

# Rule-Driven Coordination Agents: A Self-Configurable Agent Architecture for Distributed Control

M. Brian Blake  
 Department of Computer Science  
 Georgetown University  
 Washington, DC 20057  
 blakeb@cs.georgetown.edu

## Abstract

The need for coordination among autonomous entities is common in dynamically changing domains. One such domain is the coordination of components in distributed component architectures. In these architectures, asynchronous communication is used to ensure autonomy. Management systems that control such architectures must respect this autonomy by decentralizing interaction policies and control. Middle agents [6] have been introduced as brokers or mediators in such dynamic settings. Rule-Driven Coordination (RDC) Agents are middle agents that act as brokers to the individual components in component architectures. These RDC agents encapsulate the interaction policy definition (rules) and the aspects of communication, data management, and policy execution. This paper defines each of these aforementioned aspects of the RDC agents. Furthermore, there is the use of RDC agents to manage a workflow of Java-based components in a typical electronic commerce domain.

## 1. Introduction

Distributed component architectures contain individual components that may be distributed across multiple machines. These components collectively perform individual tasks to achieve an overall goal. One such scenario is the implementation of an on-line workflow domain as in electronic commerce. Components can fulfill the individual steps in these architectures. A typical architecture for the on-line workflow domain is the on-line stock purchase workflow. In this workflow, distributed components can play the roles of customer interfacing, portfolio management, payment services, and trading services. When a customer makes a request for an on-line trade, these roles must coordinate to fulfill the request. In implementing this workflow, a customer interface component receives a request from an on-line customer. This component would forward the request to a portfolio management component. Finally a payment component would debit the customers account and a trader component would make the trade. The on-line

stock trade process is illustrated in Figure 1.1. The underlying components are shaded. These components can reside on multiple machines across the corporate Intranet and the Internet. In a traditional architecture, each of the components is connected to a workflow controller that captures events and contains the policy of interaction that achieves the overall job.

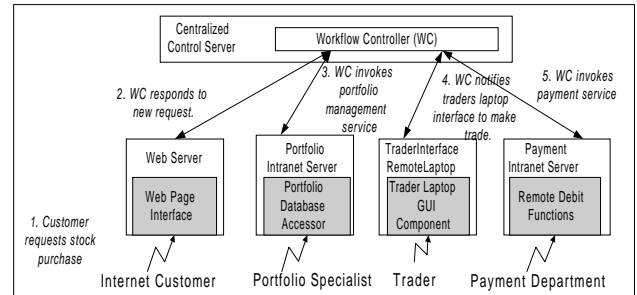


Figure 1.1. Distributed Component Architecture: On-line Stock Trade Process

RDC agents act as brokers for the individual components. Each RDC agent contains rules specific to the component(s) that it represents. This coordination of RDC agents is essentially a middleware layer between the distributed components that encapsulates the interaction policy (Figure 1.2)

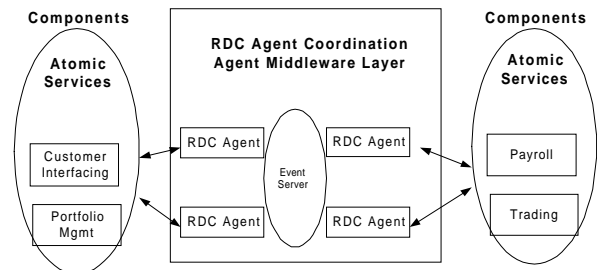


Figure 1.2 RDC Agent Coordination as Middleware

The focus of this paper is to describe the architecture of the RDC agents based on aspects of agent communication, data management, policy encapsulation

as rules, and policy execution. This paper follows in the next section with the overall architecture of the RDC agent. The subsequent sections describe each of the aforementioned aspects in detail. Finally, the RDC agents are described in context of a electronic commerce application, the on-line stock brokerage workflow.

## 2. Overview of RDC Agents

Currently, an RDC agent is composed of three main components. These components are policy execution, agent communication, and data management. Future work will include rule learning as a fourth functionality. At this point in our research, emphasis is placed on the architecture of the RDC agent. In defining a scaleable architecture, the RDC agents will be able to plug into a variety of autonomous domains. The design of the RDC agent is illustrated in Figure 2.1.

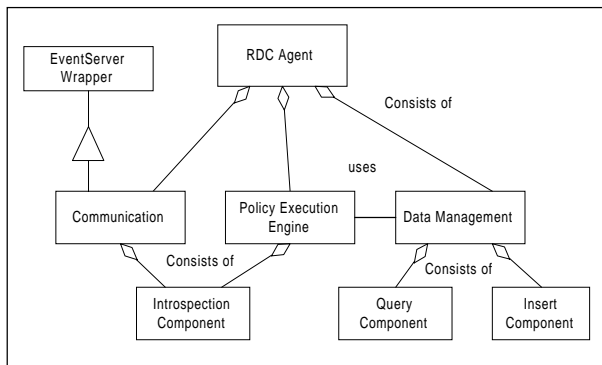


Figure 2.1 Aspects of the RDC Agent

The communication component wraps the API that connects to the centralized event server among the RDC agents. The data management component connects to a policy-oriented database. This database contains the policy information and is used as a repository to store the execution data. The Introspector component has the ability to dynamically transform data events into the software object format recognized by the event server. Finally, the policy execution engine uses the data management component to get policy information. This engine builds rules from the policy information that define the operation of the RDC agent.

## 3. Policy Execution and Data Management

RDC agents broker the execution of autonomous components as defined by an interaction policy. In Figure 1.1, that interaction policy is that of workflow. These rules follow the format of Event[Condition]Action, similar to that of UML activity diagram transitions[2],[5]. The focus in this paper is the not on the decomposition of the policy into distributed rules, but on the execution of the rules. RDC agents base their actions on either system

events or the events from other RDC agents. System events can be a result of timers, environment events, or even from system failures. RDC agents can fire events for the start or completion of an action. Actions performed by RDC agents are connected to real world functions. Agent actions can include the invocation of a service, initiating a component, or even storing information in a database. The condition is specific data that qualifies an event to a particular receiver. This idea of policy interaction can be further described in terms of a simple relational database schema as in Figure 3.1.

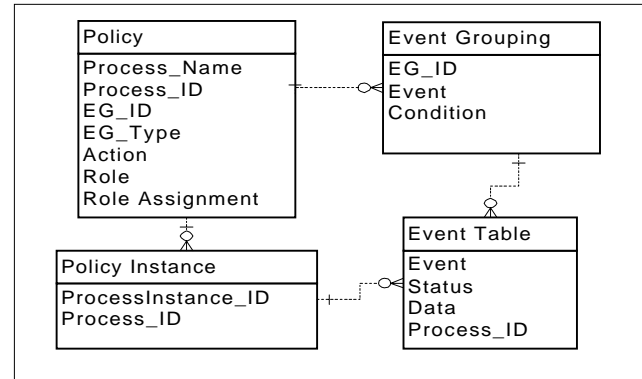
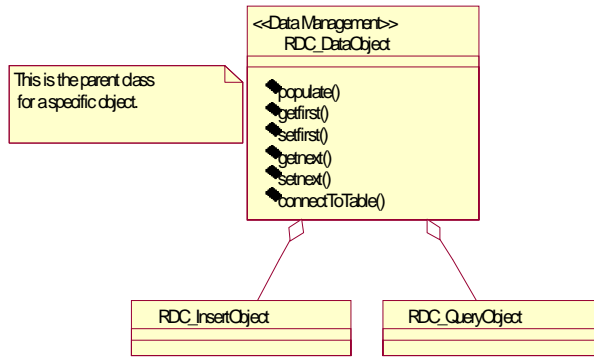


Figure 3.1 Relational Database Schema of Policy Information

The Policy Information table in Figure 3.1 contains the main information for handling events and actions. The reason that events (Event Grouping) are grouped in this table is to account for situations where multiple events are needed to precede an action. The EG\_Type field specifies additional qualifications on the grouping (i.e. AND/OR conditions, etc.). The Role and Role Assignment fields define which outside component or autonomous entity will be performing the action as the result of the event. The Policy Instance and Event Table act as persistent storage of the run-time operation. This relational illustration of policy information neglects the actual definition of the events and actions, but the intention is to show how the general information can be used for the coordination of RDC Agents. In later sections, the event and actions will be further defined for a specific domain. This relational definition can also be seen as the general ontology among the RDC agents.

The Data Management of the RDC agents is encapsulated in DataObjects. The RDC agents contain specialized data objects that broker information into and out of the policy-oriented database (these objects were integrated from [3]). Each table in the policy database has a corresponding specialization of the DataObject. Figure 3.2 illustrates a general class diagram of the DataObjects.

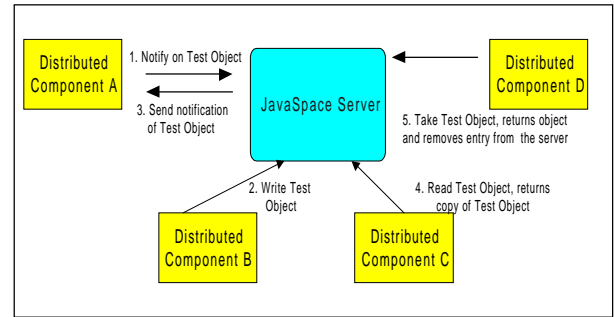


**Figure 3.2** Design of the Data Objects

When a DataObject represents a single table, the table name is supplied as a parameter. When the DataObject represents the result of a specific query, a QueryObject is instantiated and supplied as a parameter. Once a DataObject has been created, at any time, an InsertObject can be used to update the underlying table(s). These objects wrap the basic functions of Java’s ResultSet classes [12].

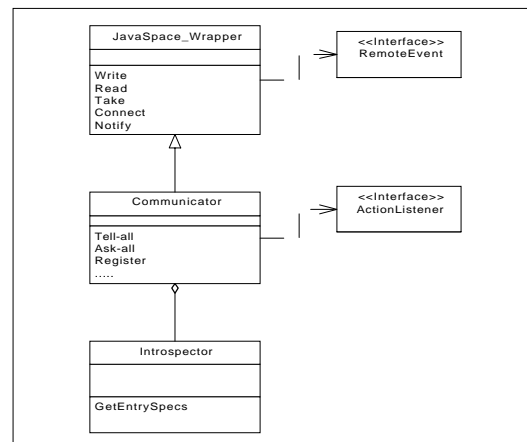
#### 4. Agent-based Communication

RDC Agents communicate through an event server. The implementation we use for the event server is the JavaSpace object server built on Jini technology. Jini is a suite of services developed by Sun Microsystems that provide a simple substrate for distributed computing [7]. Jini supports most common principles surrounding distributed coordination (i.e. remote objects, leasing, transactions, and distributed events). It is not in the scope of this paper to give an in-depth description of Jini but to describe those services that are used for agent communication, specifically JavaSpaces [8]. JavaSpace technology is based on “Tuple Spaces”. Tuple spaces, first introduced in the context of the “Linda” project in 1982, allows distributed software processes to communicate autonomously. The tuple space emulates a data storage server. The server receives entries from independent components and stores them for retrieval. Exterior components can be notified when an entry of a certain pattern or tuple is entered. Components can also read and take matching entries based on a tuple-based pattern they submit. Though JavaSpace technology was motivated by tuple space, it is slightly different. JavaSpaces are “object” storing services. It supports read, write, take, and notify on actual software objects. Sample JavaSpace interactions are illustrated in Figure 4.1. There is a set of classes that can be integrated with the RDC agents to assist in using the JavaSpace server. These classes are a subset of the overall architecture in section 2.



**Figure 4.1** Typical JavaSpace Functions

The objects specifically for agent communication are illustrated in Figure 4.2. This architecture was integrated from prior work in agent communication (KOJAC) [4].



**Figure 4.2** Agent Communication Classes

This architecture consists of a Communicator class that inherits functionality from a JavaSpace\_wrapper. The JavaSpace\_wrapper class implements all of the native JavaSpace commands. The Introspector class looks into the ontology-based package or database to construct entries used by the Communicator class. The Communicator class also brokers events between the JavaSpace server and the agents.

The agent communication sequence is illustrated in Figure 4.3. When a component completes a service, it fires a completion event. The completion event is captured by the Communicator. The RDC agent classifies the event as a completion. The Communicator invokes the Tell-all method. Within the Tell-all method the Introspector is instantiated. This Introspector searches the ontology-based package or database for an entry class that has the same name as the completed service. The introspected class is returned and the action field is populated as a completion. Finally, the inherited write function (from JavaSpace\_wrapper parent class) is called with introspected class as a parameter.

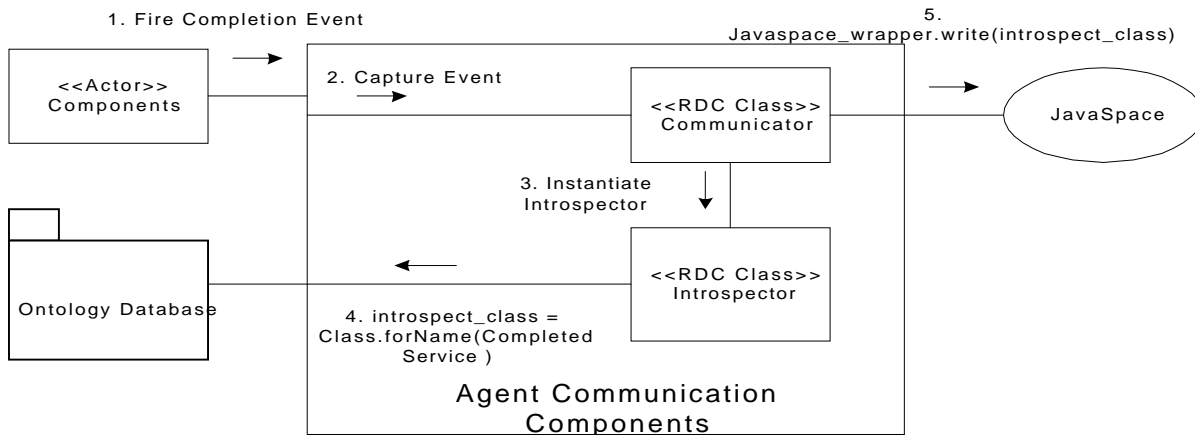


Figure 4.3 Communication Operational Flow

## 5. A Case for RDC Agents

The RDC Agents have been deployed in the workflow management domain. WARP (Workflow Automation through Agent-based Reflective Processes) is the first attempt to use the RDC agents in an operational environment. This section details the operational automation of the workflow coordination using RDC agents.

### 5.1 WARP Overview

This WARP architecture consists of software agents that can be configured to control the workflow operation of distributed services. The WARP architecture has an application coordination layer where workflow instances are instantiated and the actual workflow execution occurs. The application coordination layer consists of two agents, the Role Manager Agent (RMA) and the Workflow Manager Agent (WMA). The RMAs have knowledge of a specific workflow role. The WMA has knowledge of the workflow policy and applicable roles [10]. When a new process is configured the workflow policy is saved in a shared distributed database. The RMA plays a role in the workflow execution by fulfilling one or multiple services as defined by the workflow policy in the shared database. The RMA registers for pertinent events in the event server based on its predefined role. When an initiation event is written into the event server, the RMA is notified. Subsequently based on its localized knowledge of services and its workflow role, the RMA invokes the correct service. The WMA has similar functionality, but instead registers for workflow level events. The WMA does not control the workflow execution, but in some cases it adds events to bring about nonfunctional changes to the execution of the entire workflow. Both the RMA and WMA are specializations of the RDC agents. The WARP architecture is shown in Figure 5.1.1.

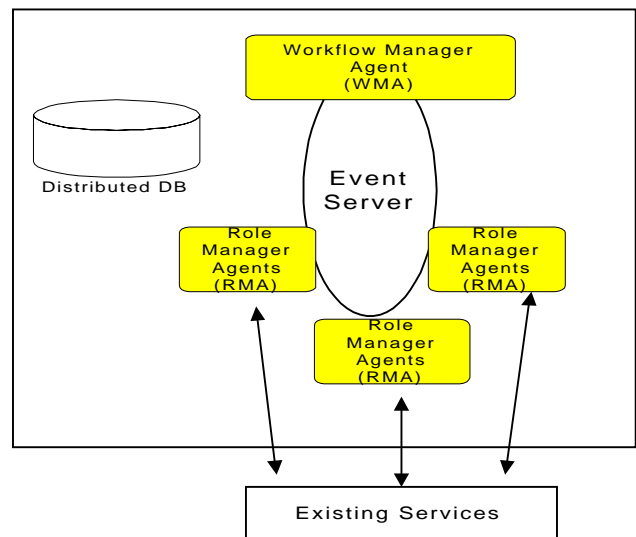


Figure 5.1.1 WARP Architecture

### 5.2 WARP Policy Database

The application layer agents manage the workflow operational environment. These agents rely heavily on a shared database to determine their actions in the operation of the workflow. The shared ontology among the RMAs and WMAs is based on the original database schema in Figure 3.1. For WARP, this schema is extended to handle the further definitions of events, conditions, and actions for workflow-oriented policy interactions. Both actions and events are further defined as invocation-based component services. Data is defined as the parameter and return data of the component-based invocations. This refined database schema is illustrated in Figure 5.2.1.

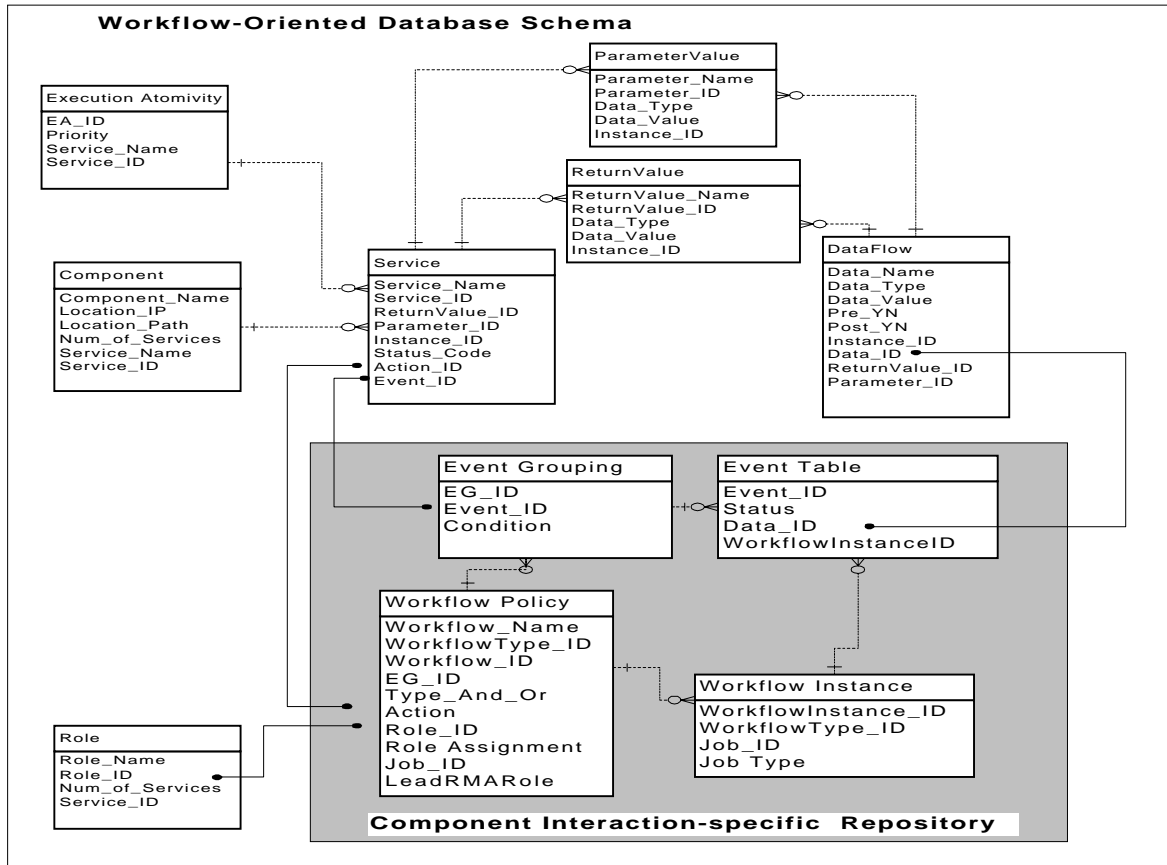


Figure 5.2.2 WARP Database Schema

### 5.3 WARP Self Configuration

The WARP architecture becomes operational once the RMA and WMA are instantiated and undergo a process of self-configuration. The self-configuration performed by the WMA can be summarized in Table 5.3.1.

WMA Self-configuration
1. Create database view for each role in the workflow composition table for the predefined process
2. Query the workflow composition table to find what nonfunctional concerns are associated with the predefined process.
3. Register for events that initiate the nonfunctional concerns
4. Register for events that initiate a workflow instance

Table 5.3.1 WMA Self Configuration Steps

The RMAs are only concerned with events, conditions, and actions as they correspond to a particular workflow instance. The RMA self-configures itself based on the views. This process is detailed in Table 5.3.2.

RMA Self-configuration
1. Query the role-based views to find the list of actions that will be performed by the predefined role
2. Establish connection to component-based services for each action to be performed.
3. Register for each event that initiates any of the above list of actions.

Table 5.3.2 RMA Self Configuration Steps

### 5.4 Operational Details

To consider the operational environment, again let's use the on-line stock-purchasing domain. A configured system would have a RMA for each of the roles. Let's consider the three aforementioned roles are the Customer Interface Role, the Portfolio Management Role, and Trading Role. There is a RMA for each role and there is one WMA that helps in the coordination of the entire workflow. Based on the Control Flow View, each role would place notify commands in the space for service

completion prior to their affiliated services. For example, the Portfolio Management Role would want a notification on the completion event of a `getTradeRequest` service. Suppose a customer invokes the `getTradeRequest` service. The RMA for the Customer Interface Role would insert the pertinent data for this service completion in the database and publish the service completion in the event

server. The RMA for the Portfolio Management Role would be notified of this completion. First it would check to see if this service is pertinent to any of its workflow instances. If the answer is yes, the RMA for the Portfolio Management Role would wait for the ready event to be written to the server by the WMA.

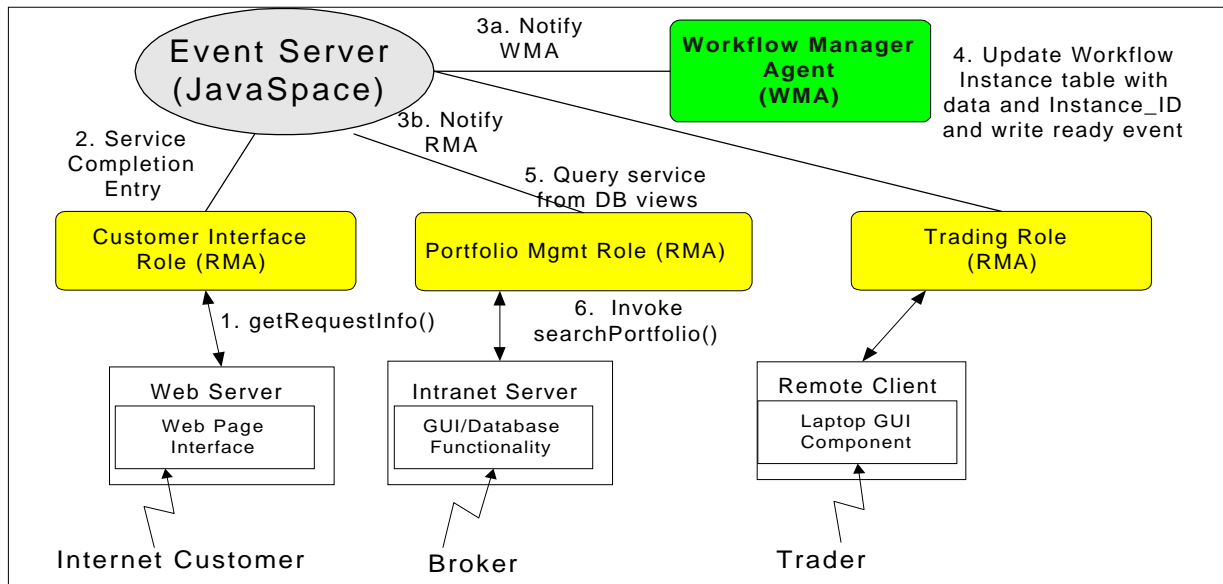


Figure 5.4 Operational Environment.

The WMA would have also been notified of the `getTradeRequest` service completion. The WMA would update the Workflow instance table with the new workflow process' instance ID. The WMA would submit a ready event to the event server upon completion. The RMA would invoke the proper service for this step in the workflow policy based on the action specified in its role-based view (`searchPortfolio` service). Subsequently the output data would be inserted in the database and the service completion event would be written to event service. This process sequence is shown in Figure 5.4 for a non-error case of the stock purchase process.

## 6. Summary

In this paper, a middleware of RDC agents has been defined as means for coordinating autonomous entities in domains that change dynamically. These RDC agents have been evaluated in the distributed workflow domain. We have found that the use of agents in these domains is consistent with need for autonomy. Encapsulating policy into agents prevents the terminal failures that occur when a centralized controller is used to manage these dynamic domains.

In future work, we plan to integrate RDC agents into simulation domains. One such domain is the air traffic management domain. There are several autonomous entities (i.e airlines, pilots, weather, FAA, etc.) that can control multiple variables in the successful coordination of air traffic for independent Flight Information Regions. In programming RDC agents with rules, adverse situations can be simulated and analyzed. These results can help air traffic officials make informed judgements in the future.

## 7. Related Work

There is other work in dynamic reconfiguration and runtime evolution. Shrivastava [11] devises a system for evolving workflow systems based on workflow scripts and task models. Using a CORBA-based support architecture, they accept workflow scripts as specifications for reconfiguration. This approach relies on a static set of workflow components. Shrivastava devises a system that takes a "top-down" approach. The system uses a set of CORBA-based services that can be dynamic in nature. Shrivastav devises a workflow script that specifies the functional nature of the workflow where nonfunctional constraints are bundled in. The WARP

approach separates functional and non-functional specifications. This separation allows for a greater ease of reuse for nonfunctional considerations. The WARP approach also distributes the execution of the workflow policy among the underlying services thus decentralizing the control. This maintains the autonomy that supports the addition of new services and the removal of deprecated services.

## 8. References

- [1] Blake, M.B. and Bose, P. "An Agent-based Approach to Alleviating Packaging Mismatch", *Proceedings of the 4<sup>th</sup> International Conference on Autonomous Agents (AGENTS2000)*, Barcelona, Spain June 2000
- [2] Blake, M.B. "WARP: An Agent-Based Process and Architecture for Workflow-Oriented Distributed Component Configuration", *Proceedings of the 2000 International Conference on Artificial Intelligence (IC'AI2000)*, Las Vegas, NV July 2000
- [3] Blake, M.B. "SABLE: An Agent-Based Library and Environment to Consolidate Enterprise-Wide Data Oriented Simulations", *Proceedings of the 4<sup>th</sup> International Conference on Autonomous Agents (AGENTS2000) Workshop on Agents in Industry*, Barcelona, Spain, June 2000
- [4] Blake, M.B. "KOJAC: Implementing KQML with Jini to Support Agent-Based Communications in Emarkets" *AAAI-2000 Workshop on Knowledge-Based Electronic Markets at the 17<sup>th</sup> National AAAI Conference*, Austin, TX, July 2000
- [5] Booch, G., Rumbaugh, J., Jacobsen, I., "The Unified Modeling Language User Guide", Addison Wesley, Reading MA, 1998
- [6] Decker, K., Sycara, K., and Williamson, M., "Middle Agents for the Internet" *In the Proceedings of the 15<sup>th</sup> International Joint Conference on Artificial Intelligence, Nagoya, Japan., August 1997*
- [7] Edwards, K. 1999. *Core Jini*. Upper Saddle River, N.J.: Prentice Hall
- [8] Freeman, E., Hupfer, S., and Arnold, K. 1999. *JavaSpaces Principles, Patterns, and Practice*, Reading, MA.:Addison Wesley
- [9] Kamath, M. and Ramamrithan, K., "Correctness Issues in Workflow Management". *Distributed Systems Engineering Journal-Special Issue on Workflow Systems*, 3(4): 213-221, December 1996.
- [10] Lei, K. and Singh, M.. A Comparison of Workflow Metamodels, *Proceedings of the ER-97 Workshop on Behavioral Modeling and Design Transformations: Issues and Opportunities in Conceptual Modeling*, Los Angeles, November 1997.
- [11] Shrivastava, S. and Wheeler, S., "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications" *The 4th International Conference on Configurable Distributed Systems (CDS'98)*, Annapolis, Maryland, USA, May 4-6 1998.
- [12] Sun Microsystems Inc., Java Beans Specification, Distributed Event Model Specification, Java Space Specification, and the Java Language Specification. <http://java.sun.com>