Light-Weight Intelligent Software Agents in Simulation

Paolo Busetta, Ralph Rönnquist, Andrew Hodgson, Andrew Lucas
Agent Oriented Software Pty. Ltd.
221 Bouverie Street
CARLTON 3053

Keywords:
Java, Intelligent Agents, BDI reasoning

ABSTRACT: Today Intelligent Agents are being used for modelling human behaviours in simulation. In particular, powerful Belief-Desire-Intention agents have been used successfully in air operations simulations where modelling of human reasoning has been needed to simulate tactical decision making and command and control structures.

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 complex to use, as well as expensive and difficult to integrate as components of larger systems.

The JACK Intelligent Agents (TM) framework presented in this paper brings the concept of intelligent agents into the mainstream of software engineering. JACK is a third generation agent framework, designed to be light-weight, with high performance, and strong typing. It merges the Intelligent Agent concept with the object-oriented Java language, and is designed for building agents for simulation.

We discuss the advantages of having Agents as light-weight components for simulation, and of defining these agents in a mainstream object-oriented language.

1 Introduction

Intelligent Agents are being used for modelling simple rational behaviours in a wide range of distributed applications, including simulation. There are however various, if not contradictory, definitions of intelligent agent. By general consensus, it is a type of software that shows some degree of autonomy, social ability, and combines pro-active and reactive behaviours [1]. One of the better known and most successful paradigms for intelligent agents is the so-called BDI (Belief-Desire-Intention) architecture, which has seen a number of academic and industrial applications, including military simulation of both independent and cooperative entities.

Agent Oriented Software Pty. Ltd. (AOS), based in Melbourne, Australia, has built JACK Intelligent Agents (TM)[i], which is a framework in Java for multi-agent system development. The company's aim is to provide a platform for commercial, industrial and research applications; one of the most important is simulation, in particular for defence and other domains characterised by a large number of entities showing sophisticated behaviour. JACK supplies a high performance, light-weight implementation of the BDI architecture as an extension of its host language, Java, and can easily support additional agent models and specific application requirements.

In the following, we discuss intelligent agents in the context of simulation. We introduce JACK, and present the architecture it currently supports, BDI. We give an overview of application development with JACK, and conclude with an analysis of the benefits it provides.

[i] For the sake of brevity, we refer to JACK Intelligent Agents simply as "JACK".

2 Intelligent Agents in Simulation

Some of the more advanced applications of computer simulation are in defence, and important uses of intelligent agents have been seen in this domain. The considerations reported below arise mostly from experience in military simulation. However, they apply equally to other simulation domains, such as emergency systems (firefighting, natural disaster protection, and so on) and any activity involving costly equipment, high levels of training and standard operating procedures (for instance, commercial aviation, air-traffic control).

Historically, simulation in defence has been used for the evaluation of acquisitions or force development options [2]. Modelling and simulation for this purpose is becoming increasingly complex as the number of entities grows and sophistication of scenarios increases. The complexity of this task is further increased by the difficulty in modelling human decision-making with sufficient fidelity using conventional software approaches. Early applications of intelligent agents 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, certain classes of agents (in particular BDI, discussed later) extend the modelling of reasoning to explicitly model the communications between simulated entities and the coordination of joint activities required for team behaviour. This is particularly helpful when conducting studies that take into consideration complex command and control systems. These are difficult to analyse manually, and advanced modelling and simulation tools are required. Intelligent agents can represent the reasoning and command capabilities appropriate to the humans' assigned roles in the hierarchy, allowing different command and control strategies to be rapidly evaluated under varying circumstances.

The high cost of live exercises has required the development of realistic synthetic training environments. However, so far 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 skills, they learn to predict the training system's response. Intelligent agents allow the simulated humans in training systems to behave in a more credible 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 involved to provide this.

2.1 A case for light-weight agents

The increasing complexity of equipment and command and control structures, together with the growing dependency on high quality communications (both voice/image and digital) in most human activities, has lead to a corresponding growth in complexity of the software for modelling and simulation. This is further complicated by the move from modelling small numbers of entities involved in short-term engagements to modelling complete theatres of war over several months. The consequent requirements in terms of computational power are growing exponentially, due to increases in communications, in the amount of information to be handled, and in the sophistication of the behaviours of the simulated entities and their interactions (e.g., teams).

Simulation systems that perform satisfactorily for relatively small scenarios do not necessarily scale well; in fact, intelligent agent behaviour is potentially a problematic area, since it has been based on technology more appropriate to A.I. laboratory research than large-scale deployment. Even when considering the expected gains in computing power and network speed (which historically have shown to be linear over time), we believe that light-weight implementations of agents are needed in order to match the exponential growth in requirements.

An issue relevant to this discussion is the rapid obsolescence of computing platforms. As a consequence, the choice of a technology for implementing agents should be made so that it is possible to benefit from continuing rapid enhancements to hardware and software systems.

3 JACK Intelligent Agents

In this section, we present JACK by first highlighting the goals set by its designers, then by providing an overview of the engineering characteristics of version 1.2 of the framework and finally by describing some features of interest to simulation systems.

3.1 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 facilitate 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. This is particularly true in the area of training, where large investments into sophisticated HCI systems may have been done over time. Thus, JACK has been designed primarily for use as a component 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 a proper engineering process are then key requirements in this kind of environment.

For similar reasons, JACK agents are not bound to any specific agent communications language. Instead, they are geared for industrial object-oriented middleware (such as CORBA) and message passing infrastructures (such as DIS). In addition, JACK provides a native light-weight inter-agent 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 (such as databases); and its increasing acceptance by both the marketplace and computer practitioners.

3.2 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 1.2, released at the end of October 1998, include support for the BDI model.

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 A.I., 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:

- the automatic management of concurrency among tasks being pursued simulatenously (*intentions* in the BDI terminology);
- the default behaviour of the agent in reaction to events, failure of actions and tasks, and so on; and
- 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 supply 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.

3.3 Support for Simulation

Some of the features of JACK have been designed to explicitly address, or have been influenced by, simulation requirements.

Agents are equipped with a *clock,* which can be used for time-driven simulation. The clocks can be configured to measure actual time as well as simulated time; the latter can be accelerated or retarded while running by means of a graphical interface. This allows us, for instance, to debug a model or to demonstrate or to rehearse scenarios by fast-forwarding up to a critical point and then time-stepping from then on. A clock can even be replaced by an application-specific timer, provided for instance by an external source.

A component called API Builder (APIB) is provided to simplify the task of interfacing JACK agents with legacy systems and external sources of data. The programmer defines the format of application-specific data structures with the APIB's data definition language. These declarations are then converted in Java and C++ classes that support platform-independent coding and decoding, sending as messages and storage and retrieval from files and databases.

Significant events and messages being exchanged among agents can be easily traced, analysed by means of the supplied tool for interaction diagrams or by a user-supplied monitor, and logged for off-line analysis.

Future versions will provide additional tools to help building and monitoring agents. They will include a graphical development environment and ways to trace and view individual and joint tasks (or *intentions* in BDI terminology) being pursued by a team of agents. These tools will allow a non-computer saviour to model and analyse the behaviour of simulated intelligent entities.

4 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 on rational agents [3]. 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 A.I. models because of the concept of intentionality, which significantly reduces the extent of deliberation required. The BDI model has demonstrated 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, application of business rules in workflows, and diagnostics in telecommunications networks.

Based on previous research and practical application, Rao and Georgeff [4] 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) [5], developed by SRI International in the mid '80s. dMARS [6], built 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.

The BDI architecture provides a powerful means for implementing complex, distributed systems using multiple BDI agents. The SWARMM air mission model [7] was the first application to introduce BDI agents in teams. Current Australian research in team is directed towards BDI agents for command and control [8].

5 Application Development With JACK

In an ideal setting, a developer building an application with JACK would 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 A.I. principles that we do not discuss here. However, it should be noted 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 (for instance, HCI equipment) or a specific communications infrastructure (such as DIS).

For the sake of this discussion, let us assume that the functionality to be provided by an agent has been identified and that the BDI model 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):

- identifying the elementary classes (that is, abstract data types and the operation allowed on them) that are required to manipulate the resources used by the agent. These could be external (relational databases, an HCI device, a GUI and so on) as well as internal (for instance, specific mathematic data structures to represent spatial information);
- identifying those elements that constitute the mental states of the agent. 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 finally
  - ◊ the procedures (that is, the plans in BDI terms) required to accomplish tasks, achieve goals and react to events in the various possible contexts.

  The same process is performed to identify the joint mental attitudes of a team.

In the current version of JACK, the implementation of an agent is 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 (such as submitting sub-goals or changing beliefs) and interactions with other agents.

In order to improve the accessibility of JACK to the analyst not trained in Java or not interested in low-level programming details, future versions will include a graphical development environment. The users will be able to mix text and GUI based development as appropriate to their competence or the tasks at hand.

5.1 An Example

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 version 1.2 user manual. The purpose is not to show agent programming but to illustrate how JACK code looks as a straightforward Java extension. This section can be passed over by those not familiar with Java.

This example has agents that "ping" each other, that is, exchange empty messages. The message being exchanged is represented as an *event* that is originated by an agent and notified to another:

```
event PingEvent extends MessageEvent V
      int valueR

      #posted as pingPint value)
      V
        this.value = valueR
      W
    W
```

Figure 1: The "Ping" event.

Note the agent-related "`event`" keyword, in place of the object-oriented "`class`" of Java. The event is also declared "`MessageEvent`", which means that it can be notified to another agent; this also 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, i.e., 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 V

      #handles event PingEvent evR
      #sends event PingEvent pevR

      bodyP)
      V
        TsendPev.from, pev.pingPev.value + 1))R
          /// Reply to the sender of the event
      W
    W
```

Figure 2: The "BouncingPlan" plan.

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 V

      #handles event PingEventR
      #uses plan BouncingPlanR

      #posts event PingEvent pevR

      void ping PString other)
      V
        sendPother, pev.pingP1))R
      W
    W
```

Figure 3: The "PingAgent" agent.

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.

5.2 Current Applications of JACK

To date, simulation applications using JACK have been built or are being developed, mainly by DSTO in Australia. In [9], Cross and Rönnquist describe a simple air combat simulator, built with the aim of allowing comparisons with SWARMM [7]. Other projects currently being undertaken include a close action environment simulator for land-based simulation, and development of team-based agent systems for modelling military command and control architectures. Consideration is also being given to simulating maritime operations, as well as tactical air defence.

A demonstrator of a multi-agent resource scheduling system for land surveillance operations, known as Collection Plan Management System, has recently been delivered to DSTO's Land Operations Division; this demonstrator was built as a functional component within a CORBA environment.

6 Benefits of JACK

The approach taken by Agent Oriented Software 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 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 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 A.I. languages, JACK has several distinctive advantages due both to design choices and 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, and particularly relevant to the current discussion, is performance, which makes the execution speed of agent code written in JACK comparable to a direct implementation in C or C++. Moreover, the ability to package multiple agents within a single computer process can lead to significant savings in terms of system resources and communication.

Last but not least, JACK is of industrial strength and accessible to a large community of engineers trained in object-oriented programming. Future versions will provide graphical development tools, making JACK accessible also to analysts not trained in Java.

As previously said, agents are being successfully applied to simulation. In summary, the approach of JACK offers the following advantages when compared with traditional software technologies or other agent implementations:

- the context sensitivity and sophisticated semantics of mental attitudes of the BDI architecture;
- ease of integration with legacy systems; and
- scalability, that is, the ability to support a large number of functional entities.

7 Conclusions

We have discussed how to handle the computational problems coming from the increase in size and complexity of simulation scenarios by the adoption of a concept actively researched in the A.I. world, intelligent agents. At the same time, we emphasised the need for scalability and for light-weight implementations of sophisticated software agents.

JACK Intelligent Agents is a multi-agent framework that extends the Java language. The current version supports the BDI agent model, and its modularity enables extensions and different models to be easily supported. We have given an outline of the software development process with JACK and presented some examples. Future versions will include a graphical development environment that will allow the use of JACK without needing to access Java.

JACK's characteristics make it more suitable to large-scale simulations and integration with existing infrastructure than frameworks based on traditional A.I. technologies, while still offering the advantages of agent programming.

8 References

1. Wooldridge, M., and N. R. Jennings. Intelligent Agents: Theory and Practice. In: The Knowledge Engineering Review, vol. 10, no 12, pp 115-152, 1995.
2. Lucas, A., and S. Goss. The potential for intelligent software agents in defence simulation. To be presented at IDC'99 Symposium, 8-10 February 1999, Adelaide, Australia.
3. Bratman, M. E.. Intention, Plans, and Practical Reasoning. Harvard University Press, Cambridge, MA (USA), 1987.
4. Rao, A. S., and M. P. Georgeff. An Abstract Architecture for Rational Agents. In: 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.
5. Georgeff, M. P., and F. F. Ingrand. Decision - Making in an embedded reasoning system. In: Proceedings of the International Joint Conference on Artificial Intelligence, Detroit, MI (USA), 1989.
6. D'Inverno, M., D. Kinny, M. Luck, M. Wooldrige. A Formal Specification of dMARS. In: INTELLIGENT AGENTS IV: Agent Theories, Architectures, and Languages. M. Singh, M. Wooldrige, and A. Rao (editors), LNAI 1365, Springer-Verlag, 1998.
7. Tidhar, G., M. Selvestrel, and C. Heinze. Modelling Teams and Team Tactics in Whole Air Mission Modelling. In: G. F. Forsyth and M. Ali, editors, Proceedings of the Eighth International Conference on Industrial Engineering Applications of Artificial Intelligence Expert Systems, pages 373-381, Melbourne, Australia, June 1995. Gordon and Breach Publishers.
8. Tidhar, G., A. Rao, and L. Sonenberg. On Teamwork and Common Knowledge. In: Proceedings of the 1998 International Conference on Multi-Agent Systems, Paris, July 1998.
9. M. Cross, R. Rönnquist. A Java Agent environment for simulation and modelling. To be presented at SimTecT 99, Melbourne, Australia.

9 Author Biographies

Paolo Busetta is concluding his Master in Computer Science at the University of Melbourne; he completed his graduate studies in C.S. at the "Universita' delle Scienze" of Torino, Italy, in 1986. He has a significant experience in operating systems and distributed applications. In 1995, he joined the Australian Artificial Intelligence Institute, Melbourne, and was involved in extensions to dMARS and in a number of prototype applications and demonstrators. He joined Agent Oriented Software in 1998.

Ralph Rönnquist completed his Ph.D. in Computer and Information Science from Linköping University, Sweden, in 1992. His research interests included intelligent information systems and formal methods for representation of knowledge, and he participated in the Scandinavian research exchange initiative, SYDPOL. In 1993, he joined the Australian Artificial Intelligence Institute, Melbourne, took part in the development of dMARS and led a number of projects building prototype applications and demonstrators using dMARS. In 1998 he moved to Agent Oriented Software Pty. Ltd. to join the development of the JACK Intelligent Agents framework, and to pursue research on application of agent technology in close cooperation with the Intelligent Agent Laboratory at Melbourne University.

Andrew Hodgson, the Technical Director of Agent Oriented Software, is responsible for the development of Agent Oriented Software's intelligent agent system JACK. He has a BSc (Hons) from the University of Melbourne. A very experienced software developer, Andrew has extensive knowledge of advanced real-time systems, A.I. , and simulation. While at the Australian Artificial Intelligent Institute, he led the development of their core agent reasoning system (dMARS).

Andrew Lucas, the Managing Director of Agent Oriented Software, and leads the marketing and applications consulting activities. With a Ph.D in Engineering from Cambridge, Andrew has 20 years experience in the engineering, technology consulting and software industries in Australia and Europe. Andrew's extensive experience in software applications includes the aviation, defence, telecommunications and manufacturing industries.