3APL Platform

2nd October 2003

Masters' thesis Computer Science

Utrecht University

E.C. ten Hoeve

INF/SCR-03-02

Supervisors: dr. M. M. Dastani
dr. F.P.M. Dignum
Prof. dr. J.-J. Ch. Meyer

Abstract

The 3APL platform is an experimental multiagent platform, designed to support the FIPA specifications. It provides a graphical interface in which a user can develop and execute agents using several tools, such as a syntax-colored editor and several debugging tools. The platform also allows for communication among agents present on the platform as well as other agents hosted on another 3APL platform. This thesis describes the design and implementation of this platform. It also compares several existing platforms, namely JADE,JACK and ZEUS with the 3APL platform on purpose, communication and the phases of traditional software development (analysis, design, implementation, deployment).

Acknowledgements

Many thanks to dr. Mehdi Dastani for his support, comments and ideas. I would also like to thank dr. Frank Dignum for his input and suggestions. Finally I would like to thank Ir. Meindert Kroese for his assistance in integrating and adapting the existing 3APL software to the agent platform.

Contents

1 Introduction **9**

2 What is 3APL? **11**
2.1 Beliefs . . . 11
2.2 Actions . . . 12
2.3 Goals . . . 12
2.4 Practical reasoning rules . . . 13
2.5 Semantics of 3APL . . . 13
2.6 Deliberation cycle . . . 14
2.7 An example program. . . . 15

3 What is a platform ? **17**
3.1 Introduction . . . 17
3.2 FIPA Standards . . . 17
3.2.1 AMS . . . 18
3.2.2 Agent . . . 18
3.2.3 Naming . . . 20
3.2.4 Directory Facilitator . . . 20
3.2.5 Message Transport System . . . 20
3.2.6 Communication language . . . 20
3.3 Features of the 3APL platform . . . 21
3.3.1 AMS . . . 21
3.3.2 Agent . . . 22
3.3.3 Naming . . . 22
3.3.4 Directory Facilitator . . . 22
3.3.5 Message Transport System / Agent Communication Channel . . . 23
3.3.6 Communication Language . . . 23
3.4 Example . . . 24

4 Manual and implementation **27**
4.1 Software Requirements . . . 27
4.2 Main . . . 27
4.2.1 Messages window . . . 28
4.2.2 Agent properties . . . 28
4.3 Sniffer . . . 29
4.4 Message sender . . . 30
4.5 Templated agents . . . 31
4.6 Implementation . . . 32
4.6.1 General architecture . . . 32
4.6.2 Agent class . . . 32
4.6.3 AMS . . . 34

5 Comparison **37**
5.1 JADE . . . 37
5.2 ZEUS . . . 38
5.3 JACK . . . 40
5.4 Comparison 3APL . . . 41

6 Conclusion and Future work **43**

1 Introduction

Intelligent agents and multiagent systems are the subject of a very active and rapidly growing research field in both artificial intelligence and computer science. Cognitive agents are specific types of intelligent agents which have complex mental states, are made up out of informational components, like beliefs, and have motivational components, like goals. Agents are supposed to act pro-actively, which means that they try to fulfill goals. Agents are also supposed to be reactive, i.e. are able to respond to their environment, and may have reflective capabilities related to their beliefs and goals. Agents can be used as a tool to analyse many complex applications, such as electronic auctions, electronic markets, or autonomous robots [34].

There are several implementation languages for cognitive agents, most notably Agent Speak(L) [19], ConGolog [8], Agent-0 [24], Ψ-calculus [12] and 3APL [11], which have been proposed to implement agents that are specified in terms of BDI [18], KARO [32] or other specification languages.

In order to successfully implement the aforementioned applications it becomes necessary for agents to communicate with each other. A common solution is to provide a so-called agent platform on which the agents can reside and communicate. A platform should provide basic services, such as the actual communication between agents which may run on other platforms by means of a transport layer. But it also supplies information to agents such as which agents have registered themselves to the platform, or what services they provide. The specifications of an agent platform have been proposed by a non-profit organization, FIPA (The Foundation for Intelligent Physical Agents) [6]. FIPA is an international organization that is dedicated to promoting the industry of intelligent agent community by openly developing specifications supporting interoperability among agents and agent-based applications. This occurs through open collaboration among its member organizations, which are companies and universities that are active in the field of agents. Several platforms that adhere to the FIPA specifications already exist, for instance ZEUS [17], JACK [1] and especially JADE [26], which is claimed to have a great compliance to the FIPA specifications.

However, on these platforms it is very hard to define a 'mapping' between the cognitive agent specifications (Goal, Belief and Desire/Intentions) to the actual implementation of an agent on the platform itself. 3APL does provide an intuitive mapping; a multiagent platform that makes it easier to develop and test 3APL agents does not exist yet.

We decided to develop this experimental platform with the ability to implement and execute 3APL agents. In addition, we will provide the possibility of using templates and debugging tools to facilitate the development of these agent systems.

This thesis describes the design and implementation of a cognitive agent platform for 3APL agents. In chapter 2 we will give an introduction to the 3APL programming language. We will conclude this chapter with an example of an agent, written in 3APL. In chapter 3 we will look at what is necessary for multiagent systems, more specifically, what is the FIPA specification for an agent platform. We will look at the features of the 3APL platform in relation to the FIPA specification. This chapter continues with how the 3APL language has been extended to allow agent communication. This chapter is concluded with an example of communicating agents. Chapter 4 describes the platform from a user's point of view. The visual aspects and features will be described here. In chapter 5 we will look at other existing Java agent platforms, i.e. ZEUS, JACK and JADE, and we will compare them to the 3APL platform. Chapter 6 concludes this thesis and describes future work.

2 What is 3APL?

An interesting metaphor in the programming community is that of an intelligent agent. The idea is to view programs as intelligent agents acting on our behalf. By using the metaphor of intelligent agents the programmer views programs as entities, which have a mental state consisting of beliefs, intentions, plans, etc. The computational behaviour of an agent is explained in terms of the decisions it makes on the basis of its beliefs and goals. Although there does not yet exist a formal definition of an intelligent agent, the following characteristics are common [11]:

- Agents have a complex internal mental state consisting of beliefs, desires, plans and intentions which may change over time;
- Agents have to act to achieve their goals (i.e. be goal-directed) and should also respond to changes in their environment within a timely manner (i.e. have reactive capabilities);
- Agent have reflective reasoning capabilities.

Beliefs represent knowledge of the outside world, i.e. the environment in which an agent resides. Because agent environments can change, it is necessary to be able to revise agents' beliefs.

As agents are goal-directed, they need to find a plan with which it can achieve its goals. It also becomes necessary for agents to have the ability to monitor its success or failure of realising goals and respond to this by for instance reconsidering non-achievable goals or possibly adopting new ones. This requires *practical reasoning*.

3APL (An Abstract Agent Programming Language) is a programming language for implementing these concepts and mechanisms. It provides programming constructs for implementing agent's beliefs, goals, basic capabilities such as belief updates or actions, and a set of practical reasoning rules through which agent's goals can be updated or revised. A 3APL agent is defined by implementing its beliefbase, goalbase, capabilitiesbase and rulebase. Thus, the following definition holds:

Definition 1 *A 3APL agent is a tuple (B,G,P,A) where B is a set of beliefs (called beliefbase), G is a set of goals (called goalbase), P is a set of practical reasoning rules (called rulebase) and A is a set of basic actions (called actionbase).*

The remainder of this chapter expands on the concepts of beliefs, actions and goals and how they are represented in 3APL. The sources [31, 33] have been use as a reference for this chapter. We conclude this chapter with an example program, which shows how the programmer can use these concepts to implement an agent.

2.1 Beliefs

As stated in the introduction, an agent is assumed to have beliefs about its task and its environment. These beliefs are represented in 3APL by a set of Prolog-like formulas, that is to say a subset of first-order predicate language. For instance a belief that there is an object on a certain location (X1, Y1), can be represented as object(X1, Y1).

Definition 2 (Programming constructs for beliefs) *Given a set of domain variables and functions, the set of domain terms is defined as usual. Let t_1 ,..., t_n be terms referring to domain elements and Pred be a set of domain predicates, then the set of programming constructs for belief formula, BF, is defined as follows:*

- $p(t_1,...,t_n) \in BF$.
- *if* φ , $\psi \in BF$, *then* $\neg\varphi$, $\psi \wedge \varphi \in BF$.

An example of representing information about the environment is an agent with the task to pick up a box at a certain location. It will have beliefs such as box(X, Y) and NOT carrybox(self). If the agent picks up the box, the agent's mental state and the state of the environment change. The belief NOT carrybox(self) denotes that the agent believes he is not carrying a box, so this has to change in carrybox(self). The beliefs NOT carrybox(self) and box(X,Y) need to be removed from the beliefbase in order to make the agent's view of the world consistent.

Definition 3 (Beliefbase) *All variables in the BF formula are universally quantified with maximum scope. The beliefbase module of a cognitive agent is the following programming construct:*
BELIEF BASE $= \{ \varphi \mid \varphi \in BF \}$.

2.2 Actions

The most primitive action that an agent is capable of performing is a basic action. Basic actions are also called capabilities. In general, an agent uses basic actions to manipulate its mental state and its environment. Before performing a basic action certain beliefs should hold and after the execution of the action the beliefs of the agent will be updated. These basic actions are the only entities of 3APL that can change the beliefs of an agent.

Definition 4 (Programming constructs for basic actions) *Let* α *be an action name with parameters X and let* φ , $\psi \in BF$. *Then, programming constructs for basic actions have the form:* $\{\varphi\}\ \alpha\ (X)\ \{\psi\}$.

An action can only be performed if certain beliefs hold. These are called the pre-conditions of an action. Take for example an agent that wants to fetch a ball, with the basic action GetBall(X0, Y0). Before this action can be executed, there should be a ball present at the position (X0, Y0) and the agent should also be at that position and not already carrying the ball itself. We define this with the following beliefs as the precondition of GetBall(X0,Y0):

{ agent(X0, Y0), ball(X0, Y0), NOT have_ball(self) }

After execution of the action its post-condition must be true. This means the agent's beliefs are updated. For example, after the fetch-action the following beliefs will hold:

{ have_ball(self) }

Definition 5 (Basic actions) *The basic action module of a cognitive agent is the following programming construct:*
BASIC ACTIONS= { C / C is a basic action }.

2.3 Goals

A 3APL-agent has basic and composite goals. There are three different types of basic goals: basic action, test goal and the predicate goal. A basic goal can be a basic action that can be executed by the agent. The test goal allows the agent to evaluate its beliefs (a test goal checks whether a belief formula is true or false). Testing if the agent is carrying a box will look like carrybox(self)?. This type of goal is also used to bind values to variables like the assignment in regular programming languages. When the test goal is used with a variable as a parameter, the variable is instantiated with a value from a belief formula in the beliefbase. The third type is a predicate goal, which can be used as a label for a procedure call. The procedure itself is defined by a practical reasoning rule (see further for the definition of practical reasoning rules). From these three types of basic goals we can construct composite goals by means of the sequence operator, the conditional choice operator and the 'while' operator.

Definition 6 (Programming constructs for goals) *Let BA be a set of basic actions, BF be a set of belief sentences,* $\pi, \pi_1, \ldots \pi_n \in GOAL$ *and* $\varphi \in BF$. *Then, the set of programming constructs for 3APL goals (GOAL) can be defined as follows:*

- *BactionGoal: BA* $\in$ *GOAL.*
- *PredGoal: BF* $\in$ *GOAL.*
- *TestGoal: if* $\varphi \in$ *BF, then* $\varphi? \in$ *GOAL.*
- *SkipGoal: skip* $\in$ *GOAL.*
- *SequenceGoal: if* $\pi_1,\ldots,\pi_n \in$ *GOAL, then* π_1 ; ... ; $\pi_n \in$ *GOAL*
- *IfGoal: IF* φ *THEN* π_1 *ELSE* $\pi_2 \in$ *GOAL.*
- *WhileGoal: WHILE* φ *DO* $\pi \in$ *GOAL*

A special case is the recently added JavaGoal. This goal enables the programmer to use an external Java class and call its methods. Each method is assumed to to return a list of wellformed formulae. Note that the list may be an empty list.

Definition 7 (Programming construct for JavaGoal) *Let C be a set of Java classes. Let class c'* $\in$ *C have a method M(*x_1, ..., x_n*) with parameters* x_1, ..., x_n *and L be a Prolog list which will contain the result of the method call. Then the set GOAL of 3APL goals is extended with the programming construct for a JavaGoal which is defined as follows: JavaGoal: Java(C, M(*$x_1,\ldots,x_n$*), L)* $\in$ *GOAL*

2.4 Practical reasoning rules

Practical reasoning rules are at the heart of the functioning of 3APL agents. They can be used to generate some sort of reactive behavior, to optimise the agent's goals or to revise the agent's goals to get rid of unachievable goals or blocked basic-actions. They can also be used to define predicate goals (i.e. procedure calls). To allow for the dynamic matching of rules, goal variables are needed. These can be used as place-holders for goals and are defined using the set GVar.

Definition 8 (Goal variables) *Let GVar be a set of goal variables. Then VGOAL is defined by extending definition 2.3.1 of GOAL with the following clause:*
If X $\in$ *GVAR, then X* $\in$ *VGOAL.*

Definition 9 (Programming constructs for practical reasoning rules) *Let* $\pi_h, \pi_b \in VGOAL$ *and* $\psi \in BF$*, then a practical reasoning rule is defined as follows:*
$\pi_h \leftarrow \psi \mid \pi_b$

This practical reasoning rule can be read as follows: if the agent's goal is π_h and the agent believes ψ , then π_h can be replaced by π_b. With the restriction that any goal variable occurring in π_b also occurs in π_h. Note that possibly π_h and π_b can be empty. If the head is empty, the rule can be used to create new goals, separate from the current goals. And if the body of the rule is empty, the rule can be used to drop goals from the goalbase.

2.5 Semantics of 3APL

The operational semantics for the 3APL language is defined by means of a transition system. A transition system consists of a set of derivation rules for deriving transitions which are associated with an agent. A transition is a transformation from one configuration to another and it corresponds to a single computation step. The semantics of 3APL specify transitions between the agent's states by means of transition rules. The state of an agent is defined as follows:

Definition 10 (State of an agent) *The state of a 3APL agent is defined as a 4-tuple* $<\Pi$, σ, Γ , $\theta>$, *where* Π *is the set of the agent's goals,* σ *is the agent's beliefs,* Γ *is the set of practical reasoning rules, and* θ *is a substitution consisting of binding of variables that occur in the agent's beliefs and goals. (Because* Γ *does not change by transitions, we omit this part for the remainder of the section.)*

The binding of variables θ is generated and updated by transitions that are responsible for the execution of test goals and the application of practical reasoning rules. The variable binding θ is passed through successive transitions. Before applying the practical reasoning rules the guard and the head of the rule, which may contain domain variables, needs to be unified with a belief formula and a goal. Both these unifications result in substitutions, which are passed on to the next states. We do not reintroduce all the transition rules, but only two of them to illustrate how they work. See [11] for other transition rules.

Definition 11 (agent transition) *Let* $\Pi=\{\pi_0,\ldots, \pi_i,\ldots,\pi_n\}$, *and* θ *and* θ*' be ground substitutions. Then,*

$$\frac{<\pi_i,\sigma,\theta>\rightarrow<\pi_i',\sigma',\theta'>}{<\{\pi_0,\ldots,\pi_i,\ldots,\pi_n\},\sigma,\theta>\rightarrow<\{\pi_0,\ldots,\pi_i',\ldots,\pi_n\},\sigma',\theta'>}$$

The above definition states that a transition on a set of goals can be accomplished by transitions on individual goals separately.

Definition 12 (transition rule for tests) *Let* γ *be a ground substitution such that dom(*γ*)*$\subseteq$*Free(*$\varphi\theta$*). Then,*

$$\frac{\sigma\models\varphi\theta\gamma}{<\varphi?,\sigma,\theta>\rightarrow<E,\sigma,\theta\gamma>}$$

For example, if the current beliefbase for an agent contains the proposition *position(10,9)* and no binding has been established yet for the variable X the test $position(X,9)$? could return the binding X=10. Note that it only returns this binding if there is only one *position* in the current beliefbase. If there are other alternative bindings possible, for instance $\gamma \neq \gamma'$ and $\sigma \models \varphi\theta\gamma$ and $\sigma \models \varphi\theta\gamma'$, then one of the possible substitutions is nondeterministically chosen.

2.6 Deliberation cycle

The previous sections only described the design and formal representation of the language 3APL. The next step is to build an interpreter that can execute 3APL programs. The components of a 3APL agent, i.e. beliefs, actions, goals and rules are coded in data structures. The interpreter uses a fixed procedure for selecting and executing reasoning rules, goals and capabilities. We call this the deliberation cycle, which decides which changes are made to the belief- and goalbase at each moment. These are the only structures that can change during the execution of a 3APL-program; the rulebase and the actions are static. In figure 1 we see how all the components are related to each other and how they fit together in the deliberation cycle. At the beginning of each cycle the agent checks whether there are still goals left or applicable rules. If there are still goals left, the deliberation cycle enters the goal-revision stage, where rules are tried to match the current goals. If there are, it is checked whether the guard of these rules can be derived from the current beliefbase. If there is more than one rule satisfying these conditions, then one is selected and applied to the goalbase. When the rule is applied or if there are no rules applicable, then the agent checks again whether there are goals left to execute. If so, it selects one and executes it. After the execution or if there are no goals to execute, it goes back to the beginning and starts over again.

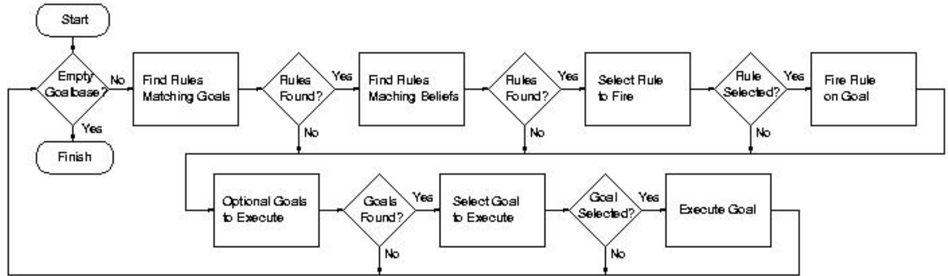

Figure 1: The deliberation cycle of the interpreter

2.7 An example program.

The following 3APL code demonstrates the concepts discussed in this chapter. The program simulates, very simplistically, a soccer player. The main goal for the agent is to win. In order to win, it must put the ball at the location of the goal. The environment for the agent is represented by its beliefbase, it knows the position of the ball, the goal and of itself. The capabilities of the agent are to move towards the ball, and to kick the ball towards the goal.

```
PROGRAM "player"
CAPABILITIES:

 { ball(X1,Y1) AND player(X0,Y0) }
    MoveToBall ()
 { NOT player(X0,Y0), player(X1,Y1) }

 { ball(X0,Y0) }
    Kick (X1,Y1)
 { NOT ball(X0,Y0), ball(X1,Y1) }

BELIEFBASE:
 player(10,10),
 ball(3,3),
 goal(0,0)

GOALBASE:
 Win()

RULEBASE:
 Win() <- goal(X1,Y1) |
 BEGIN
  WHILE ball(X0,Y0) AND NOT ((X0=X1) AND (Y0=Y1)) DO
   BEGIN
     MoveToBall();
     Kick(X1,Y1)
   END
  END
.
```

Listing 1: The source code for the soccer player.

The agent is built up from its four bases. There are two basic actions, denoted by the capabilities: MoveToBall() and Kick(). The precondition of the first one specifies the position of the player

and the ball, by a predicate *ball* which has a parameter list with two variables, X1 and Y1 and the predicate *player* which has has a parameter list with two variables, X0 and Y0. If the precondition is true, the capability can be executed and the post condition will be made true. The post condition removes the current position of the player from the beliefbase and updates the beliefbase with the position of the player at the position of the ball. The precondition of Kick() specifies the position of the ball. The postcondition removes the current position of the ball and update the beliefbase with a new position of the ball, namely the position (X1,Y1). The beliefbase consist of three belief formulas: the positions of the player, the ball and the goal. These will change during the execution of the program. The goalbase has one initial goal, Win(), which denotes the goal of the agent to win the game. When the execution starts, the rules that match this goal and also match the agent's current beliefs will be selected. The rule Win is selected, because it requires that there is a goal present. The rule is applied and the goal in the goalbase is replaced by the body of the goal of this rule. This ends the first cycle. In the second cycle, the body of Win, the While goal, is executed. While the ball is not on the position of the goal, the basic action $MoveToBall()$ is executed, followed by the basic action $Kick()$. This basic action is parameterised with the position of the *goal*, so after execution of this goal, the position of the ball is updated to be the position of the *goal*. This makes the condition of the while goal false, so the agents execution is finished, because there are no more goals to be fulfilled.

3 What is a platform ?

3.1 Introduction

In order for more realistic modelling of agents in the real world, it quickly becomes necessary for agents to communicate with each other. Different problems arise when communication is added to an agents' capabilities. For instance, what protocol should be used by an agent to communicate to another agent? Or how can an agent notify its presence to a platform? The most common solution to these problems is the introduction of a so called platform. This chapter focuses on the specifications of a platform for 3APL. First we will look at a definition of a platform according to the FIPA, and then we will describe in what way the components of the 3APL platform adhere to these standards. In order to facilitate communication, 3APL needs to be enhanced. We give an extension to the language to include communication. We conclude this chapter with an example which will demonstrate the new capabilities of 3APL agents running on the 3APL platform.

3.2 FIPA Standards

As mentioned in the introduction, the specifications of an agent platform have been proposed by the FIPA. The Foundation for Intelligent Physical Agents (FIPA) is an international organization that is dedicated to promoting the industry of intelligent agents by openly developing specifications supporting interoperability among agents and agent-based applications. This occurs through open collaboration among its member organizations, which are companies and universities that are active in the field of multiagent systems [6]. FIPA produces standard specifications for different aspects of agents, but with a focus on interoperability between agent-based systems developed by different companies and organizations which are divided over several different technical committees and working groups, each working on their own set of specifications. After approval, these are then published on the web site of the FIPA. The specifications are classified according to their status, which can be preliminary, experimental, standard, deprecated and obsolete. The standard specifications have been used in the creation of the 3APL platform. According to the FIPA an Agent Platform (AP) provides the physical infrastructure in which agents can be deployed. The AP consists of the machine(s), operating system, agent support software, FIPA agent management components and agents [28]. The agent management components are combined in the following scheme:

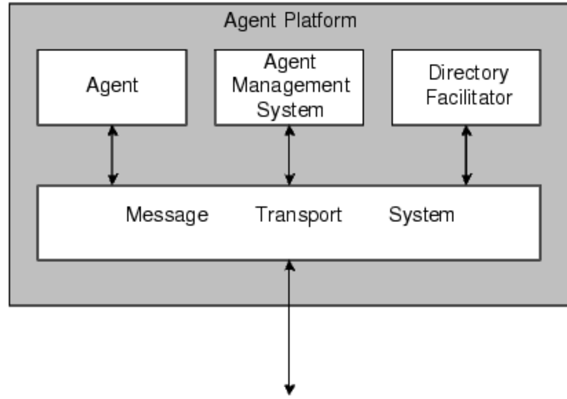

Figure 2: Agent Management Reference Model.

The agent management reference model consists of the following components, each representing a capability set:

- An **Agent** is a computational process that implements the autonomous, communicating functionality of an application. Agents communicate using an Agent Communication Lan-

guage. An Agent is the fundamental actor on an AP which combines one or more service capabilities, as published in a service description, into a unified and integrated execution model. An agent must have at least one owner, for example, based on organizational affiliation or human user ownership, and an agent must support at least one notion of identity. This notion of identity is the Agent Identifier (AID) that labels an agent so that it may be distinguished unambiguously within the Agent Universe.

- A **Directory Facilitator** (DF) is an optional component of the AP. The DF provides yellow pages services to other agents. Agents may register their services with the DF or query the DF to find out what services are offered by other agents. Multiple DFs may exist within an AP and may be federated.
- An **Agent Management System** (AMS) is a mandatory component of the AP. The AMS exerts supervisory control over access to and use of the AP. Only one AMS will exist in a single AP. The AMS maintains a directory of AIDs which contain transport addresses (amongst other things) for agents registered with the AP. The AMS offers white pages services to other agents. Each agent must register with an AMS in order to get a valid AID.
- A **Message Transport Service** (MTS) is the default communication method between agents on different APs.

The FIPA does not have standards on the internal design of an AP. For instance, inter-communication, i.e. communication between agents on the same platforms, may use any proprietary method. In the following sections we will describe the different components in more detail.

3.2.1 AMS

The AMS is responsible for managing the operation of an AP, such as the creation of agents, the deletion of agents and overseeing the migration of agents to and from the AP (if agent mobility is supported by the AP). Since different APs have different capabilities, the AMS can be queried to obtain a description of its AP. A life cycle is associated with each agent on the AP which is maintained by the AMS. The AMS represents the managing authority of an AP and if the AP spans multiple machines, then the AMS represents the authority across all machines. An AMS can request that an agent performs a specific management function, such as quit (terminate all execution on its AP) and has the authority to enforce the function if such a request is ignored. The AMS maintains an index of all the agents that are currently resident on an AP, which includes the AID of agents. Descriptions of agents, which are also maintained by the AMS, can be later modified at any time for any reason. Modification is restricted by authorisation of the AMS. Agents descriptions can be searched with the AMS.

3.2.2 Agent

From the FIPA perspective, an agent is a physical software process which exists on an AP and utilises facilities that are offered by the AP for realising their functionalities. An agent has a physical life cycle that has to be managed by the AP. A possible life cycle of a FIPA agent is shown on figure 3. This life cycle has the following properties:

- AP Bounded: An agent is physically managed within an AP and the life cycle of a static agent is therefore always bounded to a specific AP.
- Application Independent: The life cycle model is independent from any application system and it defines only the stated and the transitions of the agent service in its life cycle.
- Instance-Oriented: The agent described in the life cycle model is assumed to be an instance (an agent which has a unique name and is executed independently).

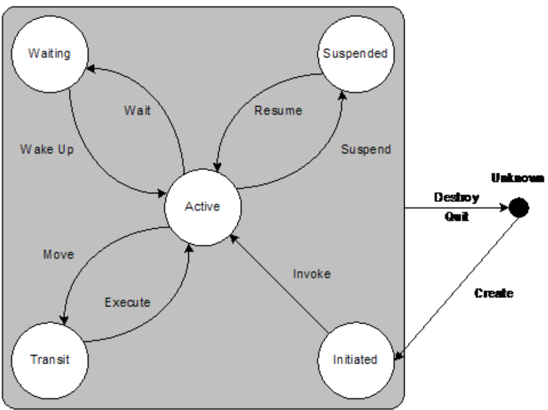

Figure 3: Agent Life Cycle

- Unique: Each agent has only one AP life cycle state at any time and within only one AP.

The delivery of messages is dependent on the state of an agent. The AMS, on behalf of the AP, has the responsibility to handle messages to an agent. This means that the AMS must control the MTS during the following states of an agent:

- Active: The MTS delivers messages as normal.
- Initiated/Waiting/Suspended: The MTS either buffers messages until the agent returns to the active state or forwards messages to a new location (if a forward is set for the agent).
- Transit: The MTS either buffers messages until the agent becomes active (that is, the move function failed on the original AP or the agents was successfully started on the destination AP) or forwards messages to a new location (if a forward is set for the agent).
- Unkown: The MTS either buffers messages or rejects them, depending upon the policy of the MTS and the transport requirements of the message.

The state transitions of agents can be described as:

- Create: The creation or installation of a new agent.
- Invoke: The invocation of a new agent.
- Destroy: The forceful termination of an agent. This can only be initiated by the AMS and cannot be ignored by the agent.
- Quit: The graceful termination of an agent. This can be ignored by the agent.
- Suspend: Puts an agent in a suspended state. This can be initiated by the agent or the AMS.
- Resume: Brings the agent from a suspended state. This can only be initiated by the AMS.
- Wait: Puts an agent in a waiting state. This can only be initiated by an agent.
- Wake Up: Brings the agent from a waiting state.

The following transistions are only used by mobile agents:

- Move: Puts the agent in a transitory state. This can only be initiated by the agent
- Execute: Brings the agent from a transitory state. This can only be initiated by the AMS.

3.2.3 Naming

Naming of an agent is defined by FIPA in order to uniquely identify an agent on a platform. Every agent resident on a platform has its own Agent Identifier (AID). An AID comprises of a name parameter, which is a globally unique identifier that can be used as a unique referring expression of the agent. An AID can be constructed by using the actual name of the agent and its home agent platform address (HAP), which is the address of the platform where the agent has been registered, separated by the '@' character. On the 3APL platform an agent can be uniquely identified by its name followed by the '@' character and the IP address of the platform, for instance *seller@192.168.0.10* .

3.2.4 Directory Facilitator

The directory facilitator (DF) provides yellow page services to the platform. It maintains a searchable list of services of agents which have been registered. It can be used to publicise services of an agent. Agents can search the DF to obtain another agent which has the appropriate services. DFs can also be federated which means that if a search has failed on the local DF, it is passed onto another DF of a different platform. The federation of DFs for extending searches can be achieved by DFs registering with each other using messages containing their description.

3.2.5 Message Transport System

The Message Transport System, also called Agent Communication Channel (ACC), is the software component controlling all the exchange of messages within the platform, including messages to/from remote platforms. It can be thought of as the single point of contact for all agents on a platform. The basic function of the ACC is to forward messages to the other agents, and to accept messages for forwarding from agents on it's platform and pass them on.

3.2.6 Communication language

In order to achieve greater inter-operability the FIPA has also specified its own standard agent communication language (ACL) [27]. A FIPA ACL message contains a set of one ore more message parameters. The following table lists all of the message parameters.

Parameter	Description
performative	Type of communicative acts
sender	Sender of the act
receiver	Receiver of the act
reply-to	Where the reply should be sent to
content	Content of the message
language	The language of the content
encoding	the encoding used for the content
ontology	the ontology for the content
protocol	A reference to a negotiation protocol
conversation-id	An identifier for the conversation
reply-with	An expression to identify the following reply
in-reply-to	An identifier for the reply (given in a previous $reply - with$)
reply-by	Denotes a deadline for the reply

Table 1: FIPA ACL message parameters.

User-defined message parameters other than the FIPA ACL message parameters are also allowed. The only mandatory parameter in all ACL messages is the $performative$. The $performative$ is based on speech act theory [22], wherein human utterances are viewed as actions in the sense of actions performed in the everyday physical world (e.g. picking up a block). FIPA has also specified

and identified 22 different performatives. In [29] the semantics of the performatives are listed. An example of the way in which performatives are defined is shown below.

Communicative Act:	inform
Summary:	The sender informs the receiver that a given proposition is true.
Meaning (Sender):	The contents of this message is a proposition that the sender believes is true. The sender can only send this message when it believes the proposition, wants the receiving agent to also believe this proposition and does not believe the receiving agent already believes the proposition.
Meaning(Receiver):	The receiver can now believe that the sender holds the proposition for true and it can also believe that the sender wishes that the receiver also believes the proposition. The receiver can decide for itself whether to believe the proposition or not.
Formal model: Feasibility preconditions: Rational Effects:	$< i, inform(j, \varphi) >$ $B_i\varphi \wedge \neg B_i(Bif_j\varphi \vee Uif_j\varphi)$ $B_j\varphi$

Table 2: Example of the FIPA Communicative Act Library Specification.

This example also shows the problems with the definitions of the performatives. All of the communicative acts are specified with preconditions, listed as 'Feasibility preconditions' and postconditions that are listed as 'Rational Effects' in the table. Note however that the FIPA does not give a formal definition of Rational Effects. The agent i who wants to send an $inform$ about a proposition φ to another agent j must belief the proposition itself and also belief that the receiving agent does not know φ. This precondition has been introduced to prevent 'flooding' of messages from an agent to other agents. The problem with this condition is that it is not very realistic to assume that an agent can know exactly what the other agents believe. Also the postcondition $B_j\varphi$ is ambiguous; it is not clear whether it should be a goal of the sending agent that the other agent also believes the proposition, or that it is a guarantee that the receiving agent believes φ. Although there are still shortcomings to the FIPA agent communication language, it is generally considered to be the best alternative [31]. The definitions of the performatives in a BDI model is easily translated in 3APL constructs. The additions that have been made to allow for this are explained in section 3.4.

3.3 Features of the 3APL platform

The specifications concerning an agent platform released by the FIPA have been used as a guideline for the development of the 3APL platform. This section discusses the implemented FIPA features of the specifications and the extend to which it complies with these features. The migration of agents to different platforms is not implemented and therefore has been omitted in the remainder of this section.

3.3.1 AMS

When the 3APL platform is launched the AMS is immediately created, and only one AMS instance is allowed per Java Virtual Machine. The AMS of the 3APL platform maintains a list of all agents known locally on the platform. It handles the lifecycle of the agents present on the platform. Because of the initial design goals of this platform, the user decides when an agent can start, stop or is to be removed from the platform. Therefore, all the user requests given by the graphical user interface (GUI) relating to the agents present on the platform are handled by the AMS. The

addition and deletion of an agent to the platform can only be done by selecting the appropriate action in the GUI. The user can also specify whether an agent should be in its suspended state. The AMS halts the deliberation cycle until the user gives the command to continue. Messages that have been sent to this agent by other agents that are still active are stored in the agent's messagebuffer. Once the deliberation continues the contents of the messagebuffer is pushed once every cycle.

3.3.2 Agent

An agent on our 3APL platform can also be viewed as a physical software process as in the FIPA perspective. It can also utilise the facilities offered by our agent platform for realising their functionalities, such as querying the AMS, sending messages to other 3APL platforms, etc.

A 3APL agent on our platform has a physical life cycle that is managed by the platform. The status of an agent is comparable to some of the FIPA example lifecycle states:

- READY. The agent has this status when the compilation of its 3APL source was successful and the deliberation cycle can be started. This status is comparable to the "Invoke" status of the FIPA reference lifecycle shown on figure 3.
- ERROR. The agent has this status when a compilation error of the agents corresponding 3APL source code occurs.
- FINISHED. The agent has this status when the deliberation cycle is finished.
- STOPPED. The agent has this status when the deliberation cycle has been halted. This status is comparable to the "Suspended" status of the FIPA reference lifecyle.
- RUNNING. The agent has this status when the deliberation cycle is executing. This status is comparable to the "Active" status of the FIPA reference lifecycle.

3.3.3 Naming

Naming of an agent is done by the programmer. Once the user has selected a 3APL source code, the AMS compiles this to not only supply the new agent with initial bases but also uses the "program" construct of the source code to supply the agent with a new name. The name is checked against the list of already locally known agents. If the name already exists on the platform the agent is not allowed to be hosted on the platform. This guarantees a unique name for the agent on the platform. Agents with the same username across different platforms is possible, because messages sent to other platforms must be supplied with the HAP address of the platform. Note that an agent created or loaded from 3APL source code can not register itself on the platform. This is done by the AMS.

3.3.4 Directory Facilitator

A directory facilitator has not yet been implemented in the 3APL platform. However, agents on the platform can use the AMS to simulate the functionality of a directory facilitator. If an agent wants to submit its capabilities, it can use a send action addressed to the AMS, with an *inform* performative, and the content of the send goal is a formula, *description* with a parameter, which is the service that is to be announced. Deregistration can be achieved by appending *NOT* before this formula content. For instance, an agent *Seller*, that wishes to announce to the AMS that it can sell products will have a goal in its rulebase such as:

```
Send(0, ams, inform, description(seller) );
```

An agent that wants to buy products can use a rule such as:

```
Send(1, ams, query, description(seller) );
```

If the AMS find agents in its agent list with the reported description, it returns a message with in its content a list with the names of the agent, in this case *Seller*. The sender of the query will have in its beliefbase the following:

```
received(1, ams, reply, name([Seller]) );
```

Note that if the AMS did not find an agent with the queried description, the returned list can be an empty list.

The agent can use practical reasoning rules to handle the content of the received message, for instance:

```
Buy() <- received(X, ams, reply, name(S) ) |
```

When the agent *Seller* has no more products to sell, it can notify the AMS it is no longer a seller by using:

```
Send(0, ams, inform, NOT description(seller) );
```

3.3.5 Message Transport System / Agent Communication Channel

The agent communication channel is handled by the AMS. Every 3APL agent has its own private messagebuffer, which is implemented as a queue. At the beginning of a deliberation cycle, this buffer is checked for contents and when it is not empty, the contents of the buffer is put into the beliefbase of the agent. If an agent sends a message, i.e. a *Send* goal is selected in the deliberation cycle, a method call is made to the AMS. The AMS handles the message by looking at the receiver field. If the name of the receiver is supplied with a HAP address, the AMS forwards the message to the AMS of the platform corresponding with this IP. If only the name of the agent is supplied in the receiver field, it is assumed that the agent is local. The list of known local agent is searched and if it is found, the message is added to the corresponding agent's messagebuffer. The reason for only allowing the AMS to actually transmit and receive messages to other platforms is based on performance. Only one instance of the Java Agent Services (JAS) library is used per platform. The JAS project is an initiative to define an industry standard specification and API for the development of network agent and service architectures. The project has also been submitted to the Java Community Process (JCP) which is a open organisation of international Java developers and licensees which controls the extensions made to the Java language. The aim of the project is to be included in the javax.agent namespace [23]. At the start of the application the user must select whether this instance of the program on a specific machine is to be server or client. If the user decides that the application is to be a server every instance of the application on other machines that wishes to contact the aforementioned server must select to be a client and supply the IP address of the server.

3.3.6 Communication Language

The handling of FIPA performatives is supported by the 3APL language by means of practical reasoning rules.

3APL is extended in order to facilitate the communication between 3APL agents. Therefore, the possible goals from definition 2.3.1 have been extended with the following goal: [5]

Definition 13 *SendGoal: if $\varphi \in$ BF, then Send(Number, Receiver, Performative, φ) $\in$ GOAL*
Number, Receiver and Performative are all parameters, to be filled by the user. Beliefs φ can be either a well-formed-formula or a variable parameter.

We also extend the definition of an agent state in 3APL by including a messagebase in the state.

Definition 14 *The messagebase Ω of an agent is a set consisting of sent and received messages having the form Sent($\tau;\alpha;\beta;\rho;\psi$) and Received($\tau;\alpha;\beta;\rho;\psi$), where τ is a term that denotes the message identifier, α and β are terms denoting the identifiers for sender and receiver of the message, ρ denotes a FIPA performative, and ψ is the content information. The state of a 3APL agent can now be defined as a 5-tuple $<id;\Pi;\sigma;\theta;\Omega>$, where id is the agent's identifier, Π is the set of agent's goals, σ is agent's beliefs, θ is a substitution consisting of binding of first order variables that denote domain elements, and Ω is the messagebase.*

The messagebase allows for the synchronous transportation of messages, because messages are delivered immediately to the corresponding agent(s) and stored in their respective messagebase(s). Messages that have been sent are also stored in the messagebase of the sending agent. This allows for construction of goals that are dependent of sent messages. Having a buffer also allows asynchronous handling of the received messages. At the beginning of the deliberation cycle the messagebase is checked for contents. If a message is found, the contents is pushed on the beliefbase. In order to not introduce new transition rules based on the message buffer, which could alter the syntax of 3APL, it has been decided that the interpretation of the messages is done by practical reasoning rules, which have guards based on formula in the beliefbase. The definition of practical reasoning rules therefor does not need to be altered. Messages, both sent and received, are now represented as beliefs, so they can be handled as usual. The semantics of Send and Receive is specified in terms of transition rules, we refer to [5] for detailed description of these rules. Please note that in this document the transition rules are specified with respect to the messagebase instead of the beliefbase. Because of the aforementioned reason, this is no longer the case as messages are handled as beliefs in the beliefbase.

3.4 Example

The following example shows the possibilities of the 3APL platform. It is a continuation of the example of the previous chapter. Once again, we simulate a situation called a combination shot, which is common in soccer. Two players are on a grid, with the goal and a ball. The mid-fielder asks the AMS for the presence of a center. The AMS replies with the name of the center. The midfielder agent then asks the center agent for its position on the grid. Once the midfielder has received an answer, it goes on to fetch the ball and then passes it to the center-forward, who then proceeds to make a goal (never misses). Finally the center-forward informs the midfielder that it indeed has scored. Here we will only describe the source code of the 'midfielder'. The source code for the 'center' is included in appendix A.

```
PROGRAM "midfielder"

CAPABILITIES:
 {player(X0,Y0) AND ball(X1,Y1)}
     MoveToBall()
 {NOT player(X0,Y0), player(X1,Y1)}

 {ball(X0,Y0)}
     KickTo(X1,Y1)
 {NOT ball(X0,Y0), ball(X1, Y1)}

 {received(V,ams,reply,name(B))}
     CenterAvailable()
 {NOT received(A,ams,reply,B), center(B)}

 {received(V,A,S,B)}
     PositionOfCenter(X,Y)
 {NOT received(V,A,S,B),positionOfCenter(X,Y)}

 {received(A,B,inform,haveWon)}
    HaveWon()
 {NOT received(A,B,inform,haveWon), won()}

BELIEFBASE:
 player(100,100),
 ball(50,80),
```

```
 goal(0,50)

GOALBASE:
 Init(),
 GetPositionOfCenter(),
 MakePass(),
 WaitForVictory(),
 Close()

RULEBASE:
 Init() <- player(X,Y) AND ball(BX,BY)|
           BEGIN
           Send(0,ams,inform,description(isMidfield));
           Send(1,ams,query,description(isCenter))
           END
,
 GetPositionOfCenter() <- received(V,ams,reply,name(B)) |
           BEGIN
           Send(0,B,query,position())
           END
,
 MakePass() <- player(PX,PY) AND received(V,A,reply,position(X,Y)) |
             BEGIN
             MoveToBall();
             KickTo(X,Y);
             Send(0,A,inform,shoot())
             END
,
 WaitForVictory()  <- received(A,B,inform,haveWon) |
          BEGIN
          HaveWon()
          END
,
 Close() <- won() |
            BEGIN
            Send(0,ams,inform,NOT description(isMidfield))
            END
.
```

Listing 2. The source code for the midfielder agent.

The conversation is shown in the following figure, taken from the output of the “sniffer”, which is a graphical conversation logger tool supplied with the platform:

The deliberation of the *midfield* agent starts with the *Init*() goal. This goal is used to construct two send goals. The first send goal informs the AMS of the platform that the agent is a midfielder. The second send goal queries the AMS for the presence of a center agent. The rule *GetPositionOfCenter* has a guard $received(V, ams, reply, name(B))$. The B parameter is unified with the name of the agent that has informed the AMS that it is a center, which is *Center* in this example. This goal creates another send goal, this time addressed to the now known center-forward player, asking it for its position on the grid. The goal *MakePass*() can be achieved if the *center* replies to the *midfielder* with a message containing its position. The basic action *MoveToBall*() is used to move the midfielder to the position of the ball. The position of the center-forward (X, Y) in the beliefbase is used as a parameter for the basic action *KickTo*(). This

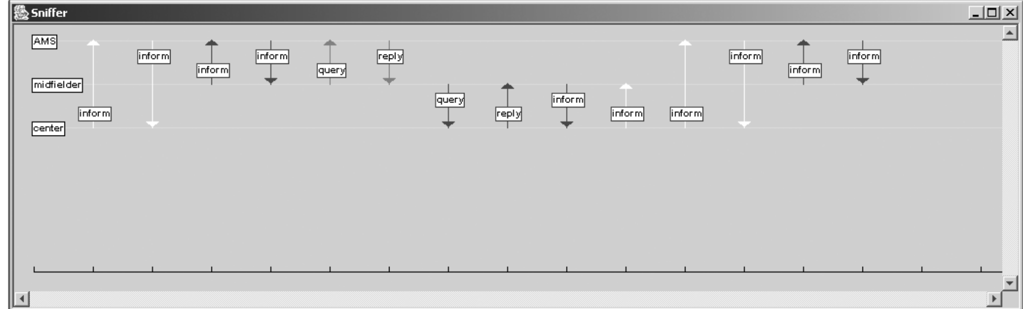

Figure 4: The conversation log of the two agents.

action updates the beliefbase with the new position of the ball, which is now placed at the coordinates of the center-forward. The last action is another send goal, with the content $shoot()$ to inform the center-forward that it should shoot the ball towards the goal. Once again a new goal is created, namely $WaitForVictory()$. This goal has the guard $received(A, B, inform, haveWon)$ which can be fulfilled if the center-forward sends this content with an "inform" performative to the midfielder. If the beliefbase is updated with the $haveWon()$ item, the final goal $Close()$ can be triggered. This goal is used to inform the AMS that the agent no longer is a midfielder. If this goal has been fired, there are no more goals to be fulfilled and the deliberation of this agent is stopped.

4 Manual and implementation

4.1 Software Requirements

The only requirement for running the 3APL platform is a recent Java installation. The platform is developed using Java 2 SDK 1.4 so this version is also recommended. It has also been tested on version 1.3 with good results. The platform comprises of several libraries: the original 3APL library, the JAS library [23] and code freely released by Slava Pestov which is used for the syntax highlighting editor. In the next section we will describe the visual aspect of the platform.

4.2 Main

When the 3APL platform is started, the user is presented with the following window:

Figure 5: The startup dialog

The user must select whether the application should act as a server or as a client. If the user selects server, the application will start a new instance of the name server, which is a part of the JAS library. If the user selects client, the IP of the server with which the platform should connect must be filled in. The AMS of the client platform is then added to the list maintained by the server.

Once the choice has been made the application continues and the user is presented with the main window, shown below:

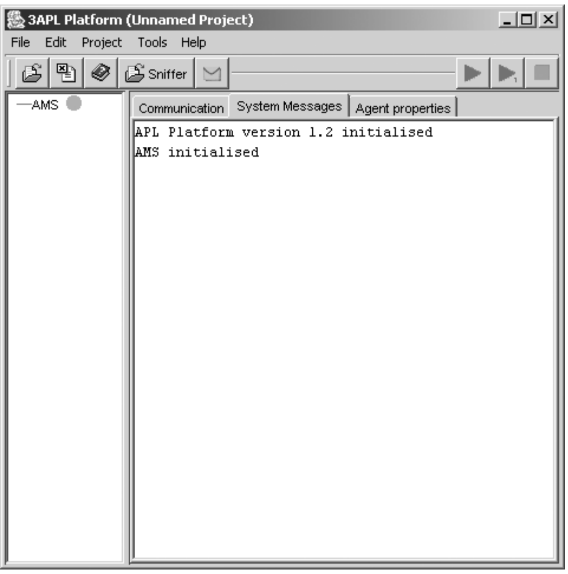

Figure 6: The main window

Left of the main window is a visual tree which represents all of the agents present on this platform. The AMS is also listed, but it is not a real agent in the sense that it can not be controlled or inspected by the user. The leafs of the tree show the names of the agents that have been added to the platform. On the righthand side of the name of an agent is an icon displaying the status of this agent. The status of an agent can be any of the following:

Icon	*Meaning*
	`Source code compiled successfully and agent is ready.`
	`Agent's deliberation cycle has started.`
	`Agent's deliberation cycle has been halted.`
	`Agent's deliberation cycle has ended.`
	`An error has occurred.`

Table 3 Status icons of agents.

An error can be given whenever the 3APL source code has been altered and a compile error occurred. The System Messages panel will display the error message. This panel is generally used to display information about the platform.

The leafs of the agents are the descriptions of the agents as they are reported to the AMS. This can be done by sending an inform to the AMS, for instance: `Send(0, ams, inform, description(seller) );`

4.2.1 Messages window

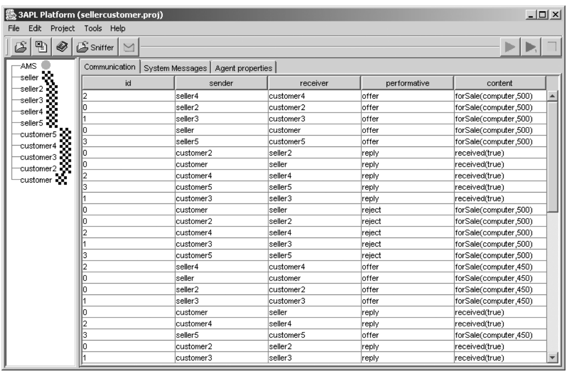

Figure 7: The communication window of the application.

If an agent sends a message to another agent or the AMS the message will be displayed in another tabbed panel, the Communication panel. If the message is addressed to an agent which is also located on the platform no HAP is displayed. The HAP is shown on the Communication panel whenever an agent receives a message from an agent on another platform, e.g. *seller*@10.0.0.12

4.2.2 Agent properties

If the user selects an agent, the tabbed panel "Agent properties" is shown. It shows the current beliefbase, goalbase, capabilitiesbase and rulebase of the agent. If the agent is running the inference log is filled with the deliberation steps taken during the cycles.

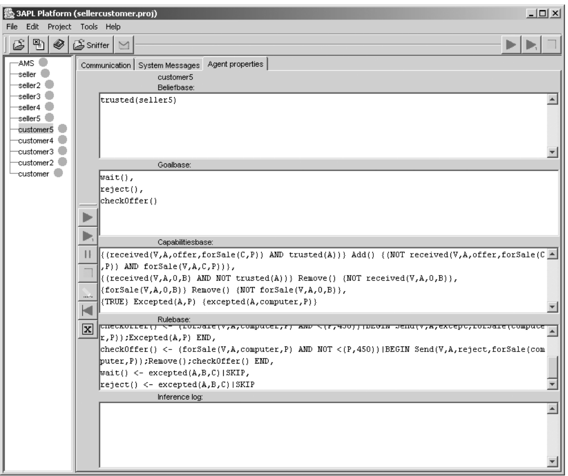

Figure 8: The Agent properties panel.

On the left side of this panel are the control buttons for the selected agent situated on a toolbar. They have the following functionality:

Icon	*Meaning*
	`Begin the execution of the selected agent.`
	`Halt the deliberation cycle.`
	`Stop the execution of this agent.`
	`Edit the 3APL source code.`
	`Recompile the source code.`
	`Remove this agent from the platform.`

Table 4 Agent status icons.

If the user selects the first button the deliberation cycle of the agent commences. This cycle is continued until either the user indicates the agent to stop or the agent has no more goals to fulfill, i.e. the goalbase of the agent is empty. Note that the agent is non-reactive. The second button causes the agent to only make one cycle. It then goes to its paused state. The user can select the third button to halt the deliberation cycle at any time during the deliberation cycle. If the user presses the stop button, the deliberation cycle is stopped. Editing of the source code can be done by selecting the fifth button. A syntax highlighting text editor is opened displaying the contents of the agent's source code. The user can now alter the source code of the agent, and the changes can be applied, which means a recompile of the 3APL code, or the changes can be ignored, in which case nothing happens. If the user wants to reset the configuration of the agent, i.e. its initial bases are set to the initial bases from the source code, the user can select the recompile button.

4.3 Sniffer

One of the tools which can be activated is the Sniffer tool. It can be executed by clicking the appropriate icon located on top of the main window. This tool produces an output like figure 9

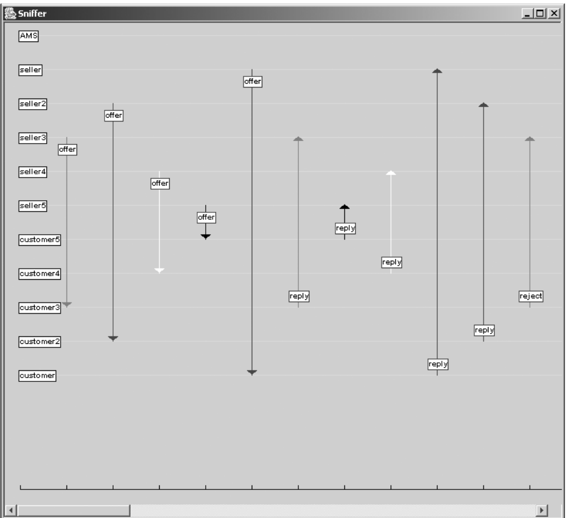

Figure 9: An example output of the Sniffer tool.

This tool is used to visualise the conversations between the agent on the platform. When the sniffer has been started, all of the local agents and the AMS are shown on the left side of the window. Every message produces an arrow, with its origin starting at the position of the sender, and it head ending at the receiver of the message. The color of the arrow depends on the message identifier used with the message. On every arrow also the performative is displayed. If a user double clicks this performative, an additional message box is opened, with an output which can be as follows:

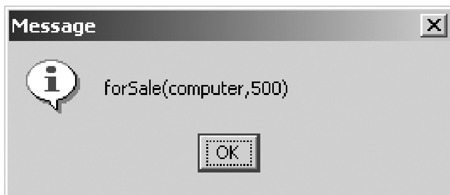

Figure 10: An example output of the message box.

This message box displays the contents of the message. We have chosen for this forms, to avoid clutter on the sniffer output window.

4.4 Message sender

Another tool available to the user of the platform is the custom message sender. This tool can be used to test if received messages are handled correctly by an agent located on the platform.

All of the message parameters must be filled in by the user, which includes the message identifier, the sender and intended receiver which can be selected from a list of agents currently on the platform, the performative which does not have to be a FIPA compliant one and the content of the message, usually a belief formula. If the user clicks the send button a confirmation dialog will be displayed if the sending of the message has succeeded. The user can then send other messages or use the cancel button to close this tool. It is important to note that the sending of a message implies the adding of the newly formed Send goal to the message buffer of the receiving agent. If the agent's deliberation cycle has started the message buffer is checked for contents. When this

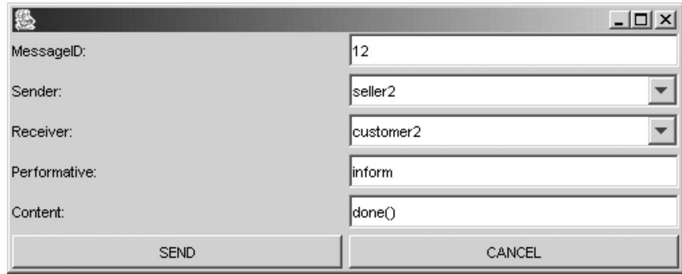

Figure 11: The message sender tool with example data entered.

buffer is not empty, the first delivered message is pushed on to the beliefbase. This process is continued until there are no more goals to be fulfilled by this agent. Whether a message sent to the agent will actually be handled depends on several factors. The most important one is if the agent is active, i.e. it is executing its deliberation cycle. If it is not, then the message is stored in its message buffer until the agent becomes active (again). Whether the message eventually gets pushed on the beliefbase depends on the rulebase or basic actions which handle the message.

4.5 Templated agents

When the user uses the "add agent.." in the project menu, a window is shown with the following visualisation:

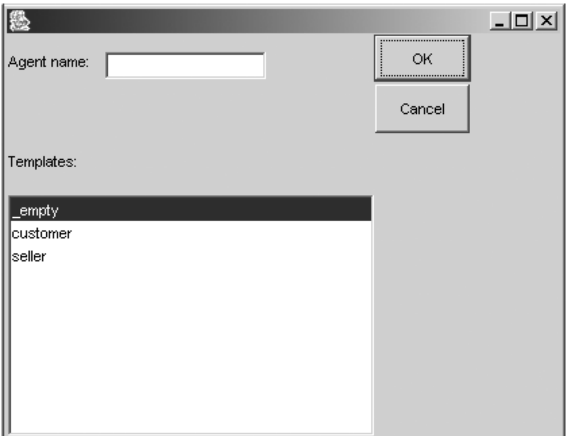

Figure 12: The "add new agent" dialog.

The list shown in the templates window is taken from the *Template* directory during startup of the application. The templates are 3APL source codes which represent agents with some common abilities, for example seller or customer. If the user has given a name for the new agent and has selected a template the agent with the chosen name will be added to the platform. The source code of this new agent is a copy of the selected template source code with the exception of the *program* sentence. This has been filled with the name of the agent the user has given in the agent name field. Users can add their own templates. In order for this to work, the template source code must be placed in the *Template* directory. A copy of the template will be created. The name of the new agent is taken from the Agent name field as shown above in figure 12. This name will replace every occurrence of "%1" in the source code. This means that a template must have a first sentence such as: *program* "%1".

4.6 Implementation

In this section we will describe the implementation of the two most important classes, *Agent* and *AMS*. But first a general description of the architecture of the platform is given. We use the following conventions in describing the source code:

- Variable names are in italic, for example: *properties* ;
- Method names are also in italic and followed by (), for example: *addToBeliefBase*() ;
- Class names which are mentioned for the first time and are defined in another package than *aplp* are given with their namespace, for example: *java.util.Observable*. If the class is mentioned further on than only the classname is given, for example: *Observable*.

4.6.1 General architecture

The following figure provides a general overview of the platform.

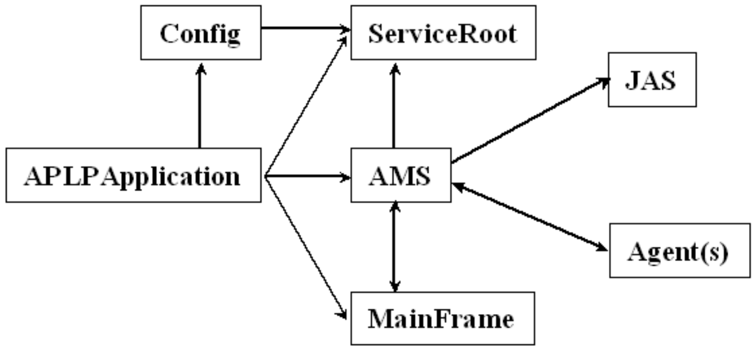

Figure 13: Overview of the 3APL platform.

The *APLPApplication* is used as a starting point for the application. This class contains the *main* method for the platform, the starting point for a Java program. The constructor of this class first creates a *Config* object. This object is used to adjust the configuration of the platform by selecting the appropriate action in the user interface, as shown in figure 13. The *APLPApplication* continues with the creation of a *ServiceRoot* object. This object is used to initialise the JAS environment and it uses the data entered in the *Config* object to alter the parameters of the components. The AMS object is then created, with a pointer of the ServiceRoot as a parameter of its constructor. Finally the graphical user interface handler, *MainFrame*, is called.

4.6.2 Agent class

The class *Agent* is the main class used for the representation of an agent of the platform. It is based on the original *Agent* class found in the single agent implementation [25]. The class is implemented as a "data" class in the model-view paradigm [7]. Therefor, it extends the *java.util.Observable* class. This is a utility class enabling the observation of an object. In the *Agent* class this is used to notify other objects such as the *AMS*, i.e. the "view" components, that data has been changed. The *Agent* class also implements the *aplp.AgentTreeItem* which is an interface used to render the agents in the tree in the user interface. The most important attributes of the *Agent* class are:

- *BeliefBase*
- *CapBase*

- *RuleBase*
- *MessageBase*
- *AMS*
- *SwingWorker*
- *Vector properties*
- *status*

These are all declared *private* in order to comply with the notion of autonomy of agents. The *BeliefBase* class is the implementation for the beliefbase. Beliefs are encoded as Prolog statements using the JIProlog prolog interpreter [14]. For a more complete description of the first three classes, we refer to [13], as they have not been modified in this project. During the design stage, the decision was made to have the simplest method of handling incoming messages. The messages are handled in a FIFO policy, i.e. the messages that arrive first are the first to be handled. As a result, the *MessageBase* is implemented as a queue. It implements the methods *enqueue* and *dequeue* for enqueuing resp. dequeuing *SendGoal* objects. Because the inference cycle of an agent can be a time consuming task, it is executed as a thread. A thread enables the execution of code concurrently. A *SwingWorker* attribute is used to handle the inferencing thread. It is an instance of the *javax.swing.SwingUtilities.SwingWorker* class [30]. It is essentially a wrapper for the *java.lang.Thread* object, with additional code for easier handling of interrupts. The attribute *AMS* is a reference to the *aplp.AMS* class, which is initialised in the constructor of the agent class. The attribute *properties* is an instance of a *java.util.Vector* object, a utility class used for storing a set of objects where the number of objects is unknown at runtime. This attribute contains *aplp.AgentProperties* objects which are created when the agent sends an inform to the AMS to register its description. The content of this message also sets the name of the *AgentProperties* object. This class is used to update the graphical tree which lists the agent as roots and its properties as leaves. The *status* attribute is used to encode the status of an agent as an integer. This attribute can be one of the following values:

- READY (0). The agent has this status when the compilation of its 3APL source was successful and the deliberation cycle can be started. This status is comparable to the "Invoke" status of the FIPA reference lifecycle shown on figure 3.
- ERROR (1). The agent has this status when a compilation error of the agents corresponding 3APL source code occurs.
- FINISHED (2). The agent has this status when the deliberation cycle is finished.
- STOPPED (3). The agent has this status when the deliberation cycle has been halted. This status is comparable to the "Suspended" status of the FIPA reference lifecyle.
- RUNNING (4). The agent has this status when the deliberation cycle is executing. This status is comparable to the "Active" status of the FIPA reference lifecycle.

The most important methods of the *Agent* class are:

- *begin*()
- *work*()
- *addToMessageBase*()
- *addToBeliefBase*()
- *sendAgent*()

When the user indicates to start the agent, the AMS calls the method *begin*() of the agent. This method starts the instance of the *SwingWorker*, that starts a thread which runs the *work*() method of the agent concurrently. The *work*() method implements the deliberation cycle. It starts with the cleaning of several strings used in the user interface for the visualisation of the bases and the *InferenceLog* which is a string containing the information about the steps taken in the deliberation cycle. The method continues with a *while* loop which checks if the goalbase is not empty, or if there are no more rules applicable. If it is, the deliberation can be terminated; there are no more goals to be accomplished or rules applicable. Then the messagebase is checked for contents. The messagebase can only be filled by the AMS when a message has been sent to the receiving agent with the method *addToMessageBase*(). If the messagebase contains a SendGoal object, then the object is pushed from the queue and placed in the beliefbase of the agent with the method *addToBeliefBase*(*Object o*) once every cycle. The contents of the message can now be used in the deliberation cycle. The *while* loop continues with an *if* statement which checks if the AMS has not requested that only a single deliberation step should be taken. The following code shows the most important part of the deliberation cycle. Parts of the code that handle user interface updates have been omitted, as well as code needed for the handling of a *SwingWorker* object.

```
Vector result1 = new Vector();
rb.findRulesMatchingGoals(gb,bb,result1);
Vector result2 = new  Vector();\\
rb.findRulesMatchingBeliefs(bb, result1, result2);\\
RuleOption ruleoption;
 if (result2.size() > 0) ruleoption =(RuleOption)result2.firstElement();
  else ruleoption = null;
gb.fireRuleOnGoal(ruleoption); Vector result3 = new Vector();
gb.optionalGoalsToExecute(cb, bb, result3);
GoalOption goaloption;
 if (result3.size() > 0)  goaloption =(GoalOption)result3.firstElement();
   else  goaloption = null;
 if (goaloption != null)  {
   if (goaloption.getGoal().getClass().getName().equals("ics.TripleApl.SendGoal"))
    {  sendAgent(goaloption.getGoal());
       gb.executeGoal(null, null, bb, goaloption, cb); }
```

In line 2 the rulebase (rb) is searched for rules with a head that matches a goal in the goalbase (gb) with the methodcall *findRulesMatchingGoals*() . The beliefbase (bb) is also used in this matching to find substitutions of variables. The resulting set of possible rules is stored in a vector *result*1. The method call in line 4, *findRulesMatchingBeliefs*() is used to find substitutions for the guards in the selected rules. If guard substitutions can be found on the rules, these rules are placed in a new vector, *result*2. If the set of matching rules is not empty, the first rule is selected from *result*2, in lines 5 to 9. The rule is cast to a *ics.TripleApl.RuleOption* and the method call *fireRuleOnGoal*() is used from the goalbase (gb) in line 10. After the selection of a rule to be 'fired' the goalbase is searched for possible goals to be executed in line 12. If this results in a possibility the first one is selected in line 15. The *goaloption* is further examined in lines 18 to 23. If the selected goal is an instance of *ics.TripleApl.SendGoal* the method *sendAgent* is called. This method sends the *SendGoal* to the local AMS which handles the actual sending of the message to the message buffer of the corresponding agent and also handles the administration of this message on the graphical user-interface. The method call *executeGoal* of the goalbase in line 24 is finally used to execute the goal.

4.6.3 AMS

The *aplp.AMS* class contains the implementation for the FIPA AMS. It also contains the implementation of the agent communication channel (ACC) because of the tight bonding between these

two services. For an example of this bonding see section 3.1.2. The most important attributes of the AMS class are:

- *MainFrame main*
- *Vector agentList*
- *String* name
- *ServiceRoot sr*

The *main* object is a pointer to the *MainFrame* object, which is used to handle the graphical user interface. This pointer is used for example whenever a message is handled by the AMS, then the communication panel should be updated appropriately, which is an object handled by the *MainFrame*.

The *MainFrame* object also has a pointer to the AMS. This pointer is used by the *MainFrame* object to notify the AMS that for example the user wants to add an agent.

The *Vector agentList* is an instance of the aforementioned *java.util.Vector* object and it is used to store and handle the *Agent* objects.

The *String* name object is used to identify the AMS of the platform to the nameserver instance of the JAS library. The name of the AMS with which the AMS registers on the JAS name server is ams@HAP, where HAP is the IP of the machine that this application is running on, for example ams@192.168.0.2 . This is in accordance with FIPA specifications for an AMS.

The *sr* attribute is an instance of the aforementioned *ServiceRoot* class of the JAS library. It is used to start the JAS nameserver and the JAS transport system.

Because the FIPA specifications state that only one AMS is allowed on a platform, the *AMS* class is implemented as a singleton instance [7]. This ensures that only one AMS object can be created on the platform. When the application is started, it starts the only instance of the *AMS*. The *AMS* extends the *Agent* class because it uses several properties which are already captured in the *Agent* class, such as the status. The status of the AMS is currently defined to always be in the "READY" state. The other possible states are currently not used for the AMS. Also the goalbase, beliefbase, capabilitiesbase and rulebase are inherited from the Agent class. These are used differently than in the actual *Agent* class, they are used as a placeholder for newly created agents. The AMS also implements the *MessageListener* interface which is used to get an automatic notification if a message has arrived from another AMS.

The methods of the AMS which we will describe here are:

- *addAgent*()
- *handleMessage*()
- *handleAMSMessage*()
- *sendMessage*()
- *receiveMessage*()

When the method *addAgent*() of the AMS is called, the *file* parameter of this method is used to call the *ics.TripleApl.TAPLCompiler.compile*() method. This method is used to compile the 3APL source code. If no compiler errors have occurred this method returns with a name for the new agent, together with an initial beliefbase, goalbase, rulebase and capabilitiesbase. The *AMS* bases (goal-, belief-, capabilities- and rulebase) are used as a placeholder for these objects. A new *Agent* object is created which uses parameters to fill its initial bases. Pointers to the *Agent* objects are placed in the aforementioned *agentList* vector. This vector is used heavily throughout the source of the AMS. It is used for instance whenever a message is sent to scan for agent names. It is also used to handle the lifecycle of the agents. If the user selects the agent from the tree in the user interface, the reference of the *Agent* is returned to the *AMS* by the *MainFrame* class.

Once removed from the *AgentList*, an agent is no longer referenced, and therefor it can be deleted during the next garbage collection of the virtual machine.

The *AMS* class also implements the ACC. If a *SendGoal* has been selected during the deliberation cycle of an agent, the agent calls the *AMS* method *handleMessage*() with the sendgoal as a parameter. The AMS uses the *SendGoal.getRecipient*() method to determine to whom the message is addressed. This method returns a *java.lang.String* containing the receiver. If the recipient is the AMS itself, the method *handleAMSMessage*() is used to handle this message.

If the recipient string does not contain an '@' symbol, then it is assumed the message is intended for an agent hosted on the same platform. The recipient is searched in the *agentList* vector and when found, the method *addToMessageBase* from the corresponding agent is used to add the message to the message base of the agent. If the recipient agent is not found, the message is discarded.

If the recipient string contains an '@' symbol, then it is assumed the message is intended for an agent located on an external platform. The external communication is handled by the Java Agent Services (JAS) library [23]. The *ServiceRoot* object passed on by the *aplpApplication* class to the *AMS* is an interface used by the JAS to start the local system services, i.e. a transport system and an agent naming service. The agent naming service contains the addresses for other 3APL AMSes which have notified their presence. The IP address located behind the '@' symbol is used to construct the AMS name of the external platform. By definition, this name is ams@HAP, where HAP is the IP address of the computer the platform is hosted on. The method *sendMessage* is used to send the external message. The method call *receiveMessage* of the AMS is automatically called by the JAS library if a message has arrived. The message object is parsed and the recipient agent of the platform is searched locally and if found the handling of the message is identical to the *handleMessage*() method except that the sending agent name has an appended string @HAP, where HAP is the IP address of the computer the external agent is hosted on.

The transport system currently uses the RMI protocol for the actual sending of the messages to other platforms [21]. This protocol is included with the Java distribution. The JAS library provides mechanisms for implementing other protocols, such as HTTP.

5 Comparison

There are several platforms which allow the development and deployment of multiagent systems. We will compare three existing platforms with the 3APL platform. They are: JADE, ZEUS and JACK. These have been selected because like the 3APL platform, these platforms are more or less modelled according to FIPA specifications and are implemented using the Java language. Also these platforms are popular and regularly maintained. We will compare the platforms by looking at what facilities are provided in the four stages of traditional software design, which are: analysis and specification, design, implementation and deployment. The platforms will be described using the following criteria:

- *Purpose* What is the aim with which the platform has been developed ?
- *Analysis and specification* How are multiagent applications analysed and specified on this platform. Which facilities are provided for this phase?
- *Design* How can agents be designed using this platform ? What facilities are provided with the platform to help design multiagent systems?
- *Implementation* How are agents implemented on the platform? What facilities are provided to help with the implementation of multiagent systems or agents?
- *Deployment* If, and how can an agent be executed on the platform? What facilities are provided to help debug or give insight into the internals of multiagent systems or agents?
- *Communication* How is communication handled, not only the transportation protocols used but also the way in which an agent can initiate communication.

To give an indication of the implementation of agents on these platforms we provide the source codes for a very simple agent, a 'PingAgent' for all platforms, including the 3APL platform. This simple agent waits until it receives a message, and if it has received a message, the agent will reply the sending agent with a message indicating successful reception. The source codes can be found at appendix B.

5.1 JADE

JADE (Java Agent DEvelopment Framework) is a software framework implemented in Java. It tries to achieve the simplification of the implementation of multiagent systems through middleware that claims to comply with the FIPA specifications and through a set of tools that supports the debugging and deployment phase [26]. JADE has been developed jointly by CSELT and the Computer Engineering Group of the University of Parma. The platform is available under Open Source License.

Purpose
The aim of JADE is to provide a software development framework aimed at developing multiagent systems and applications conforming to FIPA standards for intelligent agents.

Analysis and specification
JADE does not provide facilities to aid the analysis of multiagent systems.

Design
JADE agents are designed by defining what functionality an agent should have or service it should provide. These tasks, called 'behaviours', are then used to construct agents. The computational model of a JADE agent is multitask, where tasks (or behaviours) are executed concurrently. Each functionality/service provided by an agent should be implemented as one or more behaviours.

No evidence of facilities to help in the design of agents or multiagent systems have been found.

Implementation
Implementation of a JADE agent is done by providing Java source code. Specifically, the user has to implement a Java class that extends the base *jade.core.Agent* class. Using this construct, the user-defined class inherits features to accomplish basic interactions with the agent platform (registration, configuration,remote management, etc.) and also a basic set of methods that can be called to implement the custom behaviour of the agent (e.g. send/receive messages, use standard interaction protocols, register with several domains, etc.). A user-defined agent is then constructed by providing Java source code which must be added to the appropriate behaviours, which are then added to this derived Agent class.
The facilities provided by JADE to be used during the implementation stage are Java-based libraries, called the software development framework, to enable construction of JADE agents.

Deployment
Execution of a JADE agent is done by first compiling the Java source code into bytecode, resulting in a class file, which can be executed by a Java Virtual Machine (JVM), and can be added to a running JADE platform by either using the GUI or from the command line by providing the appropriate class file. JADE offers several facilities to help deploy multiagent systems. They are:

- a run-time environment for hosting agents;
- FIPA components: Agent Management System (AMS), Directory Facilitator (DF) and a Message Transport System (MTS).

  JADE also provides several tools to help in the administration of a multiagent system, which are:

  - a remote monitoring agent for the handling of the lifecycle of agents, which can be hosted on another JADE platform;
  - a sniffer agent tool to graphically view the communication between agents, similar to the 3APL sniffer;
  - a dummy agent tool to allow interactions with JADE agents in a custom way;
  - GUI to edit the supplied DF.

Communication
Communication is done by adding appropriate behaviour classes to the agent source code. Sending and receiving messages is done by using the *SenderBehaviour* and *ReceiverBehaviour* respectively. Messages are constructed by using the provided *ACL* message class, which is constructed by filling in the parameters such as the performative, message identifier, recipient etc. The transportation protocol used is CORBA ORB but several others can be used if these are compiled and linked with the platform. Messages are conform to the FIPA 2000 ACL specifications.

5.2 ZEUS

ZEUS is an integrated environment for the rapid building of collaborative agent applications. It has been developed by the Agent Research Programme of the British Telecom Intelligent System Research laboratory. ZEUS is open source software, released under the Mozilla Public Licence.

Purpose
The aim of ZEUS is to facilitate the rapid development of new multiagent applications by abstracting into a toolkit the common principles and components underlying some existing multiagent systems [15].

Analysis and specification
Designing a multiagent application is done by first defining the problem as role modelling of agents. This modelling can be done by using UML class diagrams and patterns. The design of agents then involves the mapping of this UML model to the ZEUS agent model.

Design
The design stage consists of finding solutions for the different role responsibilities identified during analysis. Agents are constructed by using a graphical environment in which the requirements for a ZEUS agent can be filled in, such as the used ontology, the specific role of this agent in an hierarchy, the definition of the problems the agent should tackle, etc. Graphical tools are provided with ZEUS to aid in the design of agents, which are:

- Ontology Editor
- Agent Definition Editor
- Task Description Editor
- Organisation Editor
- Co-ordination Editor

The underlying ZEUS agent model limits the design to task-oriented, goal-driven, collaborative agents [20].

Implementation
ZEUS provides Java based libraries to aid in the construction of agents. A code generator tool is also supplied with ZEUS. This tool creates Java source codes for every agent in the application. The created source code for an agent should not be edited. The code generator also generates skeleton source codes for every task which the user should only edit if there are external activities (e.g. external data from a different program). The generated Java codes by the code generator does not include programmatical links to external resources or external programs. These links should be added to the generated Java code by hand.

Deployment
Execution of an ZEUS multiagent system is done by compiling all of the Java source code to bytecode, so that it can be executed by a Java Virtual Machine. The code generator tool can be used for this task. This tool also generates scripts which can be used to launch the multiagent system.

ZEUS also provides assisting agents. These are a Name Server Agent, which takes care of the agent name to network location resolution, a Service Discovery Facilitator Agent and persistent storage through the use of a Database Proxy Agent.

The first script that is generated by the code generator tool will launch the Name Server Agent. The second script will launch all of the compiled user defined agents. Finally the third script will launch optional utility agents such as the Facilitator Agents, visualisers and Database Proxies.

ZEUS provides several tools to aid in the management of a running multiagent system. They are:

- report tool: provides a visualisation of the (sub)tasks
- statistics tool: gathers individual agents and society-wide statistics
- agents viewer: monitors the internal state of agents
- society viewer: shows the message interchange of agents in a society

These tools can both analyse a running system as well as an 'off-line' version, i.e. saved multiagent interaction sessions. This can be achieved by using the Database Proxy agent for logging activities within the multiagent system.

Communication
Communication is handled by a *MailBox* associated with each agent. Messages are created by using a Java *Message* object, which is created with the necessary requirements, such as the recipient, a message-ID etc. The resulting message obeys the FIPA 1997 ACL specification. The communication protocol used by the ZEUS project is standard TCP/IP where messages are translated into an ASCII sequence.

5.3 JACK

JACK is described as an environment for building, running and integrating commercial Java-based multiagent systems using a component-based approach. It is developed by Agent Oriented Software Pty. Ltd., an Australian commercial company [1]. It is based on the BDI model dMARS developed at the Australian Artificial Intelligence Institute (AAII) [20]. The platform is released as a commercial application. Research licenses for JACK are available and an evaluation version is available free of charge.

Purpose
The aim of JACK is to provide an environment for building, running and integrating commercial Java-based multiagent software using a component-based approach [2].

Analysis and specification
No evidence of facilities to aid in the analysis and specification of multiagent systems have been found.

Design
JACK provides a proposed BDI agent model based on dMARS. It is possible to use other agent models, but they have to implemented first.It is assumed that the specifications of the functionality of the agents are provided and that the BDI model has been chosen. If JACK's BDI agent model has been chosen, designing JACK agents then consists of the identification of the mental states elements of the agents (interactions, goals, beliefs, plans). The design further consists of the identification of elementary classes to manipulate the objects identified within the domain, such as actions which should be performed or data structures to represent domain-specific data.

JACK provides a graphical tool to aid in the design of agents. Design diagrams can be made on a graphical workarea, called the *canvas*. These design diagrams should be used to model an agent. JACK agents are build up out of five components: agent, capability, event, plan and beliefset. These design components can be linked together to provide a structure between the elements forming a JACK agent. There is also a plan editing tool which allows plan reasoning to be laid out as a diagram. At run-time this plan editing tools also allows the plans to be traced graphically.

Implementation
Agents are implemented by using the JACK Agent language. The skeletons of the source of agents are generated by using the design tool. The agent-specific components (behaviour, capability, plans, events) are described using the JACK Agent Language (JAL) which is an extension to the Java language. JACK provides a compiler for translation of JAL programs to pure Java programs. The translated Java source code uses another facility offered by JACK: a library of supporting classes, called the JACK Agent Kernel.

Deployment
Once the source JAL codes have been compiled to Java byte code, the multiagent system can be executed, which consists of manually starting the compiled classes. JACK provides a graphical tool to view the progress of plans being executed by agents if these plans have been designed using the provided graphical plan design tool.

Communication
Communication is done by adding JACK program constructs to the agent source code. Sending messages is done with the JAL. JACK provides a communication layer, known as the DCI network, which is used to allow for communication between agent systems running on different platforms. By default, DCI uses the UDP protocol. JACK agents are not bound to any specific agent communication language as this was not a design goal. It is claimed that adding higher level protocols, such as FIPA ACL or KQML, is possible. Recently an effort has been made by the School of Computer Science and Information Technology at the RMIT University to extend the JACK platform with FIPA compliant ACL communication and components such as the AMS, DF and an MTS [35].

5.4 Comparison 3APL

In this section we will compare the 3APL platform to the aforementioned platforms on the following fields:

- Purpose
- Analysis and specification
- Design
- Implementation
- Deployment
- Communication

Purpose
The purpose of our 3APL platform is to provide a software framework aimed at developing multiagent systems and to facilitate the implementation and execution of 3APL agents in a multiagent setting.
Analysis and specification
The ZEUS project is the only reviewed platform which provides a UML-like description of roles for the agents in a multiagent system. The 3APL platform does not provide facilities to aid in the analysis of multiagent systems as this was not an aim of the project.

Design
ZEUS, and to a lesser extent JACK, provide graphical tools to aid in the design of agents. Although extensive, the compelled use of the graphical design tools in ZEUS only allows for multiagent systems with agents that are task-oriented, goal-driven collaborative agents. The 3APL platform also does not provide facilities to aid in the design of agents as this also was not an aim of the project.

Implementation
The three reviewed platforms ultimately use Java source code for the implementation of agents. This makes the specification of an agent using BDI or KARO to an actual software agent difficult to achieve. As we have shown, a 3APL agent is implemented by specifying its initial beliefbase, goalbase, capabilitiesbase and rulebase. This allows for a direct and easy implementation of software agents from their logical and cognitive specifications.

Deployment

During execution of a multiagent system, the 3APL platform provides tools for viewing exchanged messages, the sniffer tool and the communication panel, similar to the tools provided with ZEUS and JADE. The 3APL platform also has the added ability to view the internal states of an agent during execution.

The JADE platform has been used as a reference during the design of the 3APL platform. During the examination phase, the question arose about implementing the 3APL platform as a module and incorporate it within the JADE platform. The following considerations have been made during the examination phase. On the JADE platform a user defines an agent by implementing a Java class that extends the base Agent class. A user-defined agent is constructed by adding one or several "Behaviours" to this derived Agent class. The computational model of a JADE agent is multitask, where tasks (or behaviours) are executed concurrently. Each functionality/service provided by an agent should be implemented as one or more behaviours. A scheduler, implemented by the base Agent class and hidden to the programmer, carries out the scheduling of all behaviours available in the ready queue, executing a Behaviour-derived class until it will release control. If the task relinquishing the control has not yet completed, it will be rescheduled the next round. The Agent class exposes two methods *addBehaviour*(*Behaviour*) and *removeBehaviour*(*Behaviour*), which allow to manage the ready tasks queue of a specific agent.

Following this methodological guideline, the deliberation cycle of 3APL should be implemented as a behaviour. A behaviour, which is implemented as a Java class, must be supplied with java code by the user. This code is executed when the behaviours *action*() method is called by the scheduler. The 3APL deliberation comprises of sequential steps which is a fixed procedure, so the best-suited behaviour would be the SequentialBehaviour [16]. The deliberation cycle is the only behaviour, or task, the cognitive agent would have to implement. This would render the scheduler useless. The JADE user interface is also different from the one we had specified during the design of the 3APL platform. This implies that also the user interface of JADE had to be altered significantly. The fact that the 3APL interpreter should be reimplemented together with the fact that we do not use the scheduler efficiently has been a good reason for us to develop the 3APL platform. Another possibility would be to use JADE as a library. A singleton instance of the JADE Runtime can be obtained to access a wrapper object. This object provides functionalities such as installing and uninstalling Message Transport Protocols (MTP) and creating new agents. However, the only functionality useful for the platform would be the MTP. It is more efficient to only use a smaller library for the handling of messages across different platforms. The Java Agent Services (JAS) library is used for this purpose as we have seen in the implementation of the AMS class. We also looked at incorporating only modules of the JADE platform in our platform, such as the sniffer module. However it quickly became clear that this module could not be integrated into the 3APL platform without large rewriting because of its tight bonding with the JADE platform itself.

Communication

A test for compliance to the FIPA standards concerning communication is the agentcities network services initiative [3]. This initiative enables the verification of connecting a platform with another platform hosted on the agentcities network. JADE passes this test without any modifications. Although the ZEUS platform conforms to the FIPA 1997 ACL specification considerable adaptations have to be done in order to connect to an agentcity [17]. The recent initiative of FIPA JACK [35] has enabled a JACK mutliagent system to connect to an agentcity. The 3APL platform currently is not able to connect to the agentcities network. The reason for this is the notion that most of the nodes listed on the agentcities network are standard JADE platform using the CORBA ORB as a transportation protocol. This protocol is currently not supported by the JAS library. The JAS Message Transport Protocol (MTS) implementation currently allows only entities, which are AMSes on a 3APL platform in our case, that have been registered with the name server to be addressed as a recipient of a message. This means that a JADE platform should be known to the JAS name server, and this is not yet possible.

6 Conclusion and Future work

This thesis began with the observation that there does not yet exist an agent platform which can handle cognitive agents, and especially a platform for 3APL agents. Thus the aim of this project was to design and implement a 3APL platform, in which we have succeeded. However, during the design and implementation we have decided to cover some but not all relevant issues regarding multiagent platforms since we had a limited time. Some aspects which have not been covered are left for future research. We will mention these aspects at the end of this conclusion.

In particular, in the second chapter we gave an introduction to the language 3APL. To allow communication between agents and solve common problems such as locating agents, naming, etc. a platform had to be constructed. Chapter 3 lists some of the requirements given by the FIPA in order to construct such a platform. In this chapter we also looked at what the 3APL platform offers in terms of those requirements. Although the FIPA definitions of performatives still needs to be worked upon, the translation between these performatives to constructs in the 3APL language is quite easy to achieve, as we have shown in the concluding examples of chapter 3. Because of the initial aims of this project, namely the development of an experimental platform, chapter 4 focuses on the graphical components and tools available to the programmer. This chapter also describes the implementation in a broad view, and a more elaborate description of the two most important classes of the platform, the Agent and AMS class. There already exist several other agent platforms. In chapter 5 we looked at three other agent platforms and compared them on purpose, communication and the four stages of traditional software design (analysis and specification, design, implementation and deployment).

Although this project has delivered a working 3APL platform with most of the functionality as was intended, a lot of work still can be done to improve the 3APL platform. Especially compared to the more mature platforms reviewed. In this section we will list some suggestions for future work.

- The addition of a belief operator to the 3APL language [31]. This will enable the translation between the semantic of the definitions of the FIPA ACL performatives and 3APL to be even more intuitive. The usage of a belief operator is currently not possible in the 3APL language. To make use of this operator, the syntax of 3APL would have to be adjusted to allow for this operator and its semantics. The addition of a belief formula has not been added by this project, as it was not one of the design goals.

- Lately there has been a lot of research in the area of a role of a platform [4]. This means that the platform 'guides' the agents present to have a certain behavior. This also includes security issues, such as determining whether an agent is allowed to use functionalities offered by the platform, or which agents it is allowed to communicate with. The 3APL platform should be extended to allow for this functionality.

- Achieving a greater adherence to the FIPA standard by implementing a true directory facilitator (DF). The functionality of a DF, the retrieval of agents using service descriptions, is currently simulated by providing descriptions to the AMS. A great feature of DFs is the ability to be federated, which means that several DFs can subsribe to each other and can make use of one another to find agents. A (federated) DF has not been implemented due to lack of time.

- Achieving a greater adherence to the FIPA standard by implementing usage of ontologies. An ontology is a "specification of a representational vocabulary for a shared domain of discource - definitions of classes, relations, functions, and other objects" [10]. The JADE and ZEUS platform both support the use of ontologies.

- JADE and ZEUS also provide security features [9]. This enables for example the exclusion of agents that do not have the right credentials or encrypting messages sent to agents. The 3APL platform currently does not have any security features as this was not a design goal.

- One of the benchmarks for complying with the FIPA standards is the ability to connect the platform to the agentcities network services [3]. This is a online community which has several different platforms present. As we have shown in chapter 5, the reviewed platforms, recently including JACK, are able to connect to an agentcity. The 3APL platform currently cannot connect to this community and making this a reality has not been achieved due to lack of time.
- The AMS should be extended with more functionality, for instance the addition of a SQL-like language to allow for more elaborate searching of the agents descriptions.
- Agent mobility is currently a hot topic. Perhaps a more lightweight version of the platform should be developed to allow mobile devices to use the platform. Currently, every 3APL agent has its own instance of a JIP engine for the beliefbase. This adds approximately 4 MB of memory for each agent present on the platform. The beliefbases should use one shared JIP engine to allow for a greater number of agents to be hosted on the platform of a mobile device.
- The latest version of the 3APL interpreter also allows for Java code to be executed by the agent. A library of common Java programs, such as a random number generator or even database connectivity, could be added to the platform to allow an easier way for users to connect the platform with external programs or resources.

Appendix

A

```
PROGRAM "center"

CAPABILITIES:
 {player(X0,Y0) AND ball(X1,Y1)}
   MoveToBall()
 {NOT player(X0,Y0), player(X1,Y1)}

 {ball(X0,Y0)}
    Kick(X1,Y1)
 {NOT ball(X0,Y0), ball(X1, Y1)}

 {received(A,B,query,position)}
    Midfielder(B)
 {NOT received(A,B,query,position), midFielder(B)}

 {ball(X0,Y0), goal(X0,Y0)}
    Won()
 {haveWon()}

BELIEFBASE:
 player(200,100),
 ball(50,80),
 goal(0,50)

GOALBASE:
 Init(),
 Wait(),
 Shoot(),
 Close()

RULEBASE:
 Init() <- player(X,Y) |
          BEGIN
          Java("Main",set(2,X,Y),U);
          Send(2,ams,inform,description(isCenter))
          END
,
 Wait() <- received(A,B,query,position) AND player(X,Y) |
         BEGIN
         Midfielder(B);
         Send(A,B,reply,position(X,Y))
         END
,
 Shoot() <- received(A,B,inform,shoot) AND goal(X1,Y1) |
           BEGIN
           WHILE ball(X0,Y0) AND NOT ((X0 = X1) AND (Y0=Y1))  DO
               BEGIN
                 MoveToBall();
                 Kick(X1,Y1);
                 Won()
```

```
                END
            END
,
 Close() <- midFielder(B) AND haveWon() |
            BEGIN
            Send(2,B,inform,haveWon());
            Send(2,ams,inform,NOT description(isCenter))
            END
.
```

Listing A1. The 3APL source code for the center football player.

B

JADE

```
package examples.PingAgent;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;

/** @author Tiziana Trucco - CSELT S.p.A. @version  $Date:
2002/08/02 08:10:01 $ $Revision: 1.5 $ */


public class PingAgent extends Agent {

    class WaitPingAndReplyBehaviour extends SimpleBehaviour {

    private boolean finished = false;

    public WaitPingAndReplyBehaviour(Agent a) {
        super(a);
    }

    public void action() {

        ACLMessage  msg = blockingReceive();

        if(msg != null){
        if(msg.getPerformative() == ACLMessage.NOT_UNDERSTOOD){
            //received a NOT-UNDERSTOOD message
            }
        else{

            ACLMessage reply = msg.createReply();

            if(msg.getPerformative()== ACLMessage.QUERY_REF){
            String content = msg.getContent();
```

```
            if ((content != null) && (content.indexOf("ping") != -1)){
                //received a QUERY_REF with correct content.
                                reply.setPerformative(ACLMessage.INFORM);
                reply.setContent("alive");
            }
            else{
                //received a QUERY_REF with uncorrect content.
                reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
                reply.setContent("( UnexpectedContent (expected ping))");
            }

            }
            else {
            //received a wrong performative.
            }
            send(reply);
        }
        }else{
        System.out.println("No message received");
        }
    }

    public boolean done() {
        return finished;
    }
    } //End class WaitPingAndReplyBehaviour

    protected void setup() {
    /** Registration with the DF */
    DFAgentDescription dfd = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("AgentcitiesPingAgent");
    sd.setName(getName());
    sd.setOwnership("TILAB");
    dfd.setName(getAID());
    dfd.addServices(sd);
    try {
        DFService.register(this,dfd);
    } catch (FIPAException e) {
        System.err.println(getLocalName()+" registration with DF unsucceeded.
            Reason: "+e.getMessage());
        doDelete();
    }
    WaitPingAndReplyBehaviour PingBehaviour = new  WaitPingAndReplyBehaviour(this);
    addBehaviour(PingBehaviour);

    }
}//end class PingAgent
```

ZEUS

part 1. The automatically generated source code.

```
/*
    This software was produced as a part of research
    activities. It is not intended to be used as commercial
```

```
    or industrial software by any organisation. Except as
    explicitly stated, no guarantees are given as to its
    reliability or trustworthiness if used for purposes other
    than those for which it was originally intended.

    (c) British Telecommunications plc 1999.
*/

/* This code was automatically generated by ZeusAgentGenerator
version 1.1
                           DO NOT MODIFY!!
*/

import java.util.*;
import java.io.*;
import zeus.util.*;
import zeus.concepts.*;
import zeus.actors.*;
import zeus.agents.*;

public class PingAgent {
   protected static void version() {
      System.err.println("ZeusAgent - PingAgent version: 1.1");
      System.exit(0);
   }

   protected static void usage() {
      System.err.println("Usage: java PingAgent -s <dns_file>
                      -o <ontology_file> [-gui ViewerProg]
                      [-e <ExternalProg>] [-r ResourceProg]
                      [-name <AgentName>] [-debug] [-h] [-v]");
      System.exit(0);
   }

   public static void main(String[] arg) {
   try {
      ZeusAgent agent;
      String external = null;
      String dns_file = null;
      String resource = null;
      String gui = null;
      String ontology_file = null;
      Vector nameservers = null;
      String name = new String ("PingAgent");
      Bindings b = new Bindings("PingAgent");
      FileInputStream stream = null;
      ZeusExternal user_prog = null;


      for( int j = 0; j < arg.length; j++ ) {
         if ( arg[j].equals("-s") &&  ++j < arg.length )
            dns_file = arg[j];
         else if ( arg[j].equals("-e") &&  ++j < arg.length )
            external = arg[j];
```

```
         else if ( arg[j].equals("-r") &&  ++j < arg.length )
            resource = arg[j];
         else if ( arg[j].equals("-o") &&  ++j < arg.length )
            ontology_file = arg[j];
         else if ( arg[j].equals("-gui") &&  ++j < arg.length )
            gui = arg[j];
         else if ( arg[j].equals("-debug") ) {
            Core.debug = true;
            Core.setDebuggerOutputFile("PingAgent.log");
         }
         else if ( arg[j].equals("-v") )
            version();
         else if ( arg[j].equals("-name")  && ++j < arg.length )
             name = name + "_" + arg[j];
         else if ( arg[j].equals("-h") )
            usage();
         else
            usage();
      }

       b = new Bindings(name);
      if ( ontology_file == null ) {
         System.err.println("Ontology Database file must be specified with -o option");
         usage();
      }
      if ( dns_file == null ) {
         System.err.println("Domain nameserver file must be specified with -s option");
         usage();
      }

         nameservers = ZeusParser.addressList(new FileInputStream(dns_file));
         if ( nameservers == null || nameservers.isEmpty() )
            throw new IOException();

         agent = new ZeusAgent(name,ontology_file,nameservers,1,20,false,false);

         AgentContext context = agent.getAgentContext();
         OntologyDb db = context.OntologyDb();


/*
         Initialising Extensions
*/
         Class c;

         if ( resource != null ) {
            c = Class.forName(resource);
            ExternalDb oracle = (ExternalDb) c.newInstance();
            context.set(oracle);
            oracle.set(context);
         }
         if ( gui != null ) {
            c = Class.forName(gui);
            ZeusAgentUI ui = (ZeusAgentUI)c.newInstance();
```

```
            context.set(ui);
            ui.set(context);
         }

/*
         Initialising ProtocolDb
*/
         ProtocolInfo info;

/*
         Initialising OrganisationalDb
*/
         AbilityDbItem item;
/*
         Initialising ResourceDb
*/
         Fact f1;


/*
         Initialising External User Program
*/

         if ( external != null ) {
            c = Class.forName(external);
            user_prog = (ZeusExternal) c.newInstance();
            context.set(user_prog);
         }


/*
         Activating External User Program
*/

         if ( user_prog != null )
            user_prog.exec(context);

      }
 catch (ClassNotFoundException cnfe) {
     System.out.println("Java cannot find some of the
       classes that are needed to run this agent.");
   cnfe.printStackTrace();}
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

part 2. The external program

```
import zeus.actors.*;
import zeus.util.*;
import zeus.concepts.*;
import zeus.agents.*;
import java.awt.*;
```

```
import java.awt.event.*;
import javax.swing.*;

public class pingExternal  extends JFrame implements
ZeusExternal,ActionListener {
    JButton sendButton = new JButton("Send");
    AgentContext context = null;

    public pingExternal () {
        setSize (200,200);
        GridLayout layout = new GridLayout(1,1);
        getContentPane().setLayout(layout);
        getContentPane().add (sendButton);
        sendButton.addActionListener(this);
        repaint();

    }

    public void actionPerformed (ActionEvent ae) {
        sendMessage();
    }

    public void exec(AgentContext context) {
        this.context = context;
        setVisible (true);
        setResponse ();
    }

    public void sendMessage() {
        Performative ping = new Performative ("query-ref") ;
        ping.setReceiver("ping_agent");
        ping.setContent ("ping");
        context.getMailBox().sendMsg(ping);
    }

   public void setResponse () {
        SimpleAPI api = new SimpleAPI(context);
        api.setHandler ("query-ref",this,"respond");
   }

    public void respond(Performative msg) {
        Performative resp = new Performative ("inform");
        resp.setContent ("alive");
        resp.setSender (msg.getReceiver());
        resp.setReceiver(msg.getSender());
        resp.send(context);
        }
}
```

JACK

```
event PingEvent extends MessageEvent {
   int value;

#posted as ping(int value)
```

```
{
   this.value = value;
   }
}

plan BouncingPlan extends Plan {

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

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

    agent PingAgent extends Agent {

    #handles event PingEvent;
    #uses plan PingPlan;

    #posts event PingEvent pev;

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

3APL Platform

```
PROGRAM "pingAgent"

CAPABILITIES:

BELIEFBASE:

GOALBASE:
 Ping()

RULEBASE:
 Ping() <- received(I,S,P,C) |
  BEGIN
  Send(I,S,reply,received(true));
  Ping()
  END
.
```

References

[1] JACK Intelligent Agents. http://www.agent-software.com.au/shared/home.

[2] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java, 1999.

[3] Agent Cities. http://www.agentcities.net/.

[4] M. Dastani, V. Dignum, and F. Dignum. Role-assignment in open agent societies. In *Proceedings of the Second International Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*. Utrecht University, ACM Press, july 2003.

[5] M. Dastani, J. van der Ham, and F. Dignum. Communication for goal directed agents. In Marc-Philippe Huget, editor, *Communication in Multiagent Systems - Agent Communication Languages and Conversation Policies*, pages 239–252. LNCS, 2002.

[6] FIPA. http://www.fipa.org.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[8] G. De Giacomo, Y. Lespérance, and H. Levesque. ConGolog, a concurrent programming language based on the situation calculus. In *Artificial Intelligence*, number 121(1-2), pages 109–169, 2000.

[9] NGUYEN T. Giang and DANG T. Tung. Agent platform evaluation and comparison. Technical report, Institute of informatics, Slovak academy of sciences, june 2002.

[10] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5:199–220, june 1993.

[11] K.V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi Agent systems*, 2 (4):357–401, 1999.

[12] D. Kinny. The psi calculus: An algebraic agent language. In J.-J. C. Meyer and M. Tambe, editors, *Intelligent Agents VIII, Agent Theories Architectures and Languages, LNAI 2333*, pages 32–50. Springer-Verlag, 2001.

[13] 3APL library documentation. http://www.cs.uu.nl/3apl/doc/index.html.

[14] JI Prolog library documentation. http://www.ugosweb.com/jiprolog/doc/index.html.

[15] Hyacinth S Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis. ZEUS: a toolkit and approach for building distributed multi-agent systems. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 360–361, Seattle, WA, USA, 1999. ACM Press.

[16] JADE programmers guide. http://sharan.cselt.it/projects/jade/doc/programmersguide.pdf.

[17] ZEUS project. http://more.btexact.com/projects/agents/zeus.

[18] A. Rao and M. Georgeff. Modeling rational agents within a BDI architecture. In *Proceedings of the KR91*, 1991.

[19] A.S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In W. van der Velde and J. Perram, editors, *Agents Breaking Away (LNAI 1038)*, pages 42–55. Springer-Verlag, 1996.

[20] Pierre-Michel Ricordel and Yves Demazeau. From analysis to deployment: A multi-agent platform survey. *Lecture Notes in Computer Science*, 1972:93–??, 2001.

[21] Java Remote Method Invocation (RMI). http://java.sun.com/products/jdk/rmi/.

[22] John R. Searle. *Speech acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969.

[23] Java Agent Services. http://www.java-agent.org.

[24] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, (60):51–92, 1993.

[25] 3APL site. http://www.cs.uu.nl/3apl/.

[26] Fabio Bellifemine (CSELT S.p.A.), Agostino Poggi (DII University of Parma), Giovanni Rimassa (DII University of Parma), and Paola Turci (DII University of Parma). An object oriented framework to realize agent systems. In *Proceedings of WOA 2000 Workshop*, pages 52–57. WOA, may 2000.

[27] FIPA ACL Message Structure Specification. http://www.fipa.org/specs/fipa00061/sc00061g.html.

[28] FIPA Agent Management Specification. http://www.fipa.org/specs/fipa00023/sc00023j.html.

[29] FIPA Communicative Act Library Specification. http://www.fipa.org/specs/fipa00037/sc00037j.html.

[30] Java Swing Worker Thread. http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html.

[31] Jeroen van der Ham. Multi-agent FIPA compliant 3APL agents. Master's thesis, Utrecht University, september 2002.

[32] B. van Linder, W. van der Hoek, and J.-J. Ch. Meyer. Formalizing abilities and opportunities of agents. In *Fundameta Informaticae*, volume 34(1,2), pages 53–101. 1998.

[33] Marko Verbeek. 3APL as programming language for cognitive robots. Master's thesis, Utrecht University, february 2003.

[34] Mike Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In P. Ciancarini and M. Wooldridge, editors, *First Int. Workshop on Agent-Oriented Software Engineering*, volume 1957, pages 1–28. Springer-Verlag,Berlin, 2000.

[35] Kenichi Yoshimura. FIPA JACK: A plugin for JACK intelligent agents. Technical report, RMIT University, september 2003.