An Internal Agent Architecture Incorporating Standard Reasoning Components and Standards-based Agent Communication

Mengqiu Wang, Martin Purvis, and Mariusz Nowostawski
Department of Information Science
University of Otago
Dunedin, New Zealand
{mwang, mpurvis, mnowostawski}@infoscience.otago.ac.nz

Abstract

This paper discusses a general architecture for intelligent software agents. It can be used to construct agents that engage in high-level reasoning by employing standard reasoning engines as plug-in *components, while communicating with other agents by means of the standard FIPA-based communication protocols. The approach discussed uses internal* micro-agents *and declarative goals to form a hierarchical internal agent architecture. It has been implemented and tested with three high-level formal reasoning system components that are used in conjunction with an existing agent platform, OPAL, which supports the FIPA (Foundation for Intelligent Physical Agents) communication standards.*

1. Introduction

Multi-agent systems presuppose an open dynamic environment in which new agents can come and go. Standardisation efforts in the agent area have so far focused on providing standard inter-agent communication protocols and facilities, such as FIPA[3] and JAS[9], which enable agents to communicate without requiring them to gain inside knowledge of each other. The internal architecture of an agent has not been standardised. There are however a few declarative reasoning methods that are frequently used. It is widely believed that by using declarative programming facilities, the semantics gap between an agent specification and its implementation in a programming language can be reduced [1, 14].

While the number of agent platforms and reasoning components grows, the goal of achieving open agent-based environment remains in the distance. All the commonly used "standard" agent platforms provide only the basic services, such as agent management, directory services, and naming services [4]. The actual agents on such platforms are developed primarily in some arbitrary imperative programming language, such as Java, thereby leaving the semantic gap as before.

One of the few platforms that do attempt to treat this problem and provide declarative support is the 3APL Platform developed at the University of Utrecht [7], by the same group that invented the 3APL language. But the 3APL Platform has a few limitations. In particular, it has a closed architecture as opposed to an open architecture: only 3APL agents can be hosted on the platform. Consequently, the platform is subject to whatever drawbacks the language itself may have.

This paper explores the bonding between high-level reasoning engines and low-level agent platforms in the practical setting of using the 3APL language, the OPAL platform[11], the ROK system and the ROK scripting language[10] and the JPRS reasoning engines[10]. We focus our interest in this paper on our approach to provide declarative agent programming support in the OPAL platform, and show how declarative goals can be used to glue the internal micro agents in OPAL to form the hierarchical architecture of the platform.

The theoretical extension discussed in this paper is accompanied by a practical implementation. The extended OPAL platform is now equipped with three powerful high-level declarative agent language and reasoning engines, as well as with a graphical IDE for constructing complex agents with a hierarchical structure.

2. The 3APL Language

3APL[1] is a programming language for implementing cognitive agents that was developed at the University of

1 3APL stands for An Abstract Agent Programming Language, and is pronounced "triple A P L".

Utrecht, the Netherlands. It incorporates the classical elements of BDI logic and also includes first-order logic features. It provides programming constructs for implementing agent beliefs, declarative goals, basic capabilities, such as belief updates or motor actions, and practical reasoning rules through which an agent's goals can be updated or revised [1].

An agent's beliefs are represented in 3APL as Prolog-like well-formed-formula (wff).

The most primitive action that an agent is capable of performing is called a *basic action*, which is also referred to as a *capability*. In general, an agent uses basic actions to manipulate its mental state and the 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. Basic actions are the only constructs in 3APL that can be used to update the beliefs of an agent.

A 3APL agent has two types of goals: basic goals and composite goals. There are three different types of basic goals: basic action, test goal, and the predicate goal. A test goal allows the agent to evaluate its beliefs, i.e. a test goal checks whether a belief formula is true of false.

This type of goal is also used to bind values to variables, like variable assignment in ordinary programming languages. When a test goal is used with a variable as a parameter, the variable is instantiated with a value from a belief formula in the belief-base. The third type of goal is a predicate goal. It can be used as a label for a procedure call. The procedure itself is defined by a practical reasoning rule. Practical reasoning rules can be used to generate reactive behavior, to optimize the agent's goals, or to revise the agent's goals to get rid of unreachable goals or blocked basic-actions. They can also be used to define predicate goals (i.e. procedure calls).

From these three types of basic goals, we can construct composite goals by means of various operators. A special type of goal that has been recently added is JavaGoal. This type of goal enables the programmer to load an external Java class and invoke method calls on it.

3. Rule-driven Object-oriented Knowledge-base System

ROK, Rule-driven Object-oriented Knowledge base system, is a forward chaining production rule system derived from JEOPS. JEOPS was developed by Carlos Figueira Filho and Carlos Cordeiro [2]. ROK provides a library and API written in Java, with a mechanism for embedding first-order, forward-chaining production rules into Java applications. It was created to provide the declarative expressiveness of production rules, which is useful for the development of large or complex systems [10]. ROK production rules can be described as condition-action patterns. Any Java object can be matched in a ROK rule, and any Java expression can be used in the condition and action part of ROK rules. There are two major modes of operation for a ROK system: native and interpreted. In the native mode the programmer declares the rules using the provided Java API. In the interpreted mode, users prepare the rules as a text script file to be parsed and interpreted by the ROK interpreter. In the interpreted mode, the programmer is freed from writing Java code and only has to write declarative pseudo-Java scripts. But there is a performance trade-off: native mode is faster in execution and is the most efficient method of operation.

3.1. Internal Structure of ROK

The heart of the ROK system is composed of three main building blocks: the object-base, the rule-base and the conflict set. The object-base is the working memory, where the facts that the agent knows are stored. The rules written by the programmer or compiled from rule scripts are placed (installed) inside the rule-base. The rule-base is the place where all the information about the rules is stored, such as their declarations, conditions, actions, and several other items of control information. The RETE network [6]is used to store the partial matches between rules and objects, and to increase the performance of the matching process. Finally, the conflict set is the component in which the rules that can be fired at a certain moment are stored, as well as the objects that have been matched to the rule declarations.

4. The Java Procedural Reasoning System

JPRS, Java Procedural Reasoning System, is a Java library and API written for performing goal-driven procedural reasoning. Its ancestors can be traced back to the architecture of the PRS system proposed by Georgeff [5], as well as UMPRS and JAM [8]. The notion of procedural reasoning is derived from the idea that some of human knowledge can be best represented as a set of procedure/steps performed in order to achieve a particular goal. A simple example of procedural reasoning can be the planning of a trip from Dunedin to Beijing. The goal is to start off from an apartment in Dunedin, and end up in Beijing International Airport. One of the possible plans is: make a booking, then pay for a economic class ticket from Dunedin to Beijing, take a shuttle to the Dunedin Airport, transit at Sydney Airport, and finally get off the plane at Beijing Airport. An alternative plan would be to take a taxi to Dunedin Airport, pay for a first class direct flight, and then get off at Beijing Airport. We may decide to choose a plan based on how much money or time we have, or the level of service we are seeking. Those represent part of our knowledge about the external world, in other words, our beliefs. Each JPRS agent

is composed of four primary components: a world model, a plan library, a plan executor, and a set of goals. The world model is an object-base that represents the beliefs of the agent. In the previous example, the agent may store information, such as a bank balance, travel departure date, etc. The plan library is a collection of plans that the agent can use to achieve its goals. The plan executor is the agent's "brain" that reasons about what the agent should do and when it should do it. An agent finishes its tasks when there are no more goals to be achieved. JPRS is implemented as a framework-like library for declaring the plans and goals, which provides the specific conventions for declaring goals and plans. The formal syntax and semantics of JPRS are available at [10].

5. Hierarchical Agent Architecture Using Micro-Agents

Before we discuss how the reasoning engines are incorporated into OPAL, it is necessary to describe the system architecture. The notion of agency on OPAL is used at all levels of abstraction. At the lowest abstraction level *micro agents*, which are the closest agent entities to the machine platform, are used. In order to be efficient at this fine-grained level, they do not have all of the qualities often attributed to typical, more coarsely-grained agents. In contrast to higher abstraction level agents, such as those based on FIPA specifications [3], micro agents are more concerned with efficiency and thus do not have all of the qualities and flexibility of FIPA-compliant agents. For instance, micro agents employ a simpler form of agent communication (they communicate via method calls) and are implemented by extending predefined Java classes and interfaces [12].

There are two kinds of micro-agent: primitive and non-primitive. Primitive agents use native services, in particular native micro-kernel libraries, and directly interact with the underlying virtual machine (in our case the Java Virtual Machine). Non-primitive micro-agents, which are typically more sophisticated and exist at a higher abstraction level, are composed only of micro-agents and do not use any native services directly (only via micro-agents).

Because the smallest building block in OPAL is an agent, the system designer can apply agent-oriented modeling methodology throughout the development process. There are two basic constructs in the micro agent system, namely agents and roles. Agents represent actors in a system that can play one or more roles. A role represents a cohesive set of services that may be provided by some agent. Agents that perform the same role are not restricted in the way that they provide the services as prescribed by the role.

There is a special type of role called a group role. When an agent performs a group role, it acts as the group owner and creates a group environment in which other agents could register as group members. By registering with a group, an agent is associated with the group owner and can collaborate with the owner agent. For example, upon receiving a task to solve or a goal to achieve, the group owner can choose to disseminate the goal to its group members and request the members to achieve the goal. And alternatively, if a group member performs a role that the owner, itself, doesn't perform, the owner may still advertise itself as an actor of that role. When the set of services of the role are subsequently requested, it can request its group members to provide the necessary services. The group membership can be dynamically modified according to the needs of individual agents. For example, if an agent is managing a group with too many members, and the action for searching for the right member becomes a lengthy operation, it may decide to dispose some not frequently used group members. Also, an agent may decide to unregister itself from a group because it is more often needed in another agent group.

Although the group concept can be effectively used for organizing agents into hierarchies, one is still faced with the problem of providing ways for the agents to exchange information and cooperate at semantic levels. One way for micro agents to talk to each other and share their capability is to use role-matching. When an agent needs other agents to perform certain services, it will need to know what type of role provides such services and then will need to recursively search through other agents and their groups for that role. If roles and services could be specified declaratively, this approach would suffice for systems in which agents would be requesting new services dynamically. But because roles and services are such generic concepts, difficulties arise in defining formal semantics specifying the services. And also, since OPAL is written in Java, a complete high level language built on top of Java is needed to support specifying services declaratively at runtime. In the current OPAL implementation, roles and services are not declarative constructs. The role-matching approach would only be suitable for systems in which all services are known before runtime. This poses potential restrictions on the dynamism and robustness of the systems.

A second approach, which is the one we are currently taking, relies on declarative goals to aid in the cooperation among micro agents. The meaning of a goal in OPAL is similar to the meaning of a goal in 3APL. It typically specifies some post-conditions that represent the states after the goal has been achieved, but doesn't enforce how these post-conditions are to be realised. In other words, a goal carries some declarative information of some state, but not the procedural information on how to reach that state. Agents collaborate through goal exchange. For example, an agent may decide according to its own internal state, what its next goal to be achieved is. And if the agent, itself, is not capable of achieving the goal, or if it wants other agents to pro-

vide alternative solutions to achieve the goal, it can send the goal to other agents. Goals in OPAL are self-descriptive, other agents can evaluate the goals and try their own way of achieving the goal. At the end it will inform the initiating agent whether it succeeded in achieving the goal or not. Similar to the role-matching approach, the goals can be recursively passed down through the agent hierarchy.

To better illustrate the goal passing concept, we will elaborate on a simple scenario as in Figure 1.

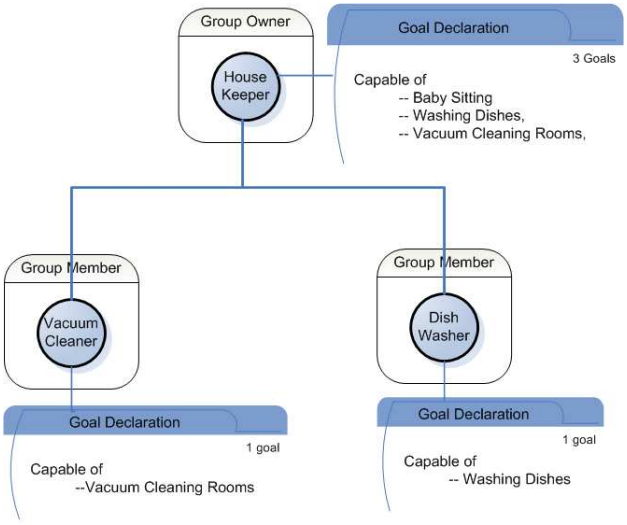

Figure 1. Goal Passing

In this scenario, we have three micro-agents: a dish-washing agent, a vacuum-cleaning agent and a house-keeping agent. Each of them is capable of handling one type of goal: dish-washing, vacuum-cleaning and baby-sitting, respectively. Unlike the other two, the house-keeping agent is also managing a group, with the other two agents as the group members. Therefore, when it declares the types of goal that it is capable of handling, it includes the goals of its group members. When the house-keeping agent is passed a goal of vacuum-cleaning, it will delegate the goal to its group member — the vacuum-cleaning agent. The advantage of using goals instead of roles becomes evident when the system designer can only describe the states of the system declaratively but does not know exactly how the transitions between states take place. In contrast to a service, a goal is a simpler concept and can be formally specified as pre-condition and post-condition clauses, which makes the implementation easier, and facilitate more dynamic interactions among agents. For example, if in a particular application domain, the semantics of the pre and post-conditions of the declared goals are commonly understood among agents, an agent can create new goals on-the-fly at runtime. One can also have rules specifying how to recompose or combine goals.

This hierarchical structure of agents allows us to construct more complex agents. And since the agents, even at the lowest level, are completely autonomous, not only systems that operate in dynamic environments can be modeled using this architecture, but we can also model intrinsically dynamic systems that are changing or evolving over time. Currently, we are investigating a system architecture in which we combine the flexibility of our micro-agent architecture and the power of evolutionary computing techniques, to simulate the evolving process of complex systems. In the proposed system, a large number of micro-agents that are capable of performing very primitive tasks form the initial population. During the evolutionary process, micro-agents can autonomously develop through acquiring other micro-agents, and natural selection criteria are applied to guide the evolutionary search process.

The implication of such an approach is not just a technology blend. There are two main potential advantages of having such a system.

Firstly, most existing multi-agent systems are restricted to a preconceived application domain. Multi-agent systems are most commonly found in modeling simple messaging-oriented tasks, or online auctions. There are no existing easy and convenient ways of embedding complex computational techniques, such as evolutionary techniques, into multi-agent solutions.

Secondly, there exists a natural relationship between agents, evolutionary computing and the biological world. In the biological world, each autonomous organism can be viewed as an agent, and larger biological systems, such as a complex organism, can be viewed as the co-functioning of a large number of those smaller agents. Evolutionary computing techniques, which originally stemmed from the inspiration taken from biological evolution, can be naturally applied to operate on the basis of agents, in a way similar to natural biological evolution. A common problem that occurs when applying evolutionary techniques on real-world problems is finding ways to represent the domain-specific information in appropriate forms accessible to algorithmic computation. The mapping process in traditional evolutionary computing can be unintuitive and convoluted. However, if we are able to model the domain information in agent terms and still be able to apply evolutionary techniques, it would not only ease the design and development of evolutionary systems, but also widen the range of agent-oriented applications.

To illustrate this concept, consider a typical scheduling problem of arranging an itinerary for overseas trips. In the problem specification, a starting point and a destination are given. The goal is to find the best solution in terms of the to-

tal cost of the flights, flight dates available, number of hours on the plane, number of hours waiting at the airport, number of overnight stays required, number of stops, aircraft rating, and the service level of the operating airline. These criteria may receive different weights according to the user's priorities and requirements. Assuming the flight information and requirements are static and fixed, solving such a problem using traditional evolutionary techniques, such as a genetic algorithm, is possible but challenging. One of the main problems is what data structure to use to store all the information, and in what way the selection criteria are going to be used to guide the selection process. Coding up a genetic algorithm solution becomes extremely difficult when the flight information is obtained through travel agencies and thus is changing in real-time. Because whenever the flight information is updated, we have to examine the current population of our solutions and delete all obsolete flights and insert all the new ones. The operation of genetic algorithms and some other similar techniques requires a fixed scope for the functioning of the search mechanism.

The modeling would appear to be more straightforward if we were to use agents. At the base level, each flight is modeled as a micro-agent. The agent keeps track of the availability of the flight, and declares the goal that it is capable of handling, for example, flying from Auckland to Beijing on May 1st 2005. The agent will fail to achieve the goal if the flight is already full. The agent also keeps other related information of the flight, for example, aircraft rating, service level, number of stops, etc. If a higher level agent is formed by acquiring two base-level flight agents, for example, Singapore Airline flight from Auckland to Sydney and Thai Airline flight from Sydney to Hongkong, then it can declare itself as capable of achieving goals of flying from Auckland to Sydney, Sydney to Hongkong, and Auckland to Hongkong. Each agent in the population will be rewarded based on both its own traits (number of stops, price, etc.) and the number of times it has been used by other agents. This rewarding mechanism can be adapted to be more sophisticated if required. Unfit agents (except those base-level ones) are removed over time and fitter ones are promoted. The evolving process continues until the convergence of the population reaches a threshold value or optimal solutions are found, just as in traditional evolutionary systems. Because the smallest unit in the population is an agent, which principally retains full autonomy, it can autonomously update itself, and therefore can be used to model real-time airline agencies and other types of real-time agents.

The hierarchical agent architecture is also highly modular and open. Since micro agents communicate with each other through declarative goals, their internal structure or state is hidden from each other. This important feature allows us to introduce new components into the platform easily. In the next section we describe how we integrate the high level reasoning engines and programming languages into OPAL.

6. Integrating 3APL, ROK and JPRS into OPAL

To integrate the three high level reasoning engines into OPAL, our idea is to introduce them as special micro agent components. It means that apart from having the original Java primitive micro agents, we also have three special kinds of micro agents — 3APL micro agent, ROK micro agent, and JPRS micro agent. The integrating process for the three components are the same in principle, and only differ slightly in implementation.

The 3APL micro agent class has a 3APL interpreter and a 3APL engine as its core. It inherits the role playing and group behavior from the primitive micro agent class. The 3APL micro agent loads its source from a prepared 3APL program script and processes the source using the interpreter.

When the OPAL platform receives an incoming message of the appropriate form, it is inserted into the 3APL system as a belief. By inserting the OPAL goal as a belief, we leave the handling of the goal to the programmer. The programmer can write rules coping with the belief change caused by receiving goals.

We take almost identical approaches for integrating ROK and JPRS micro agents. Upon receiving messages or goals, the information is wrapped up as an item of belief and inserted into the knowledge-base of ROK and JPRS agents, respectively.

Also worth noting is the knowledge sharing among micro-agents. Because the hierarchical grouping of agents is purely logical, when several micro-agents form a higher-level agent, there will be no global knowledge pool created for the higher-level agent. But rather, the low-level micro-agents manage their own local knowledge, and share it through communication, which appears to higher level of abstraction as if there is a common knowledge storage.

7. Performance Comparison of the Three Reasoning Engines

The absolute speeds (performance) of these reasoning engines are difficult to measure, because implementation bias in the test programs is inevitable. And also in a multi-agent environment, agents that are powered with these reasoning engines spend their processing time not just on local computation, but also on communication. Although precise quantitative figures are difficult to obtain, we believe some computationally intensive test could still give us suggestive evidences on the relative speed of the reasoning engines. We

Size	3APL	ROK Script	ROK Native	JPRS
50	12078	361	110	40
100	28632	661	141	80
200	83040	801	200	350
300	178797	841	360	841
400	318919	1072	341	1773
500	519127	1312	390	3265
600	761235	1472	441	6319
1000	...	3565	1402	3004
2000	...	6920	2924	20250
3000	...	10836	4737	68318
4000	...	16854	9374	168853
5000	...	21892	10495	314111

Table 1. Mergesort Time (in ms.)

choose to implement mergesort tests using all three engines, because mergesort is a standardized algorithm, which helps to minimize the amount of implementation error or bias introduced.

The tests were run on an Acer machine with a Celeron 2.4 GHz processor and 240MB RAM. We wrote a 3APL program, a ROK program running in native mode, a ROK program running in scripting mode, and a JPRS program, all running mergesort on integer arrays containing random integers.

The average times each program took to sort the arrays are given in Table 1. We can see that ROK in native mode is the fastest. It is about twice as fast as ROK in scripting mode, and many times faster than JPRS, especially when the array size becomes greater than 1000. We also observed that both ROK and JPRS are much faster than 3APL. This result is not surprising, because JPRS and ROK are built closely to primitive Java, and the goal-plan matching algorithm is not computationally expensive. On the other hand, 3APL has a more complex internal structure. It uses a JIProlog engine internally to process prolog-like wffs, and its first order logic features (variables) makes the reification process much more complicated than in ROK or JPRS. The slower speed of 3APL is a trade-off against its declarative expressiveness, and its first order logic features.

8. The Master Mind™Games

We have developed a system for benchmarking, the Master Mind™Game[13], to verify the integration of the high level reasoning engines, and also to demonstrate the two different approaches of agent development in OPAL – declarative and procedural. In the game, the *master mind* holds a key of 4 pegs, each has one of six colors. The *code breaker* tries to deduce the answer by making guesses at it. *master mind* will mark each guess with a *black* or a *white* marker. A *black* marker means one of your pegs is the correct colour and in the correct position. A *white* marker tells you that one of the pegs in the guess is of a colour which is in the solution, but not in the correct position. A full description of the Master Mind game can be found in Nelson [13]. In our implementation, three high level OPAL agents were developed. Each high level agent is composed of two lower-level micro agents. One of them is in charge of FIPA-compliant messaging services, and the other micro agent is the reasoning agent that implements the game logic. The three reasoning micro agents were a 3APL micro agent, a ROK micro agent, and a JPRS micro agent. The high level agents represent *code breakers* and have a common goal of winning the game using as few guesses as possible. They share information and cooperate through the exchange of FIPA messages. The real-world Master Mind game is not a multiplayer game and therefore not well suited for multi-agent system applications. But our purpose is to demonstrate the usability of the extended OPAL platform by showing an example of how one could use all the high-level components in OPAL. The reasoning power of the high-level engines, albeit under-utilized, are still well-demonstrated in this example system. Future work is expected to involve the development of more sophisticated and complex multi-agent applications in OPAL, using the reasoning engines and the declarative programming feature.

9. OPAL IDE

Recently, a graphical IDE has been added to OPAL. The IDE facilitates the design, development and testing phases of agent software development, without having to reboot the platform. Based on the concepts of micro agents and agent-oriented software development, the IDE provides support for:

- creation of new micro-agents by simply dragging icons and plugging them into the existing hierarchy (currently we support the graphical instantiation of 3APL agents, ROK agents, primitive Java agents, OPAL agents and primitive Java roles);
- grouping and regrouping of agents (we currently support moving, regrouping, copying and deleting micro-agents in a drag-n-drop fashion).

The intuitive graphical operations on agents in the IDE are enabled by the underlying more complex interactions with the platform. For example, when a new micro-agent is created, we first determine the agent type and its creator in the agent hierarchy, then we make an instance of the agent, and handle all the necessary registration and associations with other agents. Also, we show such associations to the developer in the GUI. A full scripting interface is implemented for 3APL micro-agents. The user can load a source file,

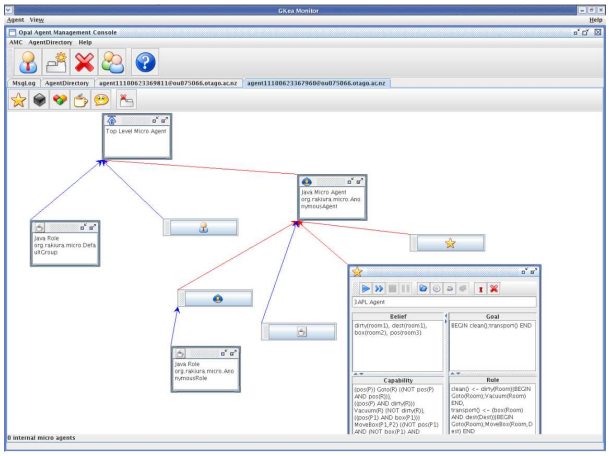

Figure 2. OPAL Agent Composition Panel Screenshot

modify and compile the source file, or create the source file on-the-fly in the text area provided. The IDE is still in the prototyping phase and not yet released. To allow the IDE to support dynamic scripting of general Java agents, we are currently evaluating different approaches of either using our own scripting engine, or relying on customized Java class-loaders. This is the next phase of OPAL development. The screen-shots of the IDE are shown in Figure 2.

10. Conclusion

We argue in this article that declarative agent programming languages and techniques can bridge the semantic gap that exists between agent specification and practical multi-agent system implementations. We have presented our approach of incorporating declarative agent programming support into the OPAL multi-agent platform. In particular, we have described in detail how the agent-oriented hierarchical architecture of OPAL can facilitate and ease the integration of high-level agent programming languages such as 3APL and ROK. The extended OPAL platform allows developers to use the powerful features of declarative languages in developing complex agent systems, while maintaining agent-oriented methodology on all abstraction levels of the systems.

References

[1] Dastani, M., Riemsdijk, B.V., Dignum, F. and Meyer, J.-J. C. (2004). "A Programming Language for Cognitive Agents Goal Directed 3APL", *In Proceedings of the First Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS03)*, Springer, Berlin, 2004.

[2] Figueira, C. and Ramalho, G. (2000). "JEOPS - The Java Embedded Object Production System", *In M. Monard, J. Sichman (eds.). Proc. of 7th Ibero-American Conference on AI (Atibaia, November 19–22, 2000). Lecture Notes in Artificial Intelligence*, pp.53-62, Vol. 1952. Springer-Verlag, Berlin, 2000.

[3] FIPA Organization, `<http://www.fipa.org>`

[4] FIPA. "FIPA Agent Management Specification", `<http://www.fipa.org/specs/fipa00023/sc00023j.html>`

[5] Georgeff, M.P. and Lansky, A.L. (1986). "Procedural Knowledge", *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74(10):1383-1398, October 1986.

[6] Forgy., C. (1982). " Rete: a Fast Algorithm for the Many Pattern/Many Objects Pattern Match Problem", *Artificial Intelligence*, 19:1737, 1982.

[7] Hoeve, E.C.ten (2003). "3APL Platform", *Master's Thesis, University of Utrecht, The Netherlands*, Oct 2003.

[8] Huber, M.J. (2001). "JAM agents in a nutshell", Nov 2001.

[9] Java Agent Service, `<http://www.java-agent.org>`

[10] Nowostawski, M. (2001). "Kea Enterprise Agents Documentation", Aug, 2001.

[11] Nowostawski, M. (2004). "Otago Agent Platform Developer's Guide", Feb 2004.

[12] Nowostawski, M., Purvis, M., and Cranefield, S. (2001). "KEA - Multi-level Agent Infrastructure", *In Proceedings of the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS 2001)*, pp.355-362, Department of Computer Science, University of Mining and Metallurgy, Krakow, Poland 2001.

[13] Nelson, T. (1999). "Investigations into the Master Mind™ Board Game", `<http://www.tnelson.demon.co.uk/mastermind/>`, 1999.

[14] Wooldridge, M.J., and Jennings, N.R. (1995). "Intelligent agents: Theory and practice", *Knowledge Engineering Review*, 10(2), 1995.