

Chapter 1

JACKTM INTELLIGENT AGENTS: AN INDUSTRIAL STRENGTH PLATFORM

Michael Winikoff¹

¹*RMIT University*
GPO Box 2476V
Melbourne, 3001, AUSTRALIA
winikoff@cs.rmit.edu.au

Abstract Software agents offer a range of benefits to the development of complex software systems. However, before these benefits can be realised by the computing industry there is a need for an agent platform that can be accepted by industry. In this paper we describe the JACK agent platform: a mature and robust commercial product. We argue that JACK meets requirements such as familiarity, scalability and integrability which make it suitable for adoption by industry. We also describe interesting features of JACK such as the use of capabilities for structuring agents, and JACK’s approach to teamwork which allows hierarchical team structures.

Keywords: Agent Oriented Programming Language, Belief-Desire-Intention, Agent Platform.

1.1 Motivation

Software agents offer a range of potential benefits to the development and deployment of complex software systems, such as increased flexibility and adaptability, and more natural models of complex “nearly decomposable” systems [28, 29, 34, 35]. These benefits stem from the combination of features that are generally considered to be associated with intelligent software agents: being *autonomous*, *proactive*, *reactive* and *social*. Some argue that because agents are autonomous they reduce coupling [28, 42]. Some focus on the use of plans and goals in Belief-Desire-Intention (BDI) agents (and similar platforms), arguing that the

resulting number of ways in which a goal can be achieved gives agents flexibility in dealing with situations, and robustness in recovering from various types of failure (e.g., [43, Section 2.5]). Others argue that aspects of agents (e.g., autonomy, flexibility) are already being adopted by mainstream software engineering, and that this is evidence that these aspects are useful to modern software systems [61].

However, in order for these benefits to be realisable by the computing industry, a number of key technological pieces are required. One of the key pieces is a *methodology* (including concepts, notations, a process and techniques) that guides practitioners in designing agent systems. We do not focus on this here, but note that a number of methodologies have been developed including Gaia [60], Tropos [6], MaSE [14], Prometheus [43], ROADMAP [30] and others [2, 21]. Some of these methodologies have also been evaluated and compared in various ways [9, 12, 48, 49].

A second key piece of technology is an *agent platform* which can be used to create agent systems. Like the term “methodology”, the meaning of the term “agent platform” is somewhat debatable. We believe that an agent platform needs to contain at least the following components:

- An agent-oriented programming language that allows agents to be written directly using agent concepts (e.g., plans, goals, beliefs), rather than encoded in non-agent-oriented languages.
- A library or framework providing facilities for inter-agent communication including facilities for transmitting and receiving messages, and for locating agents (e.g., a name server).

JACKTM Intelligent Agents (referred to as “JACK” in the remainder of this chapter) [8] is an agent platform that includes these components and more. JACK includes an agent-oriented programming language; a platform for executing agents with infrastructure such as message marshalling and a name server; and development tools including a design tool, a graphical plan editor and a number of debugging views. Additionally, JACK includes a number of additional functionalities such as the ability to construct hierarchical *teams* of agents [27].

Looking at the history of object-oriented technologies, it is interesting to note that object-oriented programming languages such as Simula (developed in the 1960s) and Smalltalk (developed in the 1970s) significantly pre-dated work on object-oriented analysis and design (in the 1980s and 1990s). By analogy with this history one could argue that the availability of a widely accepted “standard” agent-oriented programming language is more crucial to the success of agents as a technology than the development of a widely accepted methodology. To be widely accepted

an agent platform must be accepted by industry, and so it is natural to ask what an industry-acceptable agent platform might look like.

We believe that to be acceptable to industry an agent platform must be:

- Familiar: presented as an extension of objects, rather than as a revolutionary new paradigm [42]. In particular, this means that the programming language should be easily learned by programmers who are used to currently popular languages (e.g., Java). In particular, this rules out languages that are based on alternative, less mainstream, paradigms such as logic programming.
- Integratable: the platform must allow agents to communicate and integrate not just with existing software including objects written in Java or C++, but also with databases, web servers, graphical user interfaces, etc. In particular, in order to flexibly integrate with a wide range of existing systems, the agent platform must be agnostic (or at least flexible) with respect to communication infrastructure. Supporting a single approach only, such as FIPA¹, is not desirable as there are many for communication and integration approaches in current use (e.g., Web Services, CORBA, Java RMI, HLA).
- Scalable: the language must support good software engineering practice, including the provision of suitable facilities for structuring large systems.
- Industrial Strength (Robust, Stable, Efficient): the implementation must be robust and reliable, and it must be able to support large numbers of agents efficiently.
- Documented and Supported: when using a technology that is not (yet) widely-known, it is vital to have good documentation and support.

Additionally, it is important for the agent platform to provide development tools (such as an integrated development environment and design tools), and debugging tools.

It is clear that, in today's computing environment, developing an agent platform built on top of the Java platform is highly attractive: it provides for portability across computing platforms and access to a rich collection of libraries. One approach to using the Java platform as

¹Foundation for Intelligent Physical Agents, <http://www.fipa.org>.

a basis is to use the Java language as the programming language and provide a library of agent features. This approach has the benefit of familiarity – the programming language used is Java itself – but has the drawback that the language’s semantics is fixed and cannot be changed. This approach is taken by the Jadex² system (see Chapter ??), and one consequence is that when a plan has a sub-goal, the programmer must write code to check whether the sub-goal has succeeded, and if not fail the plan. This check cannot be done automatically in Jadex (as it should be) without changing Java’s execution semantics. An alternative approach to using the Java platform as a basis is to create a new language and implement it using Java, either by compiling it to Java or writing an interpreter for the new language in Java. The new language can be quite different to Java (e.g., JAM [24] or Jason (see Chapter ??)) or, as in JACK, a conservative extension of Java.

Although there are clear benefits to using Java as a basis and there are good reasons to conservatively extend Java’s syntax, this does have the drawback of yielding a programming language that is relatively verbose. To some degree, JACK addresses this issue by using development tools that generate code skeletons by interaction with a GUI.

In the following section we briefly describe the JACK agent language and its features. In section 1.3 we discuss the JACK agent platform, and in section 1.4 we present applications developed with JACK. Although JACK is a commercial platform which caters for industrial usage, one of JACK’s major design goals was “to enable further applied research” [23], and so we briefly discuss research that has extended or built on JACK in section 1.5.

1.2 Language

The JACK programming language extends Java in a number of ways, both syntactic and semantic. The JACK language is a superset of the Java programming language, so all of Java’s libraries and facilities are easily accessible.

In the following sections we briefly describe the JACK programming language and its execution.

1.2.1 Specifications and Syntactical Aspects

Syntactically, JACK extends Java in three ways:

²<http://vsys1.informatik.uni-hamburg.de/projects/jadex/>

- 1 JACK adds new top-level declaration types which are used to declare agents, beliefsets, views, events, plans and capabilities.
- 2 Each of the top-level types is defined using various `#` declarations which define the properties of the entity and relationships between entities.
- 3 Within plan bodies JACK defines a range of `@` statements such as posting an event (e.g., `@post`) or waiting for a condition (`@wait_for`). Some of the `@` statements defined by JACK are listed in figure 1.1.

Figure 1.2 shows how these three syntactic extensions are used to define a (very simple) plan called `ProcessRequest` which is triggered by a message (declared with the `#handles` declaration), and replies to it with a response.

- `@post`, `@subtask` – simple event posting within an agent. `@post` is asynchronous, whereas `@subtask` waits for the event processing to finish before continuing.
- `@send`, `@reply` – inter-agent communication.
- `@achieve`, `@insist` – post a (goal) event under certain conditions. `@achieve(condition,goal_event)` checks whether the condition holds, and posts the event if it doesn't. `@insist(condition,goal_event)` is similar, but also checks whether the condition holds after the processing triggered by the event has finished. If not, the event is posted again.
- `@maintain` – checks for condition while handling event. `@maintain(condition,event)` will subtask the event, but will monitor the condition while the event processing runs. If the condition becomes false the plans that handled the event are aborted.
- `@sleep`, `@wait_for` – do nothing for a certain amount of time (`@sleep`) or until a certain condition is true (`@wait_for`).

Figure 1.1. Some statements provided by JACK

The top-level entities that JACK defines are:

Agent: An agent is an obvious basic entity for an agent-oriented programming language! In JACK, agents are specified by defining the events they handle and send, the data (including beliefsets) they have, and the plans and capabilities they use.

```

public plan ProcessRequest extends Plan {
    #handles event Request req;
    #sends event Response resp;

    context() {
        req.isValid;
    }

    #reasoning method body() {
        // Can contain Java code as well as JACK @-statements
        ...
        @reply(req,resp.response(...));
    }
}

```

Figure 1.2. A (very simple) Plan

Beliefset: A beliefset is effectively a (small) relational database that is stored in memory, rather than on disk. JACK makes it easy to define these and to define queries on beliefsets. Beliefsets can also post events in certain situations (e.g., whenever the beliefset is modified).

View: Views are “virtual” beliefsets that are computed from other beliefsets.

Event: An event is an occurrence in time that represents some sort of change that requires a response. Events are used in JACK (and in other BDI architectures) to model messages being received, new goals being adopted, and information being received from the environment.

Plan: A plan is a “recipe” for dealing with a given event type. Plans include an indication of which event they handle, a *context condition* which describes in which situations the plan can be used, and a plan body. The plan body, which can include Java code as well as JACK code, is what is actually executed as the system runs.

Capability: A capability is a modularisation construct. We discuss capabilities in section 1.2.3.

The execution of JACK is fairly typical for a BDI architecture. Events (which include messages from other agents) trigger plans. Each event

- 1 Event posted.
- 2 Determine the set of relevant plans .
- 3 Determine the applicable plans.
- 4 Select an applicable plan and run it.
- 5 If plan fails, go to step 4 (select an applicable plan).

Figure 1.3. Event handling in BDI architectures

will normally have a number of plans that handle that event, these are the *relevant* plans. Of the relevant plans, some will be *applicable* to the agent's current situation. This is determined by evaluating the plan's context condition³. If there are no (more) applicable plans the event has failed, and failure handling is triggered. Otherwise, one of the applicable plans is selected and its body is executed. This is summarised in figure 1.3.

The execution of a plan's body is fairly straightforward: the statements in the plan body are executed in sequence. However, there is one key difference between executing Java code and executing JACK code: each statement can fail, and if it does the rest of the plan is not executed and failure handling is triggered instead.

When a plan fails the event that triggered it is considered to have not been handled, and alternative plans for handling it are considered. This process looks for another applicable plan to try. If there is another applicable plan, it is tried. If all applicable plans have failed the event cannot be handled. If the event was posted from a plan (Events can also be posted from Java code) that plan fails and its triggering event is re-posted in an attempt to find an alternative applicable plan for it.

This execution cycle of events triggering plans is common to a whole family of BDI architectures (e.g., dMARS [15], JAM [24], PRS [20, 26], UM-PRS [31]). However, there are some details of the cycle that are specific to JACK and distinguish it from other platforms. Firstly, by default⁴ JACK re-computes the applicable set when considering alternative plans due to failure. This means that when a plan fails and alternatives are considered, the applicability of these alternatives is evaluated in the

³If the context condition has multiple solutions this will lead to multiple plan instances being considered as applicable.

⁴This behaviour, and other aspects of event handling, can be customised on a per-event-type basis.

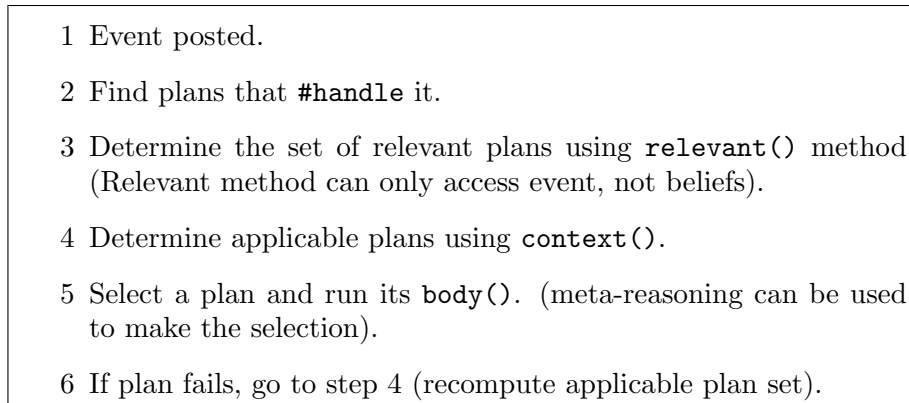


Figure 1.4. Event handling in JACK

current situation, not the situation when the event was first posted. Some other BDI architectures (such as JAM) do not re-compute the applicable plan set, and thus select plans based on out-of-date information when failure occurs.

Another detail that is specific to JACK is that the context condition is actually split into two parts: a context condition and a relevance condition. The relevance condition is a Boolean condition that is only evaluated once (eagerly) and can only access the details of the event, not any other data. The relevance condition is used to exclude plans based on the details of the event (which do not change). For example, if the event is a request for credit which specifies the amount and there are separate plans depending on the amount requested, the selection of plans can be done using a relevance condition rather than a context condition. The JACK execution cycle is summarised in figure 1.4.

When there are multiple applicable plans that can be used, the question arises of which one an agent should select (step 5 in figure 1.4). JACK provides a number of mechanisms that allow the programmer to specify how a plan should be selected. One mechanism is that plans will (by default) be selected in the order in which they are listed in the agent. Another mechanism runs another plan (a “meta-plan”) to decide which plan to select.

It is worth mentioning that JACK actually provides a variety of event types which behave differently. For example, message events do not trigger failure handling if their handling plan fails.

1.2.2 Semantics and Verification

Although JACK is quite well documented, its semantics have not been formally specified. Since JACK is a superset of Java and is compiled to Java, formally defining JACK's semantics would require a formal definition of Java's semantics, something that is still an active area of research⁵ [1].

However, although JACK itself has not been formally specified, the event-plan execution cycle which JACK shares with other BDI platforms has been formalised in various ways by various researchers. Anand Rao's work on AgentSpeak(L) [47] aims to bridge the "BDI gap" between theories and implementations by defining a language capturing the essence of BDI platforms whilst having precisely defined semantics. Although the formal semantics given by Rao is incomplete, the work has inspired a number of implementations of the language such as AgentTalk⁶, an implementation based on SIM_AGENT [36], an implementation in Java that is designed to run on hand-held devices [46], and the Java-based Jason⁷ (see Chapter ??).

Since Rao introduced AgentSpeak(L), a number of authors have published complete formal semantics for the language. The specification language Z ("Zed") was used to formally specify the essential execution cycle of AgentSpeak [16], and an operational semantics for AgentSpeak was given by Bordini and Moreira [4]. However, neither of these formalisations included the failure handling mechanism. A precise operational semantics including failure handling was given by Winikoff *et al.* [58] for a language (called "CAN") which is a superset of AgentSpeak.

Since JACK's semantics has not been formalised, JACK programs cannot be formally verified. However, verification of entire implemented systems is not currently realistic. Research into model checking of agent programs is still quite young [3], and is not yet applicable to large agent programs. Consequently, we believe that presently formal techniques are best applied to verifying *aspects* of systems, such as key algorithms or interaction patterns.

1.2.3 Software Engineering Issues

One of JACK's strengths is its support for modern software engineering practices. In addition to the features provided by Java (objects,

⁵For example, the formalisation described at http://www-sop.inria.fr/oasis/java/java_sem.html is for a subset of Java.

⁶<http://www.cs.rmit.edu.au/~winikoff/agenttalk>

⁷<http://jason.sourceforge.net/>

packages), JACK adds a number of features that can be used to structure an agent system.

One new feature is that a plan's body can be broken down into a number of separate *reasoning methods*, rather than being a single monolithic block of JACK code. This allows a single plan to be structured internally.

Another feature that was introduced by JACK (and subsequently adopted by Jadex) is *capabilities* [7]. A capability is the agent-oriented equivalent of a module, corresponding to a coherent ability that an agent has. Capabilities contain plans and beliefs, and specify which events they handle and post. In addition, capabilities can also contain sub-capabilities which allow hierarchical module structures to be specified as appropriate.

Another Software Engineering practice is *consistency checking*. JACK checks that the various declarations of which events are posted and handled by which entities (agents, capabilities and plans) are consistent. Additionally, the JACK agent programming language is, as an extension of Java, strongly and statically typed, and the type checking done at compile-time can catch a range of mistakes made by the programmer.

As JACK is a superset of Java, integrating with existing Java code is straightforward. An example of this is the work of [13] which integrated JACK with the JSHOP planner (which is written in Java). JACK can also be integrated with existing C++ code using JACOB (see section 1.3.2). JACK has also been successfully integrated with systems in Fortran, C, and Ada.

1.2.4 Other features of the language

In addition to the features discussed above, the JACK agent language includes a number of other significant features. Perhaps the most significant is its support for “team-oriented” programming.

JACK's support for teams is an optional extension which adds two new concepts (teams and roles) and extends Plans to TeamPlans [27]. A team is an entity which, like an agent, can contain plans, capabilities, data, etc. but, unlike agents, a team can also have *sub-teams*, enabling natural modelling of hierarchical organisational structures. It is important to realise that a team is an active entity that can have beliefs and execute (team) plans; it is not merely a collection of agents. Indeed, when the team extension is enabled, an individual agent is modelled simply as a team that has no sub-teams!

For each team type, roles are used to specify the interface (in terms of events received and sent) that must be fulfilled by its sub-teams.

```

teamplan FeedBaby extends TeamPlan {
  #handles event BabyHungry pfv;
  #uses role Parent parents as p1;
  #uses role Parent parents as p2;

  //establish the task team.
  #reasoning method establish() { ... }

  body() {
    @parallel(ParallelFSM.ALL,false,null) {
      @team_achieve(p1, p1.prepareFood.pf());
      @team_achieve(p2, p2.calmBaby.cb());
    };
    @team_achieve(p2,p2,feedBaby.fb());
  } // body
} // FeedBaby team plan

```

Figure 1.5. A simple TeamPlan

The team extension also extends Plans to TeamPlans by adding the ability to delegate tasks to sub-teams, and to perform steps in parallel. Team plans also differ from plans in that they have an `establish()` reasoning method which assembles the sub-teams that will be involved in the plan (the “task team”). Each TeamPlan that is run by a team can have a different assignment of sub-teams. For example, given a team of soccer-playing robots, one TeamPlan may require two attackers, whereas another TeamPlan may require both a defender and a goal keeper.

Figure 1.5 shows a simple example TeamPlan. This TeamPlan specifies that feeding a baby requires two sub-teams, both playing the role of a parent. One parent prepares the food at the same time as the other parent calms the baby. Once this is done, the baby is fed.

JACK’s teams support also includes other features, such as being able to automatically repair teams, and being able to automatically propagate beliefs from a team to its sub-teams and vice versa.

JACK’s approach to teamwork is different to standard approaches that regard teams as a collection of agents having certain patterns of mental attitudes (e.g., joint intentions), existing approaches to teamwork [11, 50] do not consider teams to be entities in their own right, and do not support hierarchical team structures. Comparisons of JACK’s approach

to teamwork with other approaches to teamwork can be found in [10, 25].

Another feature of JACK is an event type called `InferenceGoal`. Whereas other event types are handled by finding an applicable plan and executing it, with alternative plans being considered (for some event types) only if plan execution fails, an `InferenceGoal` event is handled by executing *all* applicable plans in sequence. This behaviour is useful for performing certain types of reasoning such as emulating rule firing in expert systems.

Finally, the JACK compiler is modular, and the JACK language can be extended using plugins, but this aspect is not currently well documented, and extending the JACK language in this way is difficult without extensive support from Agent Oriented Software.

1.3 Platform

In this section, we briefly discuss features and properties of the JACK platform including tool support for design, programming and debugging, as well as support for various forms of communication and integration.

1.3.1 Available tools and documentation

According to a recent survey of agent researchers, the areas that were seen as most desirable to be supported by third party tools were “Integrated Development Environments, Debugging tools, and parsers/language tools” [57]. JACK addresses the first two areas by providing an integrated development environment, and a range of debugging tools.

The JACK Development Environment (JDE) (see figure 1.6) allows the developer to create agents, events, plans, beliefsets etc. by dragging and dropping, rather than typing `#` declarations. The JACK skeleton code for the entities is automatically generated. The JDE also provides a Graphical Plan Editor (on the right side of figure 1.6) which allows the bodies of plans to be specified using a graphical notation, rather than textual code.

The JDE also includes a Design Tool (middle of figure 1.6) which allows overview diagrams in the style of Prometheus [43] to be drawn. This can be used to create the system’s structure by placing entities onto the canvas and linking them together. It can also be used to create an overview of an existing system by adding entities to a canvas, in which case the links between entities are automatically added. The JDE maintains consistency between the design diagrams and the underlying model, and therefore with the generated code.

JACK provides a number of debugging tools. The simplest is a textual trace of processing steps which is enabled from the command line. This can be configured to show various types of steps: changes to beliefsets, events being posted and processed, messages being sent and received, and steps in plans. Although this information is easy to obtain, it is obtained from a single run-time instance of the JACK platform, and is therefore less useful for debugging distributed systems of agents.

For debugging distributed agents *interaction diagrams* are more useful. An interaction diagram graphically displays messages sent between agents. A single interaction diagram can collect and display messages from agents across a distributed system.

Interaction diagrams depict the messages between agents. However, when debugging, it is also useful to be able to trace the internal execution of agents. JACK provides graphical plan tracing, which traces the execution of plans that have been specified using the Graphical Plan Editor. When a plan begins executing its graph is shown, and as the plan executes the currently executing node is highlighted. The graph also shows the values of the plan's variables and parameters. The execution of the agent can be controlled: it can be run as normal, single-stepped, or stepped with a delay in between steps.

The next version of JACK will also provide an additional debugging tool: a browser that allows the state of agents (including their beliefs and active tasks) to be inspected.

All of these development, design and debugging tools – as well as the JACK language, and other facilities such as JACK's support for teamwork, the Webbot interface to JSP, and JACOB (see next section) – have clear and extensive documentation. Additionally, JACK's documentation package also includes "practicals": a tutorial sequence introducing JACK.

1.3.2 Standards compliance, interoperability and portability

There are many approaches to communication and integration, such as CORBA, HLA, Java RMI and FIPA. Consequently, JACK's approach to communications is agnostic. While a lightweight communications infrastructure is provided, and can be used out-of-the-box, it is also possible to extend and/or replace JACK's communications infrastructure.

We begin by discussing JACK's lightweight communications infrastructure including a discussion of JACOB. We then look at an example of extending JACK to make it FIPA-compliant. Note that this extension is not part of the JACK distribution: JACK is a commercial product,

and since most agent systems today are not open and are not FIPA-compliant, there is limited demand by customers of Agent Oriented Software to make JACK FIPA-compliant. Rather, it is more important to be able to integrate JACK code with existing code in Java and C++ (which is supported by JACOB), and with existing applications such as databases (supported by JACOB using JDBC), web servers (supported by Webbot using JSP), and graphical user interfaces (provided by Java libraries such as AWT or Swing).

JACK's lightweight communications mechanism supports sending messages between agents. These messages can contain Java objects which are serialised by the sender and "reconstituted" by the recipient of the message. JACK provides a number of mechanisms for serialising objects: Java's serialisation can be used, but this tends to produce large messages, and only supports communication with other Java software. Alternatively, JACOB provides more compact serialisations, and allows objects to be "reconstituted" by Java or C++ programs. JACOB provides a number of serialisation formats: a plain ASCII format that is compact yet human readable, a binary format which is more compact, an XML format, and a JDBC format.

When the recipient of a message is in the same Java process as the sender, then the message is addressed simply using the name of recipient agent. However, JACK supports flexible distribution of agents: it is possible to have multiple agents per Java process, to have agents distributed in different Java processes (which can be on different machines), or to flexibly mix these. This flexible distribution requires a slightly more sophisticated addressing scheme than simply using agent names, and JACK introduces the concept of a *portal*. Roughly speaking, a portal can be thought of as a handle on a Java process, and sending a message to an agent at another portal is done by addressing the agent as *agent-name@portalname*. Each portal acts as a name server for other portals, i.e., each portal keeps track of the addresses of other portals.

Thus, JACK's provided communication infrastructure supports communication amongst flexibly distributed agents, as well as between agents and existing software written in Java or C++.

We now briefly describe a third-party extension to JACK which supports building FIPA-compliant JACK agents. The FIPA JACK plugin⁸ [59] was developed at RMIT University and was used as the basis for its AgentCities platform. The plugin provides FIPA compliant services, specifically an Agent Management System (AMS), Directory Facilitator

⁸Available from <http://www.cs.rmit.edu.au/agents/protocols/>

(DF) and Message Transport Service (MTS). The plugin also provides a new agent base class (FIPAAgent). Agents which extend this class automatically register with the AMS, and are able to send and receive FIPA-compliant messages. The FIPA JACK plugin also includes a GUI for examining the agents that are registered with the AMS and for sending messages for testing purposes.

1.3.3 Other features of the platform

JACK is efficient: it allows flexible distribution of agents, with multiple agents sharing a Java process. It also allows for many agents to run on a single machine, while still supporting distributed agent systems across machines. Benchmarking⁹ on an average PC running Linux shows that over 1000 agents can be created per second, and that 100,000 messages can be sent per second (within the same Java process).

These benchmarks are supported by a recent paper [56] which compared and benchmarked a number of agent platforms, including JACK (version 3.51), JADE, FIPA-OS and Zeus. It found JACK to be by far the fastest platform. JACK was also found to have the lowest memory requirement per-agent when creating 100 agents.

JACK is compact enough to be run on limited hardware. It has been demonstrated on a Psion 5mx, and, for a recent demonstration involving an Unmanned Aerial Vehicle, JACK was run on a Hewlett-Packard iPAQ PDA.

1.4 Applications supported by the language and/or the platform

Application areas for JACK can be loosely categorised as:

- *Autonomous systems* which operate independently (or mostly independently) from humans. For example, Unmanned Air Vehicles [32] and Holonic manufacturing [17, 18].
- *Modelling human-like decision making*. This takes advantage of the basis of the Belief-Desire-Intention model in human folk psychology [5]. Typically, this application category involves simulation of humans [19, 22, 41].
- *Decision support* applications where the system assists humans in making decisions. For example the Collection Plan Management

⁹<http://www.agent-software.com/shared/products/faq.html>

System (CPMS) [33] provides human decision makers with a number of possible plans.

- *Architectural “glue”* where a system is structured as a collection of autonomous agents in order to obtain the reduced coupling and improved maintainability associated with this architectural style. For example, the weather alerting system developed for the Australian Bureau of Meteorology [17, 38].

These are just some areas where JACK has been used. Other applications of agents where JACK could be used as an implementation platform include electronic commerce, business process modelling, and entertainment.

We now describe a number of applications developed in JACK. We have chosen to describe applications that illustrate different ways in which JACK has been used, and which have been described in the literature.

Many of JACK’s applications are military: usually associated with logistics (planning) and simulation, rather than with battlefield use. One such application is the Collection Plan Management System (CPMS) (see [33] and [23, Section 6.1]), which assists human in planning the deployment of surveillance and reconnaissance resources. The system comprises a database with information on the terrain, the available resources, and the tasks to be carried out; a visualisation module; and a planning system written in JACK. The planning system presents a number of possible plans for assessment by the human experts. The JACK planner is structured as a collection of agents mirroring the existing command and control (C2) structure, i.e., there is an agent for each entity (brigade, company, platoon, etc.) that constructs plans for the resources that it controls. Another application written in JACK, which concerns planning the deployment of military resources, in this case aircraft, is described by Marc *et al.* [37].

Another area where JACK has been used is as architectural “glue” to connect together components of a system. By structuring a system as a collection of agents, one obtains a system that is more loosely coupled, and that is easy to modify and extend. One example of this application of JACK is the alerting system developed for the Australian Bureau of Meteorology (see [38] and [17, Section 5]). The system receives information from a range of sources including storm predictions, current observations from automated weather stations, predictions issued for the area around airports and information about bush fires. Various conditions, such as discrepancies between forecasts and observations, are checked for and alerts are generated. The system is structured as a multi-agent

system where agents subscribe to information providers. Experiences with extending this system have been positive, for example extending the system to deal with a new type of information source only took a number of days.

A basic property of agents is that they are autonomous, and so a natural application area for JACK is developing software that operates autonomously. One example is the recent use of JACK on an Unmanned Aerial Vehicle (UAV) [32]. The role of JACK is not to control the vehicle directly, but rather to provide higher-level decision-making about what to do next, e.g., where should the UAV fly to? JACK's ability to deal with failure and to flexibly achieve goals is crucial in providing the UAV with a decision making capability that allows it to be independent and robust. In addition, JACK's support for teams can be used to allow multiple vehicles to cooperate in achieving their goals; for example, one UAV might act as a decoy allowing another UAV with a video camera to approach undetected. A feasibility demonstration of JACK onboard a UAV has been done¹⁰ and development of team-based UAV control is ongoing, with flight testing scheduled for early 2005.

Finally, *Holonic manufacturing* is another application area where JACK has been used to develop autonomous software. In this case, the software controls a manufacturing cell [17, 18]. The challenges in agent-based manufacturing are to support more flexible manufacturing — for example to allow custom orders and changes to orders — and to be robust, i.e., to deal appropriately with a range of issues such as shortage of parts and failure of manufacturing equipment.

1.5 JACK: A Platform for Research

In addition to being aimed at industrial application development, JACK has also been found to be suitable as a basis for research. We describe this here for two reasons, firstly because one the goals of JACK is to “enable further applied research” [23], and secondly because this research has, in some cases, involved extending JACK, and so it shows that JACK can be easily extended.

One area of research concerns making BDI agents more intelligent, or at the very least more rational. One issue that is shared by BDI platforms is that although a BDI agent may have multiple goals that are being pursued at a given time, no reasoning is done about the interaction between the goals. In a sequence of papers, Thangarajah *et al.* [52–55]

¹⁰<http://www.agent-software.com/shared/resources/pressReleases/Avatar-JACK-F040706USb.pdf>

described an extended BDI execution cycle which incorporates reasoning about the interactions, both negative and positive, between concurrent goals. This extended BDI cycle was implemented in JACK [51].

Another strand of research that has focused on JACK's execution model is the work in [39–41] which looks at (i) making the decision making of JACK agents more “human-like” by adding selection of plans based on recognition of situations, and learning from mistakes [39]; and (ii) adding psychologically-plausible variability in decision making by incorporating factors such as fatigue, time-of-day and human perception processes (how human vision tracks objects) [40, 41]. The latter work is being applied to simulate changes of behaviour in military personnel [19].

JACK has also been extended with look-ahead planning [13] by integrating with JSHOP, an HTN (Hierarchical Task Network) planner written in Java.

Finally, work by Poutakidis *et al.* [44, 45] has proposed and implemented on top of JACK a debugger that automatically detects errors by monitoring messages between agents and raising an alert if the messages do not conform to the interaction protocol that is meant to be followed.

1.6 Final Remarks

We have presented the JACK language and platform, including the unique features of the JACK language, such as teamwork, and the tool support that is provided by JACK.

At RMIT University we have taught an undergraduate course on agent-oriented programming and design for a few years¹¹. The course, which runs in a single 12 week semester, covers an introduction to agents, the Prometheus agent-oriented software engineering methodology, and JACK¹². During the course, the students complete a design and implement an agent system, working in teams of 1-3 students. Typical projects have included a group calendar system, a library management system, and a stock trader simulation. Our experience has been that the vast majority of the students manage to learn JACK and that the students use JACK effectively by the end of the course, i.e., that their code is agent-oriented, not just object-oriented code wrapped in plans.

¹¹The course was first taught in 2001, and has been taught subsequently in 2002, 2003 and 2004.

¹²JACK is covered in four lectures.

Acknowledgments

I would like to thank Leanne Veitch (of Agent Oriented Software) for proof-reading a draft of this paper, Nick Howden (of Agent Oriented Software) for comments on a draft of this paper, and Ralph Rönquist (of Agent Oriented Software) for writing the summary appendix. I would like to acknowledge the support of Agent Oriented Software and of the Australian Research Council under grant LP0453486 (“Advanced Software Engineering Support for Intelligent Agent Systems”). Finally, I would like to thank the people who have taught the course on Agent Oriented Programming and Design at RMIT: Lin Padgham, Wei Liu, and David Poutakidis.

Appendix: Summary

- 1(a) The JACK language supports BDI style practical reasoning as well as forward-directed inference reasoning, and allows for various agent concepts such as mental attitudes, deliberation, adaptation, reactive and proactive behaviour. There is a JACK extension towards a Cognitive Architecture, for inclusion of cognitive parameters and variations to the reasoning processes, and for modelling of cognitive influences by behaviour moderators.
- 1(b) JACK provides high-level primitives for communication between agents. Communication is peer-to-peer, and does not include broadcast or multi-cast addressing.
- 1(c) JACK is not intended for mobile agents.
- 1(d) JACK is an easy-to-use programming language in the BDI family, and the tool suite includes graphical programming tools both for program design and for decision logic.
- 1(e) The JACK language has clear and precise (but not formal) semantics.
- 1(f) JACK is a full programming language well suited for a variety of agent applications.
- 1(g) JACK allows new program elements to be defined in a systematic way, through compiler plugins.
- 1(h) This has not been investigated.
- 1(i) The JACK language is a full-flavoured programming language that combines the logic oriented BDI style with the object-oriented Java style, and it further includes programming elements providing increased support for abstraction, modularisation, information hiding and generic programming.
- 1(j).i JACK is fully integrated with Java, and it also includes integration mechanisms for combining JACK agents with C++ programs.
- 1(j).ii JACK is fully integrated with Java.
- 2(a).i JACK is well documented through a range of manuals and practicals, and is easily installed via the downloadable installer.
- 2(a).ii JACK runs on all Java platforms from 1.1.3, and has been run on PDAs (Psion 5mx and an HP iPAQ).
- 2(b) The JACK platform is itself proprietary, but includes the standard architectural elements, and there are FIPA wrapper extensions.
- 2(c) JACK is built to be open, with a range of “hooks” at various levels to simplify extensions. JACK is not open source.
- 2(d).i JACK comes with several mechanisms for logging and debugging of JACK agent execution.
- 2(d).ii The JACK package includes manuals in PDF and HTML format.
- 2(d).iii JACK includes a development tool.
- 2(e) JACK is fully integrated with Java and all Java tools can be used.
- 2(f) JACK is not tied to any specific operation environment.
- 2(g).i A single process can host thousands of JACK agents.

- 2(g).ii** JACK is a fully supported commercial product.
- 2(h).i** JACK supports open multi-agent systems and heterogeneous agents.
- 2(h).ii** JACK includes a language extension for team-oriented programming, which simplifies coordinated activity and distributed control. The JACK Teams model includes role declarations and hierarchical, dynamic teams.
- 2(h).iii** JACK does not include any pre-programmed libraries of JACK code.
- 3(a)** JACK is being used for several real-world, industrial applications.
- 3(b)** The BDI style programming is well suited to strategic robot control, as used in manufacturing plants, autonomous vehicles, and simulation, as well as business logic applications, including application of analytical procedures, compliance processes, and situated decision making.

References

- [1] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.
- [2] Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors. *Methodologies and Software Engineering for Agent Systems*. Kluwer Academic Publishing (New York), 2004.
- [3] Rafael H. Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking agentspeak. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 409–416. ACM Press, 2003.
- [4] Rafael H. Bordini and Álvaro F. Moreira. Proving BDI properties of agent-oriented programming languages. *Annals of Mathematics and Artificial Intelligence*, 42(1-3):197–226, September 2004.
- [5] Michael E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.
- [6] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos, and Anna Perini. Tropos: An agent-oriented software development methodology. Technical Report DIT-02-0015, University of Trento, Department of Information and Communication Technology, 2002.
- [7] Paolo Busetta, Nicholas Howden, Ralph Rönquist, and Andrew Hodgson. Structuring BDI agents in functional clusters. In *Agent Theories, Architectures, and Languages (ATAL-99)*, pages 277–289. Springer-Verlag, 2000. LNCS 1757.
- [8] Paolo Busetta, Ralph Rönquist, Andrew Hodgson, and Andrew Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia, 1998. Available from <http://www.agent-software.com>.

- [9] L. Cernuzzi and G. Rossi. On the evaluation of agent oriented modeling methods. In *Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, pages 21–30, Seattle, November 2002.
- [10] Dean Christopher Ho Mok Cheong. An empirical investigation of teamwork infrastructure for autonomous agents, October 2003. Honours Thesis, available as RMIT Computer Science technical report TR-03-2.
- [11] P. R. Cohen and H. J. Levesque. Teamwork. *Nous*, 25(4):487–512, 1991.
- [12] Khanh Hoa Dam and Michael Winikoff. Comparing agent-oriented methodologies. In P. Giorgini, B. Henderson-Sellers, and M. Winikoff, editors, *Agent-Oriented Information Systems (AOIS 2003): Revised Selected Papers*, pages 78–93. Springer LNAI 3030, 2004.
- [13] Lavindra P. de Silva and Lin Padgham. A comparison of BDI based real-time reasoning and HTN based planning. In *17th Australian Joint Conference on Artificial Intelligence*, 2004.
- [14] Scott A. DeLoach, Mark F. Wood, and Clint H. Sparkman. Multi-agent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.
- [15] Mark d’Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In M.P. Singh, A.S. Rao, and M. Wooldridge, editors, *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages*, pages 155–176. Springer-Verlag LNAI 1365, 1998.
- [16] Mark d’Inverno and Michael Luck. *Understanding Agent Systems*. Springer-Verlag, 2001.
- [17] Rick Evertsz, Martyn Fletcher, Richard Jones, Jacquie Jarvis, James Brusey, and Sandy Dance. Implementing industrial multi-agent systems using JACKTM. In Mehdi Dastani, Juergen Dix, and Amal El Fallah-Seghrouchni, editors, *First International Workshop, PROMAS 2003, Melbourne, Australia, July 15, 2003, Selected Revised and Invited Papers*, pages 18–48. Springer LNAI 3067, 2004.
- [18] Martyn Fletcher and James Brusey. The story of the Holonic packing cell. In *Agents at Work: Deployed Applications of Autonomous Agents and Multi-Agent Systems*, July 2003.
- [19] Martyn Fletcher, Ralph Rönquist, Nick Howden, and Andrew Lucas. Enigma variations – simulating changes in behaviour of British military personnel. In *SimTecT*, pages 21–26, May 2004.

- [20] M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74:1383–1398, 1986.
- [21] B. Henderson-Sellers and P. Giorgini, editors. *Agent-Oriented Methodologies*. Idea Group, 2005 (to appear).
- [22] Nick Howden, Jamie Curmi, Clinton Heinze, Simon Goss, and Grant Murphy. Operational knowledge representation—behaviour capture, modelling and verification. In *SimTecT*, May 2003.
- [23] Nick Howden, Ralph Rönquist, Andrew Hodgson, and Andrew Lucas. JACK intelligent agents: Summary of an agent infrastructure. In *Workshop on Infrastructure for Agents, MAS, and scalable MAS*, 2001.
- [24] Marcus J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 236–243, 1999.
- [25] Joshua Hutchison. Agent team programming: An evaluation of JACK teams, October 2002. Honours Thesis, available as RMIT Computer Science technical report TR-02-7.
- [26] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.
- [27] Jacquie Jarvis. JACK Intelligent Agents JACK Teams Manual, March 2004. Available with the JACK distribution.
- [28] N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.
- [29] N.R. Jennings and M.J. Wooldridge. Applications of intelligent agents. In Nicholas R. Jennings and Michael J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, chapter 1, pages 3–28. Springer, 1998.
- [30] T. Juan, A. Pearce, and L. Sterling. ROADMAP: Extending the Gaia methodology for complex open systems. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002)*, pages 3–10. ACM Press, 2002.
- [31] Jaeho Lee, Marcus J. Huber, Patrick G. Kenny, and Edmund H. Durfee. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *Proceedings of the Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS'94)*, pages 842–849, 1994.
- [32] Andrew Lucas, Peter Corke, Ralph Rönquist, Pavan Sikka, Magnus Ljungberg, and Nick Howden. Teamed UAVs – A new approach with intelligent agents. In *AIAA Unmanned Unlimited*, 2003.

- [33] Andrew Lucas, Ralph Rönquist, Nick Howden, Paul Gaertner, and John Haub. Intelligent battlespace awareness and information dissemination through the application of BDI intelligent agent technologies. In *SPIE - The International Society for Optical Engineering: Digitization of the Battlespace V and Battlefield Biomedical Technologies II*, April 2000.
- [34] Michael Luck, Peter McBurney, and Chris Preist. Agent technology: Enabling next generation computing: A roadmap for agent-based computing. AgentLink report, available from www.agentlink.org/roadmap, 2003. ISBN 0854 327886.
- [35] Michael Luck, Peter McBurney, and Chris Preist. A manifesto for agent technology: Towards next generation computing. *Autonomous Agents and Multi-Agent Systems*, 9(3):203–252, November 2004.
- [36] R. Machado and R.H. Bordini. Running AgentSpeak(L) agents on SIM_AGENT. In J.-J. Meyer and M. Tambe, editors, *Intelligent Agents VIII - Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*. Springer-Verlag LNAI 2333, August 2001.
- [37] F. Marc, A. El Fallah-Seghrouchni, and I. Degirmenciyan-Cartault. Coordination of complex systems based on multi-agent planning: Application to the aircraft simulation domain. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Second International Workshop on Programming Multi-Agent Systems: Languages and Tools (ProMAS 2004)*, pages 115–128, July 2004.
- [38] Ian Mathieson, Sandy Dance, Lin Padgham, Malcolm Gorman, and Michael Winikoff. An open meteorological alerting system: Issues and solutions. In Vladimir Estivill-Castro, editor, *Proceedings of the 27th Australasian Computer Science Conference*, pages 351–358, Dunedin, New Zealand, 2004.
- [39] Emma Norling. Folk psychology for human modelling: Extending the BDI paradigm. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 202–209, New York, July 2004.
- [40] Emma Norling and Frank E. Ritter. Embodying the JACK agent architecture. In Markus Stumptner, Dan Corbett, and Mike Brooks, editors, *AI2001: Advances in Artificial Intelligence. 14th Australian Joint Conference on Artificial Intelligence*, pages 368–377. Springer LNAI 2256, December 2001.
- [41] Emma Norling and Frank E. Ritter. Towards supporting psychologically plausible variability in agent-based human modelling.

- In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 758–765, New York, July 2004.
- [42] James J. Odell. Objects and agents compared. *Journal of Object Technology*, 1(1):41–53, May-June 2002.
- [43] Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004. ISBN 0-470-86120-7.
- [44] David Poutakidis, Lin Padgham, and Michael Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS’02)*, pages 960–967. ACM Press, July 2002.
- [45] David Poutakidis, Lin Padgham, and Michael Winikoff. An exploration of bugs and debugging in multi-agent systems. In *Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 628–632, Maebashi City, Japan, 2003.
- [46] Talal Rahwan, Tarek Rahwan, Iyad Rahwan, and Ronald Ashri. Agent-based support for mobile users using AgentSpeak(L). In P. Giorgini, B. Henderson-Sellers, and M. Winikoff, editors, *Agent-Oriented Information Systems (AOIS 2003): Revised Selected Papers*, pages 45–60. Springer LNAI 3030, 2004.
- [47] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John Perrame, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW’96)*, pages 42–55. Springer Verlag, 1996. LNAI, Volume 1038.
- [48] Arnon Sturm and Onn Shehory. A comparative evaluation of agent-oriented methodologies. In Bergenti et al. [2], chapter 7.
- [49] Jan Sudeikat, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Evaluation of agent-oriented software methodologies: Examination of the gap between modeling and platform. In Paolo Giorgini, Jörg Müller, and James Odell, editors, *Agent Oriented Software Engineering (AOSE)*, 2004.
- [50] Milind Tambe and Weixiong Zhang. Towards flexible teamwork in persistent teams: Extended report. *Journal of Autonomous Agents and Multi-agent Systems*, 3:159–183, 2000. Special issue on “Best of ICMAS 98”.

- [51] John Thangarajah and Lin Padgham. An empirical evaluation of reasoning about resource conflicts. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1298–1299, July 2004.
- [52] John Thangarajah, Lin Padgham, and James Harland. Representation and reasoning for goals in BDI agents. In *Australasian Computer Science Conference*, 2002.
- [53] John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, 2003.
- [54] John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting and exploiting positive goal interaction in intelligent agents. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003)*, pages 401–408. ACM Press, 2003.
- [55] John Thangarajah, Michael Winikoff, Lin Padgham, and Klaus Fischer. Avoiding resource conflicts in intelligent agents. In F. van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence*. IOS Press, 2002.
- [56] Pavel Vrba. JAVA-Based agent platform evaluation. In Vladimír Mařík, Duncan McFarlane, and Paul Valckenaers, editors, *Holonic and Multi-Agent Systems for Manufacturing (HoloMAS 2003)*, pages 47–58. Springer, LNCS 2744, 2004.
- [57] Steven Wilmott, Omer Rana, Karl-Heinz Krempels, Peter McBurney, and Georg Weichart. Networked agents: Towards large-scale deployment of agents in open networked environments (NET AGENTS). *AgentLink News*, 16:16–17, December 2004. Available from <http://www.agentlink.org/newsletter/>.
- [58] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, 2002.
- [59] Kenichi Yoshimura. FIPA JACK: A plugin for JACK Intelligent Agents, 2003. Available from <http://www.cs.rmit.edu.au/agents/protocols/>.

- [60] F. Zambonelli, N.R. Jennings, and M. Wooldridge. Developing Multiagent Systems: The Gaia Methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, July 2003.
- [61] Franco Zambonelli and Andrea Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, November 2004.

Index

- Agent platform, 2
- Agent-oriented programming language, 2
- AgentSpeak(L), 9
- Autonomous, 1

- BDI, 1
- BDI gap, 9
- Belief, 1
- Belief-Desire-Intention, 16
- Beliefset, 6

- Capability, 6, 10
- CORBA, 14

- Debugging Tools, 12
 - Browser, 14
 - Graphical plan tracing, 14
 - Interaction Diagrams, 14
 - Tracing, 12
- Desire, 1
- dMARS, 7

- Event, 6

- FIPA, 3, 14
- FIPA JACK, 15
- Flexibility, 2
- Folk psychology, 16

- Gaia, 2
- Goals, 1

- HLA, 14

- Intention, 1
- Interaction diagrams, 14

- JACK Development Environment, 12
 - Design Tool, 12
 - Graphical Plan Editor, 12
- JACOB, 15
- Jadex, 4, 10
- JAM, 4, 7
- Jason, 4, 9
- Java, 3
- JDBC, 15
- JDE, *see* JACK Development Environment
- JSHOP, 10, 19

- MaSE, 2
- Methodology, 2

- Plan, 1, 6
- Portal, 15
- Proactive, 1
- Prometheus, 2
- PRS, 7

- Reactive, 1
- Relevance condition, 8
- RMI, 14
- ROADMAP, 2
- Robustness, 2
- Roles, 10

- Social, 1

- Teams, 2, 10
- Tropos, 2

- UM-PRS, 7

- View, 6