligent Agents™

The collection plans presented are formed in the context of the current C2 structure including the current surveillance assets and their status. This background information is part of the awareness picture, and can be overlayed with manual variations; for instance, for investigating what-if scenarios. The background information further includes a current surveillance plan as input, and considers the cost of re-tasking already tasked assets in forming alternative plans.

As noted above, the CPMS planning module is implemented as a multi-agent sub-system, which directly reflects the current C2 structure. There is one agent for each 'node' in the C2 structure, and each agent plans for the assets that the unit it represents possesses or controls. This planning is then combined with the plans suggested by the agents planning for subordinate units. This design approach was chosen as a means to minimise or avoid the conceptual gap between a software implementation and the end-user's understanding of the planning process.

Due to the inherent algorithmic complexity of the problem, the planning agents operate with lists of task options for CCIRs, dealing with each CCIR independently. The alternative tasks within a list are compared with respect to several evaluation dimensions, including task duration, quality of outcome, and a qualitative cost measure. The list is then ordered with preferred asset tasks at the head of the list. The top-level agent receives the full set of task lists, and processes this set in order to produce the alternative collection plans. In this process, the agent applies all inter-dependency constraints between the tasks: common-sense constraints (e.g. that an indivisible asset cannot be at more than one place at a time); doctrinal constraints (e.g. limiting the dispersion of a unit's assets). Consistent with the given inter-dependency constraints, the agent then generates plans that include the best task for each CCIR under the given task comparison dimensions.

6.2 Modelling Human Behaviour in Defence Simulation

Historically, computer simulation in Defence has been used for the evaluation of acquisitions and of force development options. But modelling and simulation for this purpose is becoming increasingly complex as multi-role, multi-platform and multi-system aspects are taken into consideration. The complexity of this task is further increased by the difficulty in modelling human decision-making with sufficient fidelity using conventional software approaches. Current implementations of Computer Generated Forces within Simulations such as *CAEN* or *ModSAF* have proven to be very useful, but do not model human reasoning and cannot easily model team behaviour. Early applications of intelligent agents in simulations to represent operational military reasoning have proved highly effective. This success comes from the capability of agents to represent individual reasoning and from the architectural advantages of that representation to the user due to the ease of setting up and modifying operational reasoning or tactics for various studies. In addition, JACK SimpleTeam extends the modelling of reasoning to explicitly model the communications and coordination of joint activities required for team behaviour.

The emphasis on timely, accurate information in modern warfare, and the availability of modern communications, have led to the development of more and more complex command and control systems. It is important to understand the behaviour of these C3 systems under a variety of circumstances. However, as they are difficult to analyse manually, advanced modelling and simulation tools for C3 systems development are required. The challenge in C3 systems is to model the reasoning associated with different roles in the command and control hierarchy. Intelligent agents can represent the reasoning and command capabilities associated with their assigned roles in the hierarchy, allowing different command and control strategies to be quickly evaluated under varying circumstances. This power comes from the suitability of the JACK architecture for representing individual and team objectives and roles.

DSTO's Land Operation Division and Agent Oriented Software have developed a Simulation Agent Infrastructure (SAI), which connects JACK Intelligent Agents™ into the Close Action Environment simulator (CAEN). The purpose is to enhance the war-gaming capability offered by CAEN with agent based behaviour modelling, so as to simplify and speed up the

simulation scenario development process and as a result be able to provide better analysis in a shorter time.

The SAI offers a modern war-gaming solution with a clean-cut separation between simulation models, simulation engine, and simulation scenario, which increases the reusability and maintainability of the software as well as verifiability of individual scenarios.

With CAEN alone, each war game is a fully scripted scenario in which the activity of each entity is pre-programmed in isolation with respect to the clock. It offers only a trivial level of situation awareness, such that an entity may fire or not fire his weapon depending on whether or not another entity is sighted. More complex behavioural variations, such as choosing where or whether to cross a road, cannot be expressed within the one scenario. This means that effectiveness studies that involve variations in the tactics are practically impossible.

The introduction of JACK Intelligent Agents™ and SimpleTeam within SAI adds the capability of modelling entity and group behaviour based on situation awareness. By this, it becomes feasible and even easy to express tactics where entity activity is determined on the basis of the actual situations occurring.

DSTO and Agent Oriented Software are currently working on generalising SAI to be used with other simulation systems such as *ModSAF* and *Stage*.

Although SAI was originally developed for acquisition and tactics analysis in Defence simulation, it can be applied equally well as a model for Computer Generated Forces within Civilian and Defence training systems, and even within commercial gaming environments.

Within Defence, the contemporary trend towards the integration of multi-role forces, together with the high cost of live exercises, has required the development of more realistic training environments. However, these synthetic environments have not been able to model the behaviour of the humans involved, other than in a very simple manner. In particular, they have not modelled team behaviour, with the result that trainees quickly learn the range of simulated behaviours. Rather than practising their military skills, instead they learn to predict the training system's response. JACK agents allow the Computer Generated Forces in training systems to behave in a more human-like manner, with a much richer set of behaviours, including team responses and dynamic role re-allocation. The result is a more effective training environment with realistic tactical behaviour represented, whilst avoiding the expense of having humans-in-the-loop involved to provide this.

7. JACK Components and Tools

JACK Intelligent Agents™ is distributed as the following modules:

- **JACK Runtime Environment**: The kernel supports the execution of JACK agents, handling multi-threading and concurrency issues, etc. It also includes a communications layer with a simple agent naming service.
- **JACK Compiler**: This compiles the JACK Agent Language code into pure Java, and calls the Java compiler to generate executables.
- **BDI Agent Model**: This adds support for BDI reasoning, with additions to the language syntax and the runtime kernel.
- **SimpleTeam Model**: This adds support for team-based reasoning, with language extensions and kernel additions.
- **Agent Development Environment**: The Development Environment is a GUI for viewing and manipulating JACK applications.
- **Agent Debugging Environment**: This consists of an Agent Interaction Display for viewing messaging between agents along with switches to the kernel for displaying internal execution states.
- **JACOB**: The JACK Object Modeller is a powerful object modelling toolkit to support object transport and interaction with existing applications in Java and C++. It will stream objects in a human readable textual format, a fast binary format and in XML.

8. The Future

Agent Oriented Software is pursuing an aggressive release schedule for JACK, and is constantly extending the software suite with new features and tools. In order to support these developments, Agent Oriented Software is involved in multiple research projects working towards the advancement of JACK and agent systems in general. These projects include collaboration with the University of Melbourne, RMIT University, and the Cooperative Research Centre for Smart Internet in Australia, Cambridge University in the United Kingdom, and the Italian research institute IRST. This research covers areas such as: Naturalistic decision making within agent reasoning; Agent-based manufacturing controllers; Intelligent agent Internet assistants; Agent development methodologies and the simplification of agent-based software development; Enhancements to team-based reasoning in simulation.

9. Benefits of JACK

The approach taken by JACK has a number of advantages in comparison with other agent frameworks coming from the artificial intelligence world and standard object-oriented architectures.

The adoption of Java guarantees a widely available, well-supported execution environment. In addition to the promises of the language (summarised by the well-known slogan 'compile once, run everywhere' by Sun Microsystems), an increasing number of software components, tools and trained engineers are becoming available.

To the A.I. researcher, the adoption of an imperative, relatively low-level language such as Java means losing some of the expressive power offered by frameworks based on logic or functional languages. However this is compensated, not only by the universal availability mentioned above, but also by the modular approach of JACK. As stated in the previous sections, most components of the framework can be tuned and tailored. This makes JACK particularly suited to experimentation with new agent architectures in order to try out new functionality (new mental attitudes, different semantics, additional types of knowledge bases, and so on) or to study performance characteristics in specific contexts.

Moreover, when compared with frameworks based on traditional A.I. languages, JACK has distinctive advantages due to a proper utilisation of the intrinsic characteristics of Java. The most important is strong typing, which reduces the chances of programming errors introduced by simple typos. It also provides a very basic version control by making sure that interfaces are compatible at run-time. Next is performance, which makes the execution speed of agent code written in JACK comparable to a direct implementation in C or C++.

For the engineer developing a sophisticated distributed application, JACK offers several useful aspects. For instance:

- It is an efficient way to express high level procedural logic within an object-oriented environment. This also helps in rapid application development by allowing a clear distinction between abstract data types and their operations on the one side and, application-specific behaviour requiring fine-tuning or evolution when the system is already operational on the other side. While the former should be based on high performance, well tested, highly reusable and ultimately expensive code, the latter is better expressed as plans which can be easily modified;
- The context sensitivity and sophisticated semantics of mental attitudes of the BDI architecture. These characteristics enable some levels of adaptability to changing conditions.
- The ease of integration with legacy systems. This enables, among other things, an incremental approach to distributed system development.

When compared with frameworks originating from research environments, JACK has the clear advantages of being lightweight, of industrial strength and accessible to a large community of engineers trained in object-oriented programming.

10. Conclusions

JACK Intelligent Agents™ is a multi-agent framework that extends the Java language. The current version supports the BDI model and SimpleTeam, an extension to support team-based reasoning. JACK's modularity enables extensions and further models to be easily supported.

JACK is an industry-strength product, providing a framework that takes a solution founded in artificial intelligence research into practical use. Compared with its 'predecessors', e.g. the PRS and dMARS systems mentioned above and other similar agent frameworks available in the academic world, JACK is not a 'pure' A.I. system. Instead, it constitutes a successful marriage between the vision of agent research and the needs of software engineering, bringing the power of agent technology to and enriching the host language, Java.

We are confident that JACK provides benefits both to the software engineer developing distributed systems and to the academic researcher.

11. References

[1] M. Wooldridge and N. R. Jennings, "Intelligent Agents: Theory and Practice", *The Knowledge Engineering Review*, vol. 10, no 12, pp 115-152, 1995.

[2] Michael E. Bratman, "Intention, Plans, and Practical Reasoning", Harvard University Press, Cambridge, MA (USA), 1987.

[3] A. S. Rao and M. P. Georgeff, "An Abstract Architecture for Rational Agents", *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning* (KR'92), C. Rich, W. Swartout and B. Nebel (editors), Morgan Kaufmann Publishers, 1992.

[4] M. P. Georgeff and F. F. Ingrand, "Decision - Making in an embedded reasoning system", *Proceedings of the International Joint Conference on Artificial Intelligence*, Detroit, MI (USA), 1989.

[5] M. d'Inverno, D. Kinny, M. Luck, M. Wooldrige, "A Formal Specification of dMARS", *INTELLIGENT AGENTS IV: Agent Theories, Architectures, and Languages*, M. Singh, M. Wooldrige, and A. Rao (editors), LNAI 1365, Springer-Verlag, 1998.

[6] Cohen, P.R., and Levesque, H. "Teamwork. Nous", *Special Issue on Cognitive Science and Artificial Intelligence*, 25(4):487--512, 1991.

[7] Vozzo, A., and Haub, J., "Military Asset Tasking Simulation Using Intelligent Agents", in *Proceedings of the Fourth International SimTecT Conference*, Melbourne, Australia, 1999.