Using Java for Artificial Intelligence and Intelligent Agent Systems[1]

October 1999

Technical Report 99-04

By:

Paolo Busetta, Ralph Rönnquist, Andrew Hodgson & Andrew Lucas
Agent Oriented Software Pty. Ltd., Melbourne, Australia
{paolo,ralph,ash,cal}@agent-software.com
http://www.agent-software.com

[1] This article has been edited for clarity and updated to reflect the current JACK Intelligent Agents distribution

Abstract

Intelligent Agents are being used to model simple rational behaviours in a wide range of distributed applications. In particular, agents based on the Belief-Desire-Intention (BDI) architecture have been used successfully in situations where some modelling of human reasoning and team cooperation has been needed, such as simulation of tactical decision making in air operations and command and control structures. Other applications include business process reengineering, telephone call centres, and air traffic management.

However, Intelligent Agent frameworks have so far been large, monolithic software systems. With their origins in research on Distributed Artificial Intelligence, these frameworks have generally been developed as research environments in the research laboratory. Consequently they have been unduly large, complex to use and based on non-mainstream AI languages.

The JACK Intelligent Agents™ framework presented in this paper brings the concept of intelligent agents into the mainstream of software engineering and Java. JACK is a third generation agent framework, designed to be a set of lightweight components with high performance and strong typing.

We discuss the advantages and issues of using Java to implement such an Intelligent Agent framework. We present JACK's extensions to the Java language for defining the extra concepts needed in for Intelligent Agents. We discuss the benefits of our component based approach, both for experts in artificial intelligence (such as the availability of an ever increasing amount of commercial, industrial-strength software) and the software engineer developing sophisticated distributed applications (such as n-tier business systems).

1 Introduction and Overview

Artificial Intelligence is at the forefront of innovation in computing. Recent examples of common technologies derived from, or heavily influenced by, AI research include object oriented programming (Smalltalk being a major case in point), graphical user interfaces, and neural networks.

A relatively recent area of research centred on intelligent agents and multi-agent systems is exploring the modelling of simple rational behaviours in distributed applications. This research is expanding the boundaries and the technologies of what is currently considered distributed programming by mainstream engineering practice, and shows the potential for practical application in the near future.

Agent Oriented Software Pty. Ltd. (AOS), based in Melbourne, Australia has built JACK Intelligent Agents™ ("JACK"), a framework in Java for multi-agent system development. The company's aim is to provide a platform for both industrial and research applications; consequently, JACK has been built having in mind efficiency, extensibility and ease of access to the Java community.

In Section 2, we contrast agent-oriented programming with traditional distributed programming. In Section 3, we present the approach taken to develop JACK. Section 4 summarises the major technical characteristics of JACK, while Section 5 discusses how to build an application; this is also illustrated with a simple programming example. In Section 6, we present the Belief-Desire-Intention (BDI) architecture, which is the agent model natively supported by JACK. Finally, in Section 7 we summarise the benefits of using JACK while developing distributed applications.

An evaluation copy of JACK can be downloaded from http://www.agent-software.com.au.

2 Agent Oriented Programming vs. Object Oriented Programming

Intelligent Agents are currently the subject of research by a wide and varied community worldwide. Intelligent agents have received various, if not contradictory, definitions; this is not surprising, given the wide variety of goals set by different researchers. Good starting points for a review of the literature, even though not particularly recent, are papers by Wooldridge & Jennings (1995) and Franklin & Graesser (1996). In general, researchers agree that an agent is a complex object that shows some degree of autonomy and social ability, and combines pro-active and reactive behaviours. We discuss, in Section 6, a specific and successful model, the Belief-Desire-Intention architecture, and how it incorporates the features mentioned above. To help put agents and JACK into a correct engineering perspective, we have included some general considerations regarding what has been called 'agent oriented programming' (Shoham 1993).

Broadly speaking, an agent can be seen as a process that pursues a number of goals over a long period of time (relative to the application domain), and somehow reacts and adapts to the evolution of its environment. A multi-agent system attempts to pursue some kind of common goals by a combination of cooperation, negotiation and competition among agents.

From an engineering perspective, agent-based systems differ from traditional distributed systems because of their emphasis on distributed problem solving; programming is at a higher level of abstraction than is currently allowed by mainstream languages and methodologies.

Distributed object oriented applications are commonly developed by creating or customising classes at different levels of abstraction and stacking them, starting from some communications infrastructure at the lowest layer. Typically, a traditional system does not incorporate any representation of

global or per-process goals, which remain in the minds of its designers and are somehow lost in the process of top-down decomposition and distribution over the network.

In contrast, building an agent-based system commonly follows a process that is the reverse of what is described above. An agent is described in terms of its high-level objectives, which usually consist of handling certain messages and events and achieving given goals; multi-agent frameworks may allow the declaration of objectives for the whole system. It is only during a later refinement phase that scripts or rules are associated with high-level objectives; in some instances, agents can even plan while executing or 'learn', for instance, by querying other agents in the same network. At run-time, it is possible to trace the reasons (that is, the high-level objectives) that triggered the observed behaviour of an agent. Furthermore, agents can often choose between different courses of actions (that is, scripts, rules, plans, and so on) in order to pursue their objectives, and can try many of them sequentially or concurrently, depending on their state and that of the environment.

In many instances, agent development frameworks are based on, or allow access to, other AI technologies (for example, logic or functional programming, knowledge bases, fuzzy logic, and so on).

An agent-based system could be seen as little more than an application of patterns such as the Active Object. However, its development process and tools are different from conventional distributed programming. These tools enable the declaration of the objectives and behaviour of agents at a higher level of abstraction and support a corresponding view of the activity of the system at run-time. Among other advantages, this allows a richer set of distributed architectures than client-server (including cooperation in teams, market-style negotiation for the distribution of tasks among participants, and so on) and rapid application development. The implementation of distributed procedures (business rules are a typical example) tends to be direct and straightforward.

Thus a framework for intelligent agents is more than just a scripting language or a set of components for distributed applications. Such a framework must facilitate correspondence of the observed behaviour of an agent to some high-level objectives. The framework must also take care of tasks being pursued concurrently and prioritise them when required. Importantly, it must help in coordinating potentially conflicting tasks, in choosing the best course of action when alternatives are possible and reacting appropriately on failure. Managing these aspects is sometimes referred to as 'meta-programming', they are first-order elements of agent programming and represent another important distinction compared with traditional procedural or object-oriented programming.

Most frameworks currently available in the research environment have shortcomings. For example, some of them are based on languages or technologies considered (quite rightly) esoteric by mainstream engineering. Also, a very high level language or framework is usually not appropriate to solve problems for which proven, efficient algorithmic solutions are available. Moreover, agent-based applications require access to existing computing infrastructures and software in order to re-use components or information already in place and to add new functionality to legacy systems (by either 'wrapping' them into an agent infrastructure or adding high-level procedures, such as business rules, as an external component). These considerations are some of the motivations for JACK, described in the next section.

3 The approach of AOS

When Agent Oriented Software set out to design JACK Intelligent Agents it had a few major goals deriving from the previous experience of its partners and engineers with both distributed object oriented systems and intelligent

agents. These goals 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 facilitate further applied research.

JACK has been designed primarily for use as a component of larger environments, since most organisations possess and depend upon large legacy software systems or have built a base of valuable re-usable software over time. Thus, a JACK agent coexists and is visible as simply another object by non-agent software. Conversely, a JACK programmer is allowed to easily access any other component of a system. Type safeness when accessing data, reliability, and support for a proper engineering process are key requirements in this kind of environment.

For similar reasons, JACK agents are not bound to any of the agent communications languages already existing or being developed. JACK has been geared towards industrial object oriented middleware (such as CORBA) and message-passing infrastructures (such as PVM). In addition, JACK provides a native lightweight communications infrastructure for situations where high performance is paramount.

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. The choice of Java as development platform was made after considering: its availability on all modern and foreseeable future platforms; the high quality of its implementation; the rapidly growing body of off-the-shelf software components and interfaces to external systems; and its increasing acceptance by both the marketplace and computer practitioners.

4 Overview of JACK Intelligent Agents

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 JACK, include support for the BDI architecture described in Section 6.

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 broken down 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, or interaction with other agents).

Importantly, the programmer can also include Java statements within the components of an agent.

For the convenience of programmers, in particular those with a background in AI, JACK also supports logical variables and cursors. These are particularly helpful when querying the state of an agent's beliefs. Their semantics are 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 (Section 6).

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

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

Importantly, the JACK kernel supports multiple agents within a single process. This is particularly convenient for saving system resources. For instance, agents which 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 architecture with new plug-ins. Moreover, a number of graphic development and monitoring tools are being developed in order to improve usability both to software practitioners and to domain experts (for instance, business analysts writing business rules but not interested in programming in Java).

5 Application development with JACK

In an ideal setting a developer building an application with JACK should start by identifying the distributed functional components of the system. The design of a multi-agent application requires a sound understanding of distributed system development and distributed AI principles that we do not discuss here. However, observe that in practical situations the decision as to how to distribute functionality may be dictated by a number of external constraints, such as the existence of legacy systems or a specific communications infrastructure.

Let us assume, for the sake of this discussion, that the functionality to be provided by an agent has been identified and that the BDI architecture (Section 6) has been chosen as appropriate to the task. At this stage, two main activities have to be performed (not necessarily in the order given below):

- Identification of the elementary classes (that is, abstract data types and the operation allowed on them) required to manipulate the resources used by the agent. These could be external (relational databases, the Internet, the arms of a robot, a GUI and so on) as well as internal (for instance, specific mathematic data structures to represent financial or spatial information).
- Identification of those elements that constitute the so-called mental states of the agent, that is, its high level objectives and the logic driving its behaviour. In the case of BDI, this boils down to finding:
- which external events drive the agent (including messages from other agents);
- which goals the agent can set for itself;
- which beliefs influence the adoption of plans; and,

- procedures (that is, the plans in BDI terms) required to accomplish tasks, achieve goals and react to events in the various possible contexts.

The implementation of an agent is then: a mix of normal Java code for the elementary classes; and, extended Java, as described in Section 3.2, for the agent-specific components. The plans of a JACK agent are, in general, sequences of operations on elementary objects, manipulations of the mental states (e.g., submitting sub-goals or changing beliefs) and interactions with other agents.

A JACK plan can be considered as procedural logic, and represented as a *flow diagram*, *state diagram*, *coordination diagram* or other similar notations in an object oriented methodology such as UML. Consequently, JACK could be used as an extended object oriented framework supporting event-driven and procedural logic in a concurrent execution environment. To this, JACK adds the benefits of sensitivity to the context, explicit goal representation and sophisticated management of failure provided by the BDI architecture.

To give a sample of the code of a JACK agent, we have extracted an example from one of the tutorials that are part of the JACK documentation set. The purpose is not to show agent programming but to illustrate that JACK code is a straightforward Java extension. Further examples can be found in the user documentation provided with JACK, which can be downloaded as part of the evaluation package.

This example has agents that "ping" each other; that is, exchange empty messages. The message being exchanged is represented as an event that originates with one agent is received by another:

```
event PingEvent extends MessageEvent {
int value;

  #posted as ping(int value)
  {
        this.value = value;
  }
  }
```

Note the agent-related "`event`" keyword, in place of the object-oriented term "`class`" of Java. The event is also declared "`MessageEvent`", which means that it can be notified to another agent; this causes JACK to bring in all the required communications support. The "`#posted as`" statement declares how the event is generated (in this case, by invoking a Java method "`ping()`" with one integer parameter).

The following plan handles the notification of the event above and replies to its sender by "bouncing" the event back. This simple example is not sensitive to the event context, ie, there is no restriction on the state of the beliefs of the agent for its applicability; context checking can be introduced as an additional method of the plan.

```
plan BouncingPlan extends Plan {

#handles event PingEvent ev;
#sends event PingEvent pev;

body()
{
@send ( ev.from, pev.ping(ev.value + 1) );
/// Reply to the sender of the event
```

```
}
}
```

Fig. 2 - The BouncingPlan

A trivial “ping agent” is defined below. It has a single plan and handles a single event. When its method “`ping (String other)`” is called, it notifies the `PingEvent` with value 1 to the agent called “`other`”. If the latter is another `PingAgent`, then `BouncingPlan` above is invoked, a `PingEvent` with value 2 is sent back, and so on to infinity.

```
agent PingAgent extends Agent {

#handles event PingEvent;
#uses plan BouncingPlan;

#posts event PingEvent pev;

void ping (String other)
{
send(other, pev.ping(1));
}
}
```

Fig. 3 - The PingAgent definition

The application can instantiate as many `PingAgent` agents as it desires. The name of the agents and their network addresses are determined by the communications mechanism in use; as said before, JACK provides a high performance messaging system with a simple naming scheme.

6 Belief-Desire-Intention Agents

The BDI agent model supported by JACK v1.2 has its roots in both philosophy and cognitive science, and in particular in the work of Bratman (1987) on rational agents. A rational agent has bounded resources, limited understanding and incomplete knowledge of what happens in the environment 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 revise 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.

BDI agents depart from purely deductive systems and other traditional AI models because of the concept of intentionality, which significantly reduces the extent of deliberation required. The BDI model has demonstrated its suitability 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, application of business rules in telephone call centres, and diagnostics in telecommunications networks.

Based on previous research and practical application, Rao and Georgeff (1992) have described a computational model for a generic software system implementing a BDI agent. Such a system is an example of an event-driven program. 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 pre-compiled 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 in a plan must 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 or any other atomic action on the agent's own state or the external world.

This abstract BDI architecture has been implemented in a number of systems. Of these, two are of particular relevance to JACK as they represent its immediate predecessors. The first generation is typified by the Procedural Reasoning System (PRS) (Georgeff & Ingrand 1989), developed by SRI International in the mid '80s in LISP. dMARS (d'Inverno et al. 1998), built in C++ in the mid '90s by the Australian Artificial Intelligence Institute in Melbourne, Australia, is a second generation system. dMARS has been used as development platform for a number of technology demonstrator applications, including simulations of tactical decision-making in air operations and air traffic management.

Research on BDI is very active worldwide. In Australia, it is currently addressing cooperative, distributed computation by teams of agents [8], reliable computation by means of transactions [9], and other areas.

7 Benefits of JACK

Agent Oriented Software's approach with JACK has a number of advantages in comparison with other agent frameworks coming from the artificial intelligence world and with standard object oriented architectures.

The adoption of Java guarantees a widely available, well-supported execution environment. In addition to the promise of the language (summarised by the well known slogan "compile once, run everywhere" by Sun Microsystems), we expect that an increasing number of software components, tools and trained engineers will be available in the next few years.

To the AI 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 said in the previous sections, most components of the framework can be both tuned and tailored to particular requirements. This makes JACK particularly suited to experimentation with new agent architectures for evaluating novel 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 AI languages, JACK has several 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 mis-typing. 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++.

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

- 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, on the other side, application-specific behaviour requiring fine-tuning or evolution when the system is already operational. 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.
- 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.

8 Conclusions

JACK Intelligent Agents is a multi-agent framework that extends the Java language. The current version supports the BDI model, and its modularity enables extensions and different 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 first and second generation 'predecessors', (such as the PRS and dMARS systems mentioned above and other similar agent frameworks available in the academic world) JACK is not a 'pure' AI system. Instead, it constitutes a successful marriage between the vision of agent research and the needs of software engineering, bringing with it the power of agent technology and enriching the host language, Java.

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

References

Bratman M.E., *Intention, Plans, and Practical Reasoning*, Harvard University Press, Cambridge, MA (USA), 1987.

Busetta P. & Kotagiri, R., (1998), "An Architecture for Mobile BDI Agents", *Proceeding of the 1998 ACM Symposium on Applied Computing (SAC'98)*, J. Carroll, G. B. Lamont, D. Oppenheim, K. M. George and B. Bryant (editors), Atlanta, Georgia (US).

d'Inverno, M., Kinny, D., Luck, M., & Wooldrige, M. (1998), "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.

Franklin, S. & Graesser, A. (1996) 'Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents', *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag.

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

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

Shoham, Y., (1993) 'Agent-Oriented Programming', in *Artificial Intelligence*, vol. 60, no 1, pp 51-92.

Tidhar, G., Sonenberg, E.A. & Rao, A. (1998) 'On Team Knowledge and Common Knowledge', *Proceedings of the Third International Conference on Multi-Agent Systems*, ed. E. Demazeau, IEEE Press.

Wooldridge M. & Jennings, N.R. (1995) 'Intelligent Agents: Theory and Practice' *The Knowledge Engineering Review*, vol. 10, no 12, pp 115-152, 1995.