Declarative Agent Programming Support for a FIPA-Compliant Agent Platform

Mengqiu Wang, Mariusz Nowostawski, and Martin Purvis

University of Otago, Dunedin, New Zealand,
[mwang, mnowostawski, mpurvis]@infoscience.otago.ac.nz

Abstract. Multi-agent system(MAS) is a blooming research area, which exhibits a new paradigm for the design, modelling and implementation of complex systems. A significant amount of effort has been made in establishing standards for agent communication and MAS platforms. However, communication is not the only difficulty faced by agent researchers. Research is also directed towards the formal aspects of agents and declarative approaches to model agents. This paper explores the bonding between high-level reasoning engines and low-level agent platforms in the practical setting of using three formal agent reasoning implementations together with an existing agent platform, OPAL, that supports the FIPA standards. We focus our discussion in this paper on our approach to provide declarative agent programming support in connection with the OPAL platform, and show how declarative goals can be used to glue the internal micro agents together to form the hierarchical architecture of the platform.

1 Introduction

Multi-agent system(MAS) research is an important and rapidly growing area in distributed systems and artificial intelligence. The agent notion has evolved from a monolithic artifact of software to a new and exciting computing paradigm, and it is now recognized that MAS, as a conceptual model, has the advantages of high flexibility, modularity, scalability and robustness [16]. Proprietary MASs have existed for years, but the lack of agent communication standards hindered the convergence of individual research efforts and restricted further growth until the emergence of the current set of standards, the FIPA[1] specifications and the JAS[2] standards, which afford agents the ability to communicate with each other without requiring them to gain inside knowledge of each other.

However, communication is not the only difficulty faced by agent researchers. There exists a gap between the semantics of an agent and its practical implementation. For example, if an agent is specified to have a BDI[3][19] architecture

[1] FIPA, Foundation for Intelligent Physical Agents, has developed specifications supporting interoperability among agents and agent-based applications[3].

[2] The Java Agent Services (JAS) project defines a standard specification for an implementation of the FIPA Abstract Architecture within the Java Community Process initiative[10].

[3] BDI stands for Belief, Desire and Intention.

but is implemented with a conventional programming language, it is difficult to verify whether the agent satisfies the specifications [1,18]. This problem has been addressed by various research groups around the world, and the result is a set of agent programming languages, e.g. 3APL, Agent-0, AgentSpeak. [7,15,17]. We are particularly interested in the 3APL language developed at Unversity of Utrecht, which allows the programmer to design an agent in a declarative way, by specifying the rules, goals, beliefs and capabilities of the agent. The declarative nature of 3APL helps bridge the semantic gap, and allows flexibility in agent development.

As the number of agent platforms grows, a gap remains inbetween these two building blocks. That is, though an intelligent agent can be built in isolation, one may find it difficult to migrate such an agent onto a platform which hosts other types of agents that it needs to cooperate with. On the other hand, only few agent platforms provide facilities to ease the development of complex agents, or provide a unified approach for integrating such agents onto the platform [16]. Many platforms that exist today only provide the basic services that are required by the standard, such as agent management, directory service, and naming service. [4] Agents on such platforms are developed primarily in some arbitrary imperative programming language, such as Java, but the semantic gap remains.

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 [8], 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[12], the ROK system and the ROK scripting language[11] and the JPRS reasoning engines[11]. 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 hierachical structure.

2 The 3APL Language

3APL[4] is a programming language for implementing cognitive agents, and was developed at the University of Utrecht, the Netherlands. The 3APL language incorporates the classical elements of BDI logic and also embraces 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]. In this section we give a very brief introduction to the main constructs of the language; the formal syntax and the semantics of the 3APL language can be found in Dastani et al. [1].

2.1 Beliefs

An agent's beliefs are represented in 3APL as prolog-like well-formed-formulae (wff). An example of using beliefs to represent information about the environment is an agent with the task of going to a lecture class at a certain time. The agent will have beliefs such as *class_starts_at(X)* and *NOT in_class(self)*. If the agent attends the class, the agent's mental state and the state of the environment change. The belief *NOT in_class(self)* denotes that the agent believes he is not attending a class, so this has to be updated to be *in_class(self)*. The beliefs *NOT in_class(self)* need to be removed from the belief-base(knowledge database) in order to make sure the agent's view of the world is consistent.

2.2 Actions

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.

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 attend a meeting, using the basic action *AttendMeeting(Room1)*. Before this action can be executed, there should be a meeting pending at room *(Room1)*, and the agent should also be at another location but not already engaged in a meeting. The precondition of *AttendingMeeting(Room1)* is represented as:
{ *meeting(Room1), position(Room2), NOT in_a_meeting(self)* }

After performing the action, the post-condition will become true, and the agent's beliefs will be updated. For example, after the *AttendingMeeting* action took place, the following beliefs will hold:
{ *in_a_meeting(self), position(Room1)* }

[4] 3APL stands for An Abstract Agent Programming Language, and is pronounced "triple A P L".

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 test goal allows the agent to evaluate its beliefs (a test goal checks whether a belief formula is true of false). For example, a test goal for testing if the agent is carrying a box looks like *carrybox(self)?* 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 are introduced in the next subsection). From these three types of basic goals, we can construct composite goals by using the sequence operator, the conditional choice operator and the 'while' operator. 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. Each method is assumed to return a list (possibly empty) of well formed formulae.

2.4 Practical Reasoning Rules

Practical reasoning rules are at the heart of the way 3PAL agents operate. They 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). To allow for the dynamic matching of rules, goal variables are used as place-holders. Unification mechanisms are used when performing goal-matching.

3 Rule-driven Object-oriented Knowledgebase 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 [11]. 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 psuedo-Java scripts. But there is a performance trade-off: native mode is faster in execution and is the optimal method of operation.

3.1 An Example ROK Program

Here we present a simple ROK program as an example. The rules in this program say that "If a salesman is selling a product the customer needs, for a price the customer can afford, then the deal is made". Supposing that Salesman, Customer and Product are Java classes, previously defined by the programmer (or even by third-parties), the rule should be stated as the following:

```
import example.Salesman;
import example.Customer;
import example.Product;
rule: trade {
  declarations:
    Salesman s;
    Customer c;
    Product p;
  conditions:
    c.needs(p);
    s.owns(p);
    s.priceAskedFor(p) <= c.getMoney();
  actions:
    s.sell(p);
    c.buy(p);
}
```

In this example, if there is an object for each of the Salesman, Customer and Product classes, and all the expressions in the condition part evaluate to be true, then the action part of the rule will be executed. The formal syntax description of ROK and more examples can be found in [11].

3.2 Internal Structure of ROK

The heart of the ROK system is the knowledge-base. It is composed primarily of three main 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.

The object-base is simply a collection of objects. It could be simply implemented as a Java vector, but we decided to store the objects in a structure from where we could retrieve the objects of a given class in a more efficient way. Hence, the object base is implemented with a hashtable that maps fully-qualified

names of the classes to the set of objects belonging to that class. With that arrangement, we can efficiently retrieve all objects that belong to a given class, which is a necessary operation in the matching stage of the inference engine. The object-base is also responsible for storing the inheritance relationships between the class of the objects stored in it, so that when the inference engine asks for all objects of some class, it will return both the direct instances of this class and the instances of its subclasses (i.e., its indirect instances).

RETE is a classical algorithm used in production systems to minimize the number of tests required in the matching process [6]. Partial matchings are stored in a RETE, and they do not have to be re-tested. New objects that arrive in the network are tested only where necessary.

The conflict set of the knowledge-base is the area where rules ready to be fired are stored. The user has the ability of choosing the conflict resolution policy to be used in the knowledge-base. In most of the cases, the user will not need to use any complex policy, and the predefined classes will be sufficient. ROK has some predefined classes that implement different policies for choosing which rule is to be fired at any given moment. The predefined classes are the following:

1. DefaultConflictSet: The conflict set used when none is specified. Its conflict resolution policy is not specified. In other words, any of the instantiations can be returned, and it was implemented to be as efficient as possible.
2. LRUConflictSet: A conflict set that will choose the least recently used rule. If there is more than one rule in the conflict set, it will choose one that was fired before the remaining ones.
3. MRUConflictSet: A conflict set that will choose the most recently used rule. If there is more than one rule in the conflict set, it will choose one that was fired after the remaining ones.
4. NaturalConflictSet: A conflict set that will not allow a rule to be fired more than once with the same objects. This conflict set requires a large amount of memory to store the history of rule firing, so it must be used with care. It also tends to get inefficient when the history grows.
5. OneShotConflictSet: A conflict set that will not allow a rule to be fired more than once.
6. PriorityConflictSet: A conflict set that will give priorities to the rules. Rules defined first in the rule base file will have higher priorities than rules defined later.

3.3 The Reasoning

The current implementation of ROK enables the user to operate in two modes on the knowledge-base: one-shot mode and continuous mode. When the user calls the run() method on the knowledge-base, the inference engine is triggered to operate. It will perform reasoning on the objects of its working memory until the conflict set is empty. Then it will return the control (return from the method call). Another possible method is to call runInLoop() method. This method will block the current thread and perform reasoning of the knowledge-base continuously, i.e. until the halt() method is called. In the continuous mode,

the reasoning will not stop when the conflict set is empty, but will be triggered on every change of the knowledge-base state, i.e. on addition/removal of rules or facts. To get information from the knowledge-base, the agent (user) can use the objects() methods to retrieve all the objects of a given class that are stored there. Another way of retrieving the information gathered during the execution of the run() method is to store the information needed in the internal state of some fact object.

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 [9]. 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 a database 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 uses a framework-like model 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 [11].

5 Hierachical Agent Architecture Using Micro-Agents

Before we discuss how the reasoning engines are incorpoated into OPAL, it is necessary to describe the system architecture. The notion of agency is used at all abstraction levels in modelling OPAL agent systems. 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 [13].

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.

Because the smallest building block in OPAL is an agent, the system designer can apply agent-oriented modelling approaches 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 enviroment 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 the role. When the set of services of the role are subsequently requested, it can request its group members to provide the services for him. 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 get rid of some not frequently used group members. Also, an agent may decide to deregister 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 sementics level. 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 for it, 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 of the systems.

A second approach, which is the one we are currently taking, relies on using 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 realized. 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 exchanging. 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 provide 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 succeed in achieving the goal or not. Similar to the role-matching approach, the goals can be recursively passed down through the agent hierarchy, or even from the bottom-up. The advantage of using goals instead of roles becomes evident when the system designer can only describe the states of the system declaratively but doesn 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. It allows more dynamic interactions among agents. For example, if the semantics of the pre and post-conditions of the declared goals is commonly understood among agents, an agent can creating a new goal on the spot.

This hierachical structure of agents allows us to construct more complex agents. And since the agents, even at the low-est level, are completely autonomous, not only systems that operate in dynamic enviroments can be modelled using this architecture, we can even model intrinsically dynamic systems that are changing or evolving over time.

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 compiles the source using the interpreter. Recall that in Section 2.3, we mentioned a special kind of goal in 3APL language called JavaGoal, which represents a simple Java method invocation. In our implementation, we have modified the JavaGoal construct so that when the 3APL program produces a JavaGoal, an OPAL goal is created to wrap up the contents of the JavaGoal. The 3APL micro agent can then treat it as a normal OPAL goal and decide whether it has the ability to solve this goal using its local capability or not; and in the latter case, it can distribute this goal to other micro agents for assistance. When external information arrives at the 3APL micro agent, whether it is a message or a goal, the 3APL micro agent will insert the information into its belief-base, in the following format:

- if the arriving data is a message, the belief *message(content)* will be added to the belief-base.
- if the arriving data is a goal(which represents a service request), the belief *goal(precondition, postcondition)* will be added to the belief-base.

In both cases, the belief-base is treated as a knowledge-base for holding information. It would be more straightforward to insert the incoming OPAL goal as a 3APL goal instead of inserting it into the belief-base as a belief item. But the problem is that there is a set of eight programming constructs for 3APL goals (BactionGoal, PredGoal, TestGoal, SkipGoal, SequenceGoal, IfGoal, WhileGoal, JavaGoal), and in order to do the goal transformation, one is faced with the problem of interpreting the content of the OPAL goal, and deciding which one of the eight types of 3APL goal to transform to. 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. This implementation restricts the dynamism and flexibility we gained from having declarative OPAL goals, because the 3APL programmer needs to know what kind of OPAL goal the program will receive in order to prepare sensible rules for it. But on the other hand, even with 3APL goals, true dynamism is not possible. The 3APL programmer can only specify the plan of achieving goals knowing what goals to expect. In this sense, this implementation compromise is not too severe. Nevertheless, to provide better bridge of this existing semantic gap remains as a future goal for OPAL development.

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, respectingly.

7 Performance Comparison of the Three Reasoning Engines

The absolute speeds 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 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 and are plotted in Figure 1 and Figure 2. When the length of the array to be

Table 1. Mergesort Time (in ms.)

No. of int to sort	3APL	ROK Native	ROK Scripting	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

sorted became larger than 1000, the 3APL program took too long (longer than 15 minutes) to compute and thus the results are omitted.

We plotted the results of the 3APL program separately from the other results, because the scale of the 3APL program's execution time is too large for visual comparison with the other reasoning engines. From Figure 2 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 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 ROK or JPRS. The slower speed of 3APL is a tradeoff against its declarative expressiveness, and its first order logic features.

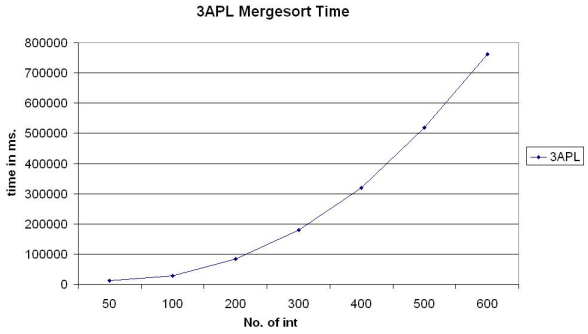

Fig. 1. 3APL Mergesort Program Runtime Plot

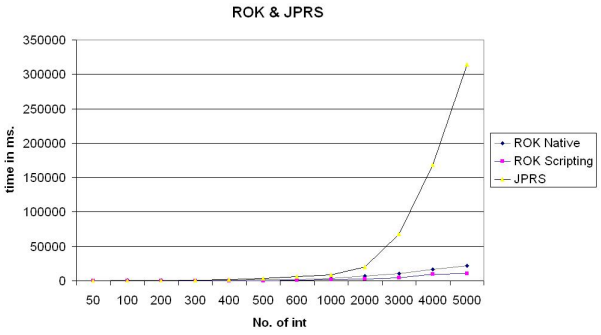

Fig. 2. ROK and JPRS Mergesort Program Runtime Plot

8 The Master Mind^{TM}Games

We developed a system for benchmarking, the Master Mind^{TM}Game[14], 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 [14]. 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 exchanging FIPA messages. The real-world Master Mind game is not a multi-player game and therefore adopting it and setting it into a multi-agent scenario is not naturally 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 systems 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 faciliates 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 association 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, 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 classloaders. This is the next phase of OPAL development. The screenshots of the IDE are shown in Figure 3.

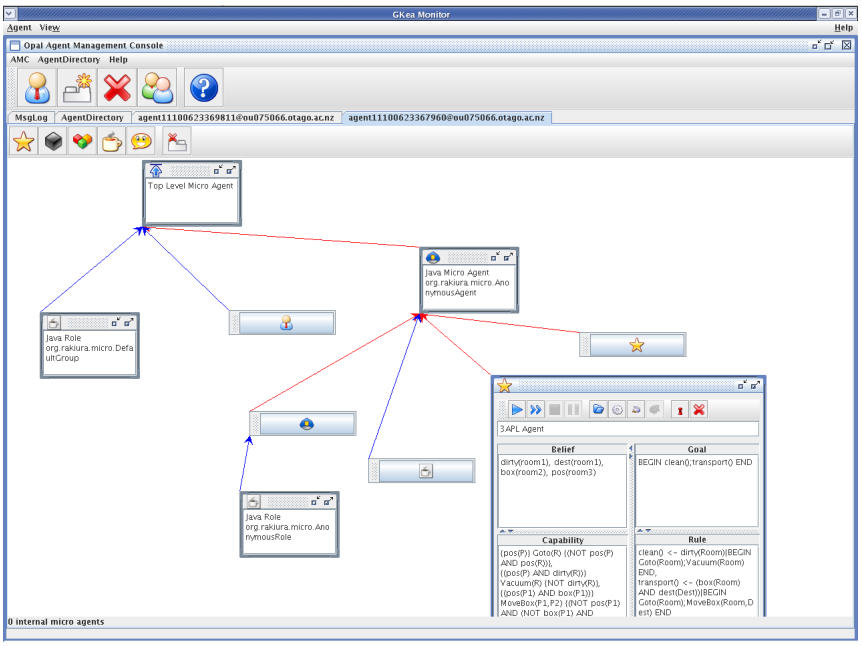

Fig. 3. OPAL Agent Composition Panel Screenshot

10 Conclusion

As discussed at the beginning of this paper, declarative agent programming languages and techniques bridge the semantic gap that exists between agent specification and practical implementation. 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 faciliate the easy 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 maintainingt-oriented architecture.

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. Hindriks, K., De Boer, F., Van der Hoek, W. and Meyer, J.J. (1999). "Agent programming in 3APL", *Autonomous Agents and Multi-Agent Systems*, 2(4): pp.357-401, 1999.
8. Hoeve, E.C.ten (2003). "3APL Platform", *Master's Thesis, University of Utrecht, The Netherlands*, Oct 2003.
9. Huber, M.J. (2001). "JAM agents in a nutshell", Nov 2001.
10. *Java Agent Service.* `http://www.java-agent.org`
11. Nowostawski, M. (2001). "Kea Enterprise Agents Documentation", Aug, 2001.
12. Nowostawski, M. (2004). "Otago Agent Platform Developer's Guide", Feb 2004.
13. 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.
14. Nelson, T. (1999). "Investigations into the Master Mind^{TM}Board Game", `http://www.tnelson.demon.co.uk/mastermind/`, 1999.
15. Rao, A.S. (1996). "AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language", In W. van der Velde and J.W. Perram, editors, *Agents Beaking Away (LNAI 1038)*, pp.42-55, Springer-Verlag, 1996.
16. Ricordel, P.M., and Demazeau, Y. (2000). "From Analysis to Deployment: A Multiagent Platform Survey", *ESAW*: pp.93-105, 2000.
17. Shoham, Y. (1993). "Agent-oriented Programming", *Artificial Intelligence*, 60: pp.51-92, 1993.
18. Wooldridge, M.J., and Jennings, N.R. (1995). "Intelligent agents: Theory and practice", *Knowledge Engineering Review*, 10(2), 1995.
19. Wooldridge, M.J. (2002). *An Introduction to MultiAgent Systems*, West Sussex, England: Wiley, 2002.