A Multiple Rule Engine-Based Agent Control Architecture

Edward P. Katz
Software Technology Laboratory
Hewlett Packard Laboratories
Hewlett Packard Company
1501 Page Mill Road, M/S 1U-14
Palo Alto, California 94304 USA
650.857.8725
ed_katz@hpl.hp.com

ABSTRACT

This paper introduces a multiple rule engine agent architecture intended for prototyping heterogeneous agents. It also briefly describes how this same architecture might be used for concurrent and hierarchical goal processing. The proposed framework has an embedded *main* rule engine supporting goal processing as well as providing high level decision and control for the agent. It also possesses a mechanism for dynamically spawning rule engine-based agent behaviours (using separate rule engines) to process subgoals, and it maintains information/status communication channels between the *main* rule engine and the rule-based agent behaviours. The agent together with its agent behaviours are all realized in a similar manner. Each embedded rule engine has the ability to configure multiple agent behaviours either in parallel or hierarchically. In addition, this approach allows the construction of reuseable, rule-based agent capabilities that can easily be composed and layered to achieve a specific, application-oriented composite capability.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence-*Coherence and coordination, Intelligent agents.*

General Terms

Design.

Keywords

Rule-based agents, goal-directed processing, multiple rule engine-based agents.

1. INTRODUCTION

A rule-based, agent control architecture provides several desirable properties. Such an agent could have an embedded expert system with a human expert's knowledge encoded in its rules to aid in the high level decision making and control. But an intelligent agent needs more capabilities than just the encoding of an expert's knowledge. It is important for an agent to also be able to process composite goals (e.g. a goal tree hierarchy) in which parallel subgoals may be actively processed concurrently. It would be desirable for a rule-based agent architecture to be able to spawn concurrent subgoal processing while at the same time maintaining an agenda of concurrent active goals that the agent is pursuing.

This work is motivated by several problems. The first problem is need to define an agent architecture in which a system consisting of many heterogeneous agents can be quickly prototyped. Each agent possesses its own specializations and capabilities which may or may not be duplicated and can cooperate and negotiate in an apparently intelligent manner. The second problem is the need to easily represent concurrent goal-directed task processing such that the agent can multitask goals on its agenda. Finally, there is the implementation of goals that are themselves hierarchical goal trees (where each node represents a subgoal realized by a goal-directed task).

We introduce a multiple rule engine agent architecture intended for prototyping heterogeneous agents and briefly describe how this same architecture might be used for concurrent and hierarchical goal processing. In addition, this approach allows the construction of reuseable, rule-based agent capabilities in a modular fashion such that they can easily be composed and layered to achieve a specific, application-oriented composite capability. This is not limited to just simply coupling modules. Pre-existing modules can be coupled and the composed capability may be tailored to the particular application. Another dimension of this approach is the ability to layer capabilities in which successive layers provide more sophisticated, rule-based capabilities and knowledge representation. While there have been previous rule-based agent systems developed, the effort described here appears to go beyond the typical approach.

This paper's organization consists of a brief description of the problem being addressed (section 2), a model solution to address the problem (section 3), a proposal for a multiple rule-engine architecture with compositional behaviours as a candidate problem solution based upon this model (sections 4 and 5), an overview of how this architecture might be implemented in an existing agent system (section 6), and a brief discussion of how this work differs from related work (section 7).

2. PROBLEM

When initially developing new agent-based systems, there are often specific needs which if they can be addressed will facilitate the development effort. In particular, the problem here is to address some of these specific development needs:

- **Rapid agent prototyping.** Foremost is ease of prototyping multiple, heterogeneous agents used to construct agent-based applications where each has its own specialization and capabilities.
- **Behaviour reuse.** Behaviour reuse is needed not only to facilitate prototyping but also to allow modular construction of new composite behaviours for the different agent instances.
- **Goal directed-task processing.** Goal-directed task processing among multiple concurrent goals is essential for the particular agent systems of interest to us. In this paper, the term *goal* is used to mean a goal-directed task in which there is a goal is to be achieved and there are already known methods for satisfying the goal which may be hierarchical.

3. A MODEL FOR AGENT STRUCTURE

Our model integrates several distinct ideas:

- **Agent *Main* behaviour**. An agent performs actions using the agent's *behaviours*. The agent must have a persistent *main* behaviour.
- **Agent *personality.*** The agent's *main* behaviour contains an embedded rule-engine implementing the agent's rule-based *personality.*
- ***Auxiliary* behaviours.** An agent may in addition have any number of application-oriented, *auxiliary* behaviours. These behaviours can be dynamically constructed, executed, and deleted as needed.
- **Auxiliary behaviour personality**. Similar to the *main* behaviour's personality is the *auxiliary* behaviour's rule-based personality. Each behaviour has its own separate, distinct rule-engine.
- **Encapsulated behaviours.** All behaviours run within the agent and cannot be accessed nor influenced from the outside.
- **Personality composition.** Behaviour *personalities* may be composed of simpler, reusable rule-based *personalities*. Composing several *personalities* together with specific customization can produce an individual agent's *composite personality* (that may or may not be unique) executing in its *main* behaviour. Likewise, the agent's *auxiliary* behaviours may also have similarly composed *personalities*. In either case, a layered composition can build up sophisticated composite action capabilities.

4. A RULE AGENT ARCHITECTURE

This model forms the basis of the *RAgent* (**R**ule **Agent**) architecture which uses an embedded rule engine for an agent's primary control and decision making.

4.1 RAgent Shell

The RAgent architecture consists of an agent containing a persistent *main* rule-based behaviour. All RAgent-type agents have this *shell.* The only difference among them is the rule sets that they are processing in their own rule engines. The *main* behaviour's purpose is two-fold. First, it accepts all incoming messages delivered to the agent. An incoming message is asserted into the agent's rule engine knowledge base as an incoming message fact (i.e. a fact whose content is the actual message). Second, this behaviour is the main repository and execution engine where the agent's primary control, decision making, and knowledge base functions occur.

4.2 Embedded Rule Engine Personalities

Our approach uses the concept of a *personality* (a cluster of rules, functions, data structures, and global variables) intended to provide a category of functional abilities. A *personality* may be thought of as either a reusable application package or a layer in a control hierarchy. In either case, it is a conceptual package implementing some agent capability. In the layered approach (Figure 1), each successive layer raises the aggregate behaviour capabilities to higher sophistication. This may be accomplished in one or both of two ways. First, a new personality may simply augment the agent's currently accumulated extant personalities. The second way is by adding a new personality layer which has one or more rules and functions that simply name replace (supersede) lower layer personality rules and functions. Layering allows agents of different sophistication levels to be easily constructed from reusable, rule-based capabilities of various personality packages. (The obvious assumption made here is that no conflicts exist using this *mix-and-match* approach. Care must be taken to avoid any unwanted or undesirable conflicts.)

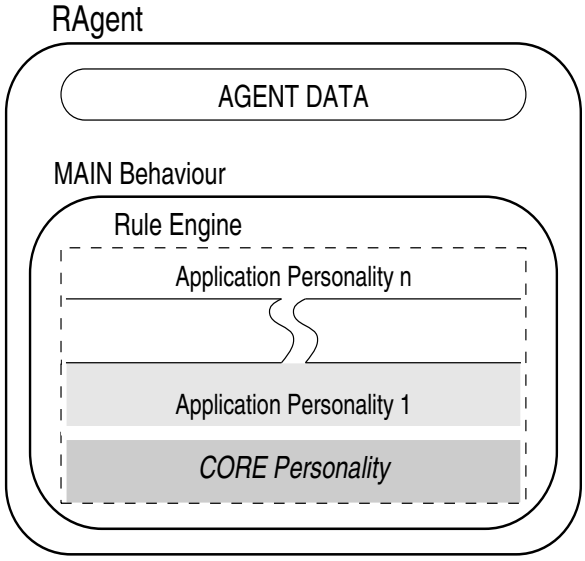

Figure 1. RAgent Architecture

Whenever an RAgent is instantiated, it is automatically imbued with a basic functional layer called the *core personality.* The *core* contains the fundamental agent default capabilities which can be

replaced or augmented by adding higher level application personality layers. In the above example where an incoming message is asserted into the rule engine fact base by the RAgent's main behaviour, if there is no extant application personality rule whose antecedent pattern matches the message fact (e.g. an unexpected message type), then the unrecognized message fact is *caught* by a default *core* rule, generating an appropriate graceful response.

4.3 RAgent Architecture Benefits

The RAgent architecture approach has several distinct advantages. The ability to program (or reprogram) agents by constructing rules brings the agent system developer a higher level of expressiveness over conventional procedural programming languages. This facilitates building agent systems more quickly by reducing the edit-compile-debug loop time. Re-using the same RAgent architecture in which each instance shell is identical except for its own individual rule-base (and hence its own *personality*) allows more rapid development of functionally heterogeneous agent systems. When an RAgent is instantiated, a construction script argument is passed describing how to build up the particular agent's *composite personality.* This script specifies what extant personality modules to use, the proper order, and any personality layering. Thus, creating several different RAgent-type agents involves multiple RAgent instantiations, each having a (not-necessarily unique) personality determined by its own construction script argument.

Another benefit of using this RAgent approach is developmental flexibility. The RAgent *core* personality is to be constructed to accept special unique messages which can only be sent from the developer. Suppose for example, the developer needs to "tweak" some rules in a particular application personality of a particular active RAgent instance. The editing can be done off-line while the specific RAgent is still running. After the editing is complete, the developer sends to the specific RAgent-type agent a special reload personality message. Upon validating the message, the RAgent's *core personality* dynamically reloads the newly modified application personality. Thus the RAgent instance dynamically acquires a change in its functional capability while the agent system and the particular agent are still running (and does not disturb the rule engine fact base)

To illustrate this, suppose that three agents are used to play the card game of blackjack. Two agents are regular blackjack players while the third is the blackjack dealer (Figure 2). The two player agents are given the same *player* personality. The third agent is given the *dealer* personality which is a layered composition containing the same *player* personality plus a *dealer player* personality on the next higher layer. Basically the dealer agent is a special kind of player agent and can use some of the *player* personality capabilities, though they may need to be enhanced and refined. For example, the same basic blackjack game regulations for point counting (e.g. not going over 21) applies to both player and dealer. The *dealer* personality, however, has slightly different playing rules such as the *hit or hold on 17* rule based on particular style of blackjack being played (e.g. Las Vegas vs. Reno rules) that the regular player does not have. Additionally, the *dealer* personality has enhanced capabilities for dealing cards, collecting wagers, etc. that the player personality does not have.

To illustrate prototyping flexibility, the blackjack agent game developer while watching game play, may decide to modify the dealer's play by changing the dealer's *hit or hold on 17* rule. After making the rule change offline in the *dealer-player* personality, the developer sends the special reload personality message to the dealer agent to reload its complete personality. Upon message validation, the dealer's *core* personality immediately performs the reload including the modifications and continues play with the new dealer personality rules and functions.

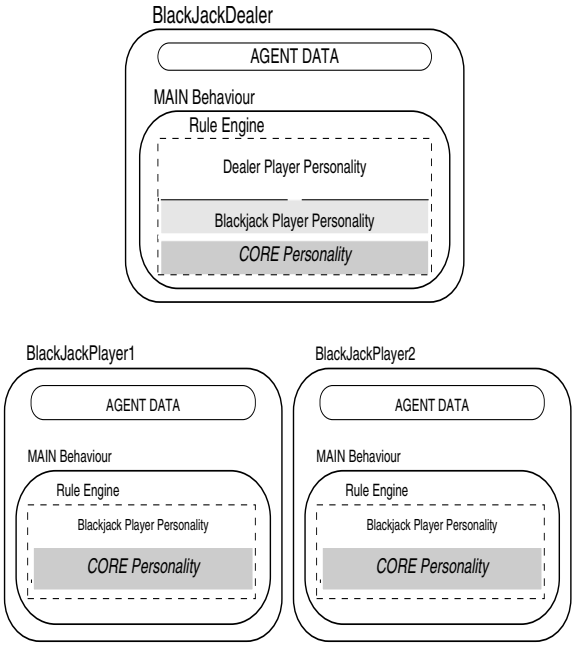

Figure 2. Agent Blackjack Game

5. RAgent++

As stated above, the RAgent architecture essentially describes an agent *shell* in which the agent functionality is represented by a knowledge base of rules, functions, and facts embedded in the agent's *main* behaviour rule engine. As demonstrated in the blackjack program example, the RAgent architecture appears to be sufficient for satisfying many of our needs.

We anticipate that concurrent, goal processing will be important for agent-based systems such as personal assistants and meeting agents [4] and thus, propose the RAgent++ architecture as a viable implementation framework. This concept of a shell with its own knowledge base (via an embedded rule engine) can be extended to the notion of an agent application-based, *auxiliary* behaviou*r* shell (Figure 3). When the agent has multiple concurrent goals (each may be hierarchical) that it is trying to satisfy, a variation of the shell concept can be applied. In like manner to the RAgent shell, a RAgent++ shell is a RAgent shell (and its associated *main* behaviour) capable of dynamically adding and deleting *auxiliary* behaviour shells.

5.1 RAgent++ Behaviour Shell

The RAgent++ *auxiliary* behavior's primary purpose is to contain a separate, distinct rule-based environment for the subgoal's task processing. Secondarily, it is to take advantage of the host system's subtasking facilities, if available. Goal-directed processing can be realized via spawning and executing a separate *auxiliary* behaviour for each subgoal node in the goal tree. Another case is an agent's need for concurrent task processing where each concurrent task is a separate *auxiliary* behaviour.

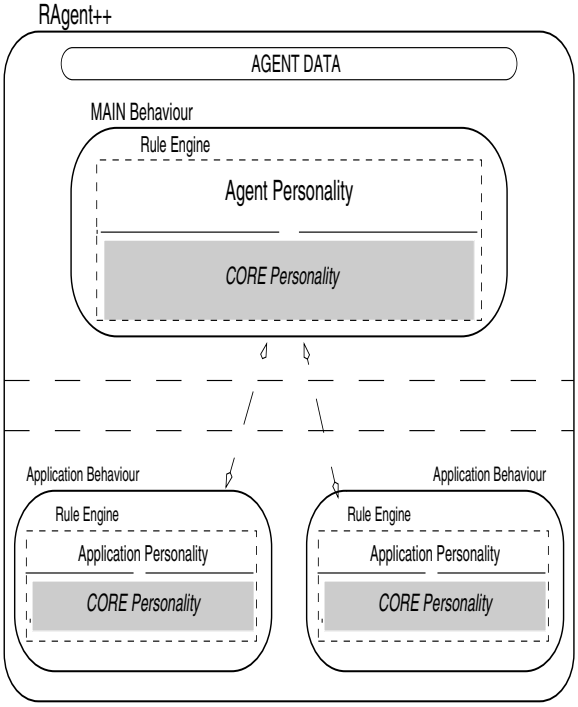

Figure 3. RAgent++ concurrent auxiliary behaviours.

The RAgent++ *auxiliary* behaviour shell consists of a behaviour instance together with its own embedded rule engine (completely separate from that of the RAgent++ *main* behaviour). Just as the RAgent++'s *main* rule engine has a possibly composite or layered personality realized via rules, functions, and facts, each *auxiliary* behaviour also has its own local personality including a special *auxiliary* behaviour *core personality* analogous to the RAgent++ *main core personality.* The essential functionality of the *auxiliary* behaviour *core* is to provide default behaviour and communications channels with the RAgent *main* rule engine, and other *auxiliary* behaviour rule engines. It is the case that the two rule engines have a limited degree of direct communications. (For this last reason, the RAgent++ *main core personality* is slightly more sophisticated than the RAgent version.) Each rule engine can assert or modify facts in the other's fact base. This allows the particular behaviour's rule engine to directly exchange information with the agent's *main* rule engine's fact base and with other behaviours' rule engines directly. This special access (based on privileges) directly into the other's knowledge base facilitates the transmission of status and synchronization control information.

RAgent++ *auxiliary* behaviours have a similar instantiation process to RAgent instantiation. Each time an RAgent++ *auxiliary* behaviour class is instantiated, a construction script is given as a parameter which describes how to construct that behaviour personality.

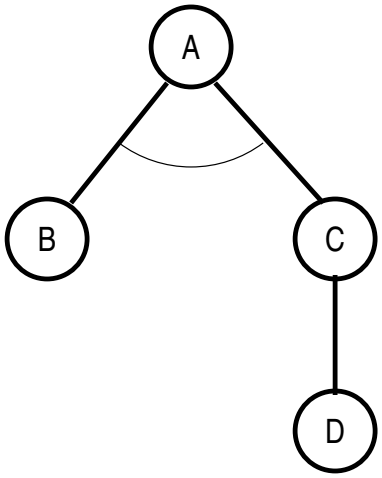

Figure 4. Goal Tree.

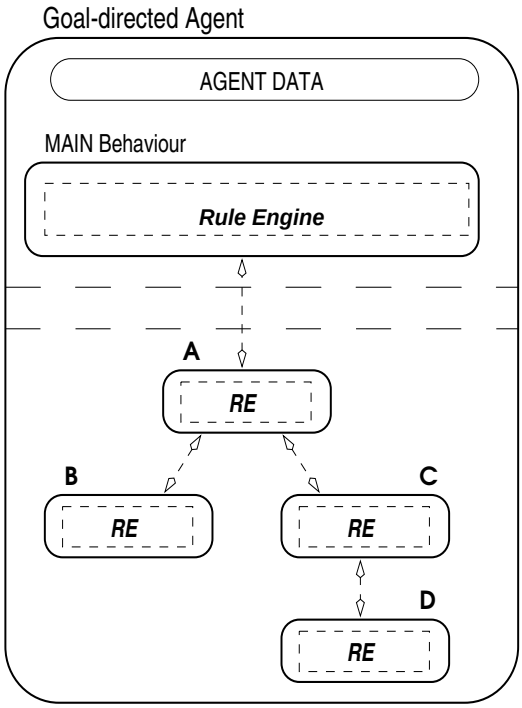

Figure 5. Goal-directed RAgent++ example.

With this *auxiliary* behaviour shell feature, the RAgent++ agent can dynamically spawn one or more rule-based, concurrent *auxiliary* behaviours. For the case where a goal tree contains multi-level, hierarchical subgoals, each subgoal at whatever level can be represented by a separately spawned *auxiliary* behaviour. Each

subgoal *auxiliary* behaviour has the responsibility to synchronize with other subgoal behaviours above and below it in the goal tree during processing. This can be especially useful for goal processing having a multilevel goal tree where each tree subgoal node is distinctly represented via a RAgent++ *auxiliary* behaviour.

Suppose the RAgent++ agent needs to achieve some goal **A** (figure 4) for which there already exists a task set (subgoals plus goal tree) encoded in rule-based form. Subgoals **B** and **C** are conjunctive, but are disjointedly independent and may be processed concurrently. Two concurrent *auxiliary* behaviours (figure 5) may be spawned for each subgoal processing. While **C** is executing, it spawns an *auxiliary* behaviour to process its subgoal **D**

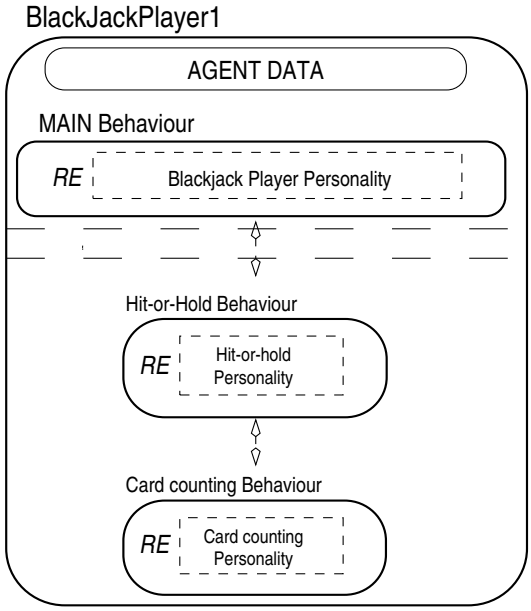

Figure 6. RAgent++ BlackJackPlayer1

If the BlackJackPlayer1 agent in the blackjack example above is reimplemented as a RAgent++ (Figure 6), it might be implemented as having a subgoal *auxiliary* behaviour deciding whether to *hit-or-hold* (accept another card from the dealer or stop). This subgoal might itself use a subgoal behaviour to keep counts for each card played that would affect game play probabilities. Likewise, if the BlackJackPlayer2 agent is similarly implemented and has been dealt two cards of identical rank from the BlackJackDealer agent, then BlackJackPlayer2 may elect to play the *split* hand. A split hand is actually two concurrent hands, each with one of the two identically ranked cards (e.g. *Jack* of Diamonds and *Jack* of Spades) and each hand with an additionally dealt card from the BlackJackDealer. The splitting can easily be represented in the agent via concurrent *auxiliary* behaviours (Figure 7)

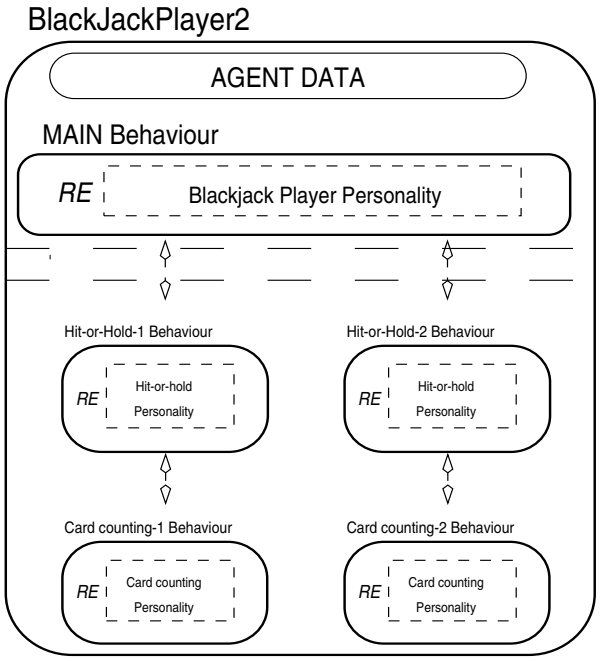

Figure 7. BlackJackPlayer2 playing split hand.

5.2 RAgent++ Benefits

In addition to the RAgent benefits, the major advantage introduced with RAgent++ is the ability to dynamically spawn rule-based *auxiliary* behaviours. Each *auxiliary* behaviour *core* personality knows how to exchange information and synchronize status directly with the RAgent's *main* rule engine and with the rule engines of other *auxiliary* behaviours belonging to the same agent instance. Each *auxiliary* behaviour can execute its own rule-based application independently and concurrently of the other tasks. This feature allows the agent to concurrently process goals representing the multiple activities in which the agent is presently engaged.

The RAgent *main* behaviour's rule engine could function as a passive communication medium such as a blackboard [13] where each sub-behaviour can have read and update access. Each *auxiliary* behaviour can view the blackboard and establish *daemon* rules to activate or spawn *auxiliary* behaviours as necessary. Also, the blackboard may also be used to display each behaviour's state and any changes in status. Alternatively, the main behaviour could act as an active overseeing controller requiring the sub-behaviours to synchronize their processing, possibly in a lockstep manner.

6. PROTOTYPING IN JADE

The feasibility of both the RAgent and RAgent++ architectures is being evaluated by prototyping in JADE [2]. It is a FIPA [5] compliant software framework for developing agent applications of inter-operable, intelligent, multi-agent systems. JADE is a kind of agent platform middleware and development framework supporting distributed, interoperable multi-agents together with the corresponding communications infrastructure including the FIPA Agent

Communication Language (ACL) [6]. Each agent belongs to a particular JADE platform which has infrastructure components for supporting multiple agents and ACL message passing. The platform's user GUI allows the user to see the platform agents and the graphical depiction of message exchanges. The developer can use the GUI to manually create an ACL message and send it to a particular agent (this feature will be used below).

6.1 RAgent Implementation in JADE

An agent in the JADE sense uses the JADE Agent abstraction and models an agent's tasks via the JADE Behaviour abstraction where each agent may dynamically instantiate its own behaviours as needed. Multiple behaviours belonging to the agent may execute concurrently using a round-robin, non-preemptive policy. The Java language is used to implement the agent infrastructure and base classes for agents and behaviours. However, there are no limitations for the kinds of software systems which can be embedded in the agents and behaviours.

In effect, a RAgent conceptually is an JADE Agent *shell* having an embedded rule engine. This is achieved by implementing each RAgent as a class extension of the JADE Agent class. A RAgent agent is paired with a *main* rule-based behaviour implemented as an instantiation of an extended JADE Behaviour class allowing a Java-based, embedded rule engine [7]). At RAgent class instantiation time in the JADE system, a parameter is passed to the class constructor which serves as the RAgent *personality* constructor script locator. The class constructor initializes the agent, instantiates the *main* behaviour which includes starting an embedded rule-engine, loading in the *main core* behaviour, and loading the agent *personality* as indicated by the constructor script.

As stated in section **4.3**, the RAgent *core personality* is constructed to accept special unique ACL messages which can only be sent from the developer using the JADE Remote Monitoring Agent (RMA) user interface GUI. A developer can instruct a particular RAgent instance to reload its *personality* by using the RMA GUI to send a reload personality ACL message while the specific agent in the JADE system is still running. Both the RAgent instance and the JADE system continue to run while this *personality* update occurs.

Analogous to the RAgent implementation above, a RAgent++ is implemented using the same process for the agent and *main* behaviour. Each *auxiliary* behaviour is an instantiation of an extended JADE behaviour class having its own embedded rule engine. During the instantiation, the construction script parameter specifies the application-based *personality* to be loaded into the rule engine's knowledge base.

6.2 JADE Behaviour Usage

Consistent with the JADE model of agents, an instantiated RAgent (or RAgent++) agent may use one or more standard JADE behaviours written in Java. (This is to leverage reuse of previously written, conventional JADE behaviours.) The RAgent and main behaviour have the capability (just as regular JADE agents) to dynamically spawn, control, and discard conventional JADE behaviours. Both the RAgent and main behaviour classes allow method access from the JADE behaviours to update the RAgent's knowledge base for process control and execution synchronization and information exchange.

As of this writing, a simple proof-of-concept personal assistant-based meeting arranger using this architectural framework is being constructed for demonstration using JADE.

7. RELATED WORK

Rule-based agent construction has been used for some time in general agent programming. Typical examples [10][9][3] involve a rule-based system programmed to give the agent some narrow "intelligent" appearing capabilities implemented in the form of rules.

Another application [15] uses rules for the dynamic generation and editing for specific kinds of agent programming, e.g. personal agents programming by the user. Because of the expressive power for problem description in a rule-based system, this can be used for end-user programming of personal agents to express goals, habits, and preferences. This does require a sophisticated system to validate user input to resolve with the user any conflicts between the user's rule.

A rule-based approach can also be used as a method of agent learning. If a rule-based agent could continually acquire new rules, then conceivably it could expand its knowledge base and capabilities over time. Machine learning processes that can generate new rules allows rule-base agents to incrementally acquire knowledge and programming from the environment and experiences [12][3].

Various other agent applications use rule-based subsystems such as in agent conversation management [1], in reasoning tools for intelligent applications such as control in virtual worlds [14], in automated citation finding [11], and in business rules applications [8] just to name a few.

These are token representatives for some rule-based agent applications which tend to use rules as a means of bringing some form of intelligent processing to a particular application domain. This author is unaware of any other similar work that uses the concept of hierarchically layered, application-based personalities for agent control. In addition, this author is unaware of any similar work using the concept of multiple rule engines to implement concurrent agent behaviours. The effort described here goes beyond these previous approaches.

8. CONCLUSIONS

The RAgent architecture uses personalities consisting of rules and functions to create agent capabilities. Higher level *composite* personalities (layering) can be built up by first installing the lower *personality* layers and then incrementally adding additional layers to augment or replace the lower level rules and functions. Thus, a particular RAgent instance can be customized to have the desired decision/control sophistication using previously defined personalities with a *core personality* providing a default base agent functionality. Rapid prototyping is a major advantage of using the RAgent shell that allows quick instantiation of multiple heteroge-

neous agents. This is accomplished via multiple RAgent-type agent instantiations, each with its own construction script directing the building of its personality composition and layering. An auxiliary benefit is the ability to reload personalities dynamically without needing to restart the agent platform or agent instances thus allowing dynamic functional redefinition for developmental debugging purposes.

RAgent++ extends the RAgent capabilities by allowing the dynamic spawning of agent *auxiliary* behaviours with their own *personalities* which could interact with the RAgent++ *main* behaviour.

The major ideas presented include the rule-based construction of an RAgent-type agent's *main* behaviour, the similarly constructed rule-based agent *auxiliary* behaviours, and the introduction of reusable rule-based *personalities*. Each agent has an individual agent *personality* (that may or may not be unique) in its *main* behaviour. Likewise, the agent's application-based *auxiliary* behaviours have similar rule-based personalities. Both the *main* and *auxiliary* personalities may be a layered composition building up sophisticated capabilities

9. Acknowledgements

This investigation effort has benefitted greatly from discussions with Reed Letsinger, Martin Griss, and Richard Cowan and editorial assistance from Harumi Kuno.

10. References

[1] Barbuceanu, Mihai, Fox, Mark S. Integrating Communicative Action, Conversations and Decision Theory to Coordinate Agents. Proc. of the First Intl. Conf. on Autonomous Agents, (Marina del Rey, February 1997), 49-58.

[2] Bellifemine, Fabio, Poggi, Agostino, and Rimassa, Giovanni. JADE--A FIPA-Compliant Agent Framework. 4th Intl. Conf. and Exhib. on The Practical Application of Intelligent Agents and Multi-Agents(London, UK, April 1999), 97-108.

[3] Bigus, Joseph P. The Agent Building and Learning Environment. Proc. 4th Int'l Conf. on Autonomous Agents 2000 ACM Press, 108-109.

[4] Chen, Hsinchun, Houston, A., Nunamaker, J., and Yen, J. Toward Intelligent Meeting Agents. Computer, IEEE Comput. Soc, vol.29, no.8, Aug. 1996, 62-70.

[5] Foundation for Intelligent Physical Agents (FIPA). http://www.fipa.org.

[6] FIPA ACL Message Structure Specification. http://www.fipa.org/repository/aclspecs.html

[7] Friedman-Hill, E.J. Jess, the Java Expert System Shell. Sandia National Laboratories. http://herzberg.ca.sandia.gov/jess.

[8] Grosof, Benjamin N., Labrou, Yannis, Chan, Hoi Y. A Declarative Approach to Business Rules in Contracts. Proc. of the first ACM conference on Electronic commerce, (Denver, Colorado, November 1999), 68-77.

[9] Grosof, Benjamin N., Levine, David W., Chan, Hoi Y., Parris Colin J., and Auerbach Joshua S. Reusable Architecture for Embedding Rule-based Intelligence in Information Agents. Proc. of the Workshop on Intelligent Information Agents, ACM Conf. on Information and Knowledge Management (CIKM-95), Dec. 1995. (Also available as IBM Research Report RC 20305 (Dec. 05, 1995).)

[10] Hindriks, K.V., de Boer, F.S., van der Hoek, W., and Meyer, John-Jule C. Proc. 5th Intl. Workshop on Intelligent Agents V Agent Theories, Architectures, and Languages (ATAL-98), Springer-Verlag, 1999, 381-396.

[11] Loke, Seng Wai, Davison, Andrew, Sterling, Leon. CiFi: An Intelligent Agent for Citation Finding on the World-Wide Web. Proc. 4th Pacific Rim Intl. Conf. Artificial Intelligence (PRICAI), Springer-Verlag, 1996, 580-591.

[12] Nonas, E. and Poulovassilis, A. Optimising Self Adaptive Networks by Evolving Rule-Based Agents. Proc. Evolutionary Image Analysis, Signal Processing and Telecommunications. First Europe Workshops, EvoIASP'99 and EuroEcTel'99, Springer-Verlag, May 1999, 203-14.

[13] Pang, Grantham K. H., Development of a blackboard system for robot programming, Proc. of the Third Intl. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Vol. I, July 1990, 123-130.

[14] Schweiss, Elsa, Musse, Soraja Raupp, Garat, Fabien, Thalmann, Daniel, An Architecture to Guide Crowds Using a Rule-Based Behavior System, Proc. of the Third Annual Conf. on Autonomous Agents, ACM, April 1999, 334-335.

[15] Terveen, Loren G., Murray, La Tondra, Helping Users Program Their Personal Agents, Conf. Proc. on Human Factors in Computing Systems, ACM, April 1996, 355-361.