A Tuple Space Based Agent Programming Framework

Yu Zhang

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada

April 2004

Your file Votre référence
ISBN: 0-612-94761-0
Our file Notre référence
ISBN: 0-612-94761-0

Abstract

A Tuple Space Based Agent Programming Framework

Yu Zhang

Software agent has become a research focus in distributed systems in recent years. This thesis aims at developing a methodology that facilitates the design and implementation of distributed agent applications. We propose an agent programming model called TSAM, which is a development framework for building distributed agent systems. TSAM provides an agent architecture that distinguishes three types of agent behaviors as (i) sensory behaviors, (ii) reactive behaviors, and (iii) proactive behaviors. Role models are used to design different proactive behaviors assigned to an agent. TSAM supports agent couplings with both message passing and distributed tuple spaces. A tuple space facilitates dynamic coordination among a group of agents that work together towards a common goal. We apply TSAM to an example of an e-market system to validate its usefulness, simplicity and support for dynamic couplings among application agents. Performance testing is conducted on the implemented system to demonstrate that the flexibility of tuple space based coordination does not incur significant runtime overhead when compared with message passing.

Acknowledgments

I would like to express my gratitude and respect to my thesis supervisors Dr. Hon F. Li and Dr. T. Radhakrishnan for their invaluable guidance, encouragement and support during the whole period of the research work.

I would also like to thank my colleagues in my research group for their suggestions, discussions and help on my thesis.

Table of Contents

Lists of Figures

List of Tables

Preface

Software agent has become the subject of much research in a wide range of fields, especially in distributed system design. The objective of developing methodologies for building distributed agent systems pulls research results from agent theory, agent languages, and agent based applications. However, two major limitations have blocked the extensive use of multi agent systems in applications. First, there is a lack of general agent programming models enabling designers to clearly define agent behaviours in a multi-agent system. Second, traditional agent interaction models are based on message passing that may not efficiently support complex collaborations among agents. This thesis is aimed at these problems by proposing a tuple space based agent programming model that is a development framework for building distributed agent systems. The main results include the following two parts:

(1) We provide a tuple space based agent programming model, called TSAM, which includes an agent architecture separating agent behaviors into sensory behaviors, reactive behaviors and proactive behaviors. In TSAM, we incorporate role models for analysis and design of proactive behaviors. In addition, we use tuple space as the agent coupling medium supplementing message passing to support dynamic couplings among agents.

(2) In order to validate the usefulness of TSAM, we apply it in an e-market application to build a distributed agent system. Through the design and implementation of this example system, we try to show that TSAM can simplify the development process and facilitate the design and implementation of flexible agent collaborations. We also perform simulation tests on our implemented system to ascertain that tuple spaces need not induce performance overhead when compared with message passing counterpart.

The thesis is organized into six chapters introduced briefly in the following.

Chapter 1 (Introduction) reviews the relevant works on agent-oriented methodologies for development of agent-based systems. These include role based agent design methodologies, agent frameworks for developing agent based applications and comparisons between message passing and tuple space based agent interaction. Based on the reviews and analysis, the contributions of this thesis are introduced.

Chapter 2 (Tuple Space based Agent Programming Model: TSAM) presents in detail the agent programming model. For the agent architecture, we model the behaviors of an agent in three distinct forms. This characterization captures both the reactivity and the mission-oriented life cycle of an agent. Role models are used to analyze and design proactive behaviors. In the implementation level, tuple space is adopted as a coordination medium for dynamic couplings among agents. Finally, the distinctive characteristics of TSAM are analyzed.

Chapter 3 (Case Study: An E-Market Application) introduces an example application of an electronic market. Role analysis is applied for the construction of proactive behaviors of agents of the e-market. The collaboration protocols among agents are described using colored Petri Net. The semantics of the agent model enables the concurrent agent behaviors to evolve with minimal internal synchronization. The objective of this case study is to show the simplicity of the design and the support of dynamic couplings among agents through TSAM.

Chapter 4 (Implementation of the E-market Application) describes the process of implementing the example system through TSAM. We show how tuple space can easily be used to realize dynamic information sharing, asynchronous couplings, and concurrent (bulk and logic template) retrieval of information in the application. In addition, we

demonstrate how to implement e-market agent behaviors through TSAM agent behavior classes.

Chapter 5 (Performance Test) reports performance tests conducted with the implemented system. We wish to ascertain the performance implication when agents communicate through tuple space rather than message passing. While it is not the intention that tuple space will completely replace message passing, it is useful to validate whether the tuple space lowers the system performance significantly. Different operating environments are tested. These include different number of agents, different sizes of markets/malls and multiplicity of markets. The results do not turn up with many surprises.

Chapter 6 (Conclusions) summarizes the results and draws conclusions that TSAM can simplify the development of agent applications and can effectively support dynamic couplings among agents without incurring performance degradation at reasonable system size.

Chapter 1 Introduction

1.1 Motivations

Software agent has become the subject of much research in a wide range of fields, especially in distributed system design. Agents provide a high level of abstraction for developing software to simplify the design of complex systems. Researchers generally agree that an agent is a software object located in a dynamic environment. An accepted description proposed by Wooldridge & Jennings [43] portrays an agent with the following characteristics:

(1) *Autonomy*: is the ability of an agent to be active without relying on direct and continuous intervention of its environment (human or other agents). With autonomy, an agent is responsible for its behaviors and internal state. It knows its goal, and makes decisions in order to move toward its goal. In other words, autonomy is a self-control feature that makes an agent adapt itself in a dynamic and complex environment.

(2) *Reactivity*: is the ability of an agent to sense and react (stimulus-response) to external stimulus with appropriate simple actions. An agent receives stimulus through its sensory and communication ports.

(3) *Pro-activeness*: is the ability of an agent to manage a set of behaviors to perform a mission. It exhibits appropriate behaviors based on monitored conditions and interacts with other agents in order to achieve common goals.

(4) *Sociality*: is the ability of an agent to cooperate with humans or other agents. Agents must interact and cooperate with other agents when they perform their tasks.

The agent paradigm has led to new and promising research on agent-oriented methodologies for design and development of application systems. Traditionally, a process-based model is used to build software systems. In such cases, the design process involves functional decomposition. It is unnatural to handle the inherent complexity of some open and dynamic systems, such as complex information system, e-commerce, and business processes management. On the other hand, aforementioned characteristics of an agent can be meaningfully applied to capture the presence of such real-life applications. The success of an agent-based system depends on a thorough and integrated agent based design methodology.

Generally, an agent-oriented methodology involves an agent architecture and a corresponding programming framework to support the analysis, design and implementation of an agent application at multiple levels of abstraction. In the literature, there are diverse researches on agent methodologies and a lot of efforts have been devoted into agent programming frameworks. However, there are relatively few systematic research results on practical methodologies for analyzing and designing a multi-agent system in the software development process. The social abilities of agents involve interaction protocols and behavioral coordination. This in turn leads to more complex software development associated with agents. The following two aspects in developing an agent-based application remain as important research focus:

(1) First, an agent programming model is needed that combines all meaningful mainstream technologies.

(2) Second, traditional agent interaction models are based on message passing that may not efficiently support the flexible requirements and dynamic coordination among agents. A complementary facility to efficiently support agent couplings is desirable.

Our aim is to address the two issues. We hope to develop a model that is flexible and easy to use, and at the same time will not incur much degradation in performance in the resulting implementation.

For this purpose, an agent programming model called TSAM, is developed in this thesis. TSAM provides not only agent architecture in abstraction, but also an easy to use programming environment for implementing the resulting design. The incorporation of role models derived from object-oriented methodologies into the design of agent behaviors supports the analysis and design of multi-agent systems from the perspective of agent-oriented software engineering. In the implementation level, TSAM also supports tuple space based agent coupling mechanism. This leads to some positive effects as illustrated through a case study of an e-market application in this thesis.

The remaining sections review the relevant developments of agent-oriented modeling methodologies, agent programming frameworks, and agent interaction methods based on message passing. Finally, the contributions of the thesis work are presented.

1.2 Role based Agent Design Methodologies

Object-oriented role modeling is a software engineering technique for specifying, analyzing, and designing systems in object-oriented way. Whereas classes describe the capabilities of individual objects, roles emphasize collaboration relationship between objects and offer a more goal or action-oriented perspective [14][37]. Generally, a role specifies a set of possible behaviors of an agent at an appropriate abstraction level and its high-level interactions with others. Over the past few years, various research results on how to use object-oriented role models in agent systems have appeared. According to Kendall [22], a role is characterized by its responsibilities, collaborators, and relationships to other roles. A role model describes roles and the relationships between

the roles. In the agent-oriented modeling approach of Zamboneli [46], role models are used to express organizational structures of multi-agent systems. The following discussions review some example models and issues of relevance.

The Gaia Methodology for Agent-Oriented Analysis and Design

Wooldridge [44] presents a general methodology, Gaia, for agent-oriented analysis and design that can be used to systematically develop an implementation–ready design based on system requirements. The main models are shown in Figure 1-1.

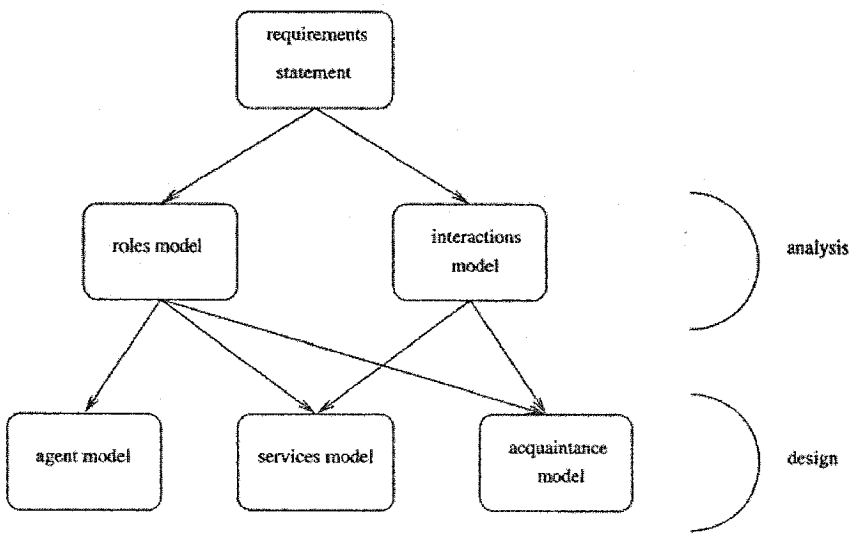


Figure 1-1 Gaia's Models

In Gaia, the objective of the analysis stage is to capture system organization structure through abstract entities of roles (roles model) and interactions (interactions model). A roles model identifies key roles in the system, which can be viewed as an abstract description of an entity's expected functionalities. The interaction model represents the interaction links between the various roles. The process of Gaia's design is to map roles into agents. It involves generating three models: (1) the agent model identifies the agent types that will make up the system; (2) the services model identifies the main services

that are required to realize the roles assigned to an agent; (3) and finally the acquaintance model documents the lines of communication between the different agents.

Gaia is concerned with how a society of agents cooperates to realize the system-level goals. However, it is not concerned with how an agent realizes its services, and leaves this problem to specific applications. In TSAM, a generic role-based agent architecture is provided for analysis and design of agent behaviors, and an agent programming framework can directly support the deployment of agent systems.

Role Models as Patterns for Agent Analysis and Design

Kendall [22] describes role modeling as a software engineering technique for analysis, design and implementation of multi-agent systems. She extends role models to represent patterns of agent interactions that can be identified and reused in the design of agent applications. Collaborations among agents are emphasized and abstracted by role models. A supply chain is selected as an example for the role models of agent systems. At the highest level, a supply chain is made up of three roles – Supply Chain (SC) Head, SC Tail and SC participants, and the pattern of interaction has been represented in terms of a role model of Predecessor-Successor. A set of agents (aCustomer, Enterprise1 and Enterprise2) can be arranged to play these roles as shown Figure1-2. Enterprise1 is a manufacturing company with a hierarchical structure of Manager-Subordinate pattern. The relevant roles appear in the upper half of the diagram. The dashed arrows indicate role assignments. An agent can play more than one role (for example, aPlantManager can play the roles of a Successor in the supply chain and as a Manager in the bureaucracy management).

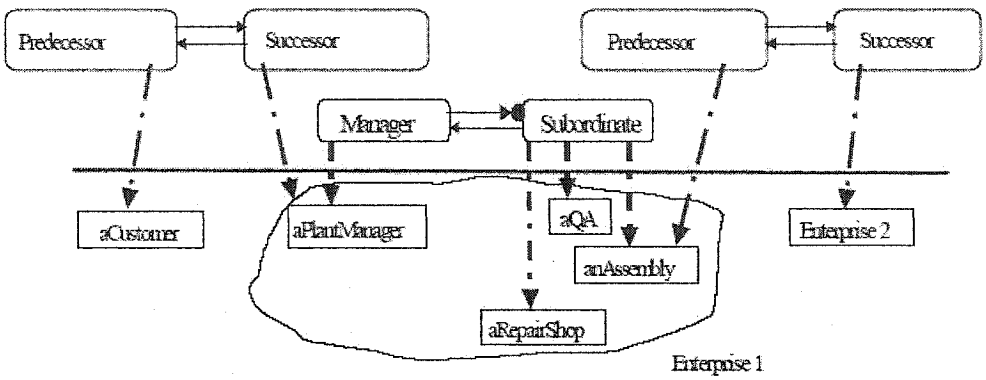


Figure 1-2 Role Model in Supply Chain Application

This approach emphasizes the analysis and design phase of agent-based systems. In TSAM, besides using role models to build the system organization, we also extend them in the definition of agent proactive behaviors as role-based behaviors. The beliefs, plans of an agent are partitioned on the basis of the roles the agent plays into different role-based behaviors. The role model can also support the couplings among different role-based behaviors within one agent.

Role-based Infrastructure for Agents

According to Cabri [4], roles are used to construct an infrastructure between applications and environment to simplify the design and implementation of agent applications. In the role-based infrastructure, four levels are proposed as shown in Figure1-3. The agent level is the application level where agents live. The infrastructure level contains the roles the agents will play. The policy & mechanism level defines the relationships among the roles and the policies related to the environment. Finally, the resource level contains the resources related to implementations. The infrastructure level is not bound to a given application or to a given environment. However, it can be adapted to different implementations of the policy and mechanism level.

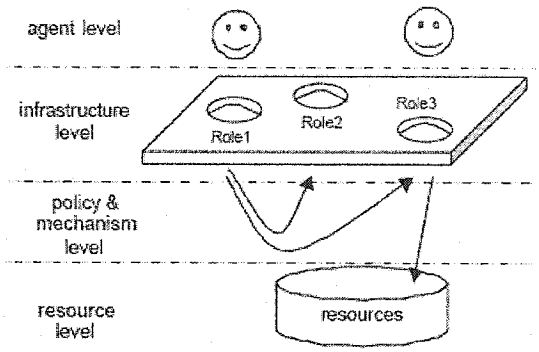


Figure 1-3 Role-based Infrastructure

The objective of the infrastructure is to achieve the separation of concerns between matching agents to roles and the actual implementation of the policies & mechanism level. The role-based behaviors of an agent are the external behaviors that are captured by roles, and they can be generated by the definitions of roles and their concrete implementations. However, the coordination of the different role-based behaviors within an agent is not considered. In TSAM, the definition of a role-based behavior in a high-level is provided. Role-based behaviours of an agent can be implemented according to the different roles the agent plays, and they are conformable with the internal mechanisms of an agent, such as the management of the execution of each behavior and the couplings among the behaviors.

1.3 Agent Frameworks for Developing Agent based Applications

There is an emerging need for agent frameworks in developing agent-oriented systems. The objectives of agent frameworks are to provide a rapid prototyping development environment for the systematic construction and deployment of agent-oriented applications and to encourage code reuse and standardization of agents.

Many agent frameworks consider both architecture and implementation of agents. They all have agent behavior engine, communication interface, and corresponding primitive processing objects. However they differ in the prominent issues they address. Agent frameworks may be categorized into: (a) architecture based agent frameworks that put agent architectures as the basis of development, and (b) platform based agent programming environments that provide an agent programming and execution environment. JAFIMA [23], ZEUS [31], and AA [28] are the typical architecture based frameworks, whereas JADE [59], Aglet [48], Ajanta [49], and Grasshoper [56] are platform based agent programming environments. In the following part, some representative agent frameworks are reviewed.

1.3.1 Architecture based Agent Frameworks

JAFIMA [23]—Java Framework for Intelligent and Mobile Agent

The efforts of JAFIMA concentrate on developing architectures, the associated tools and techniques for development of agent systems. In its agent architecture, an agent is decomposed into seven layers shown in Figure1-4.

In the layered architecture, the higher-level behavior depends on the lower-level capabilities and there is two-way information flow between the neighboring levels. In bottom-up transactions (following the arrows in the figure along the right side), an agent's Beliefs layer is based on the input from its Sensory layer. Then, the Reasoning layer may reason about its beliefs to determine what to do based on which the Action layer will decide on an action. If the agent has no capabilities to perform the action, the Collaboration layer will decide how to collaborate with other agents. These will involve the Translation layer to formulate the actual messages, and the Mobility layer to transport agents to distinct societies. In top-down transactions (following the arrows on the left side

of the figure), an incoming message enters into the Mobility layer and is translated into the agent's semantics. Then there is an information flow from the higher layer to lower layer to determine each layer's works.

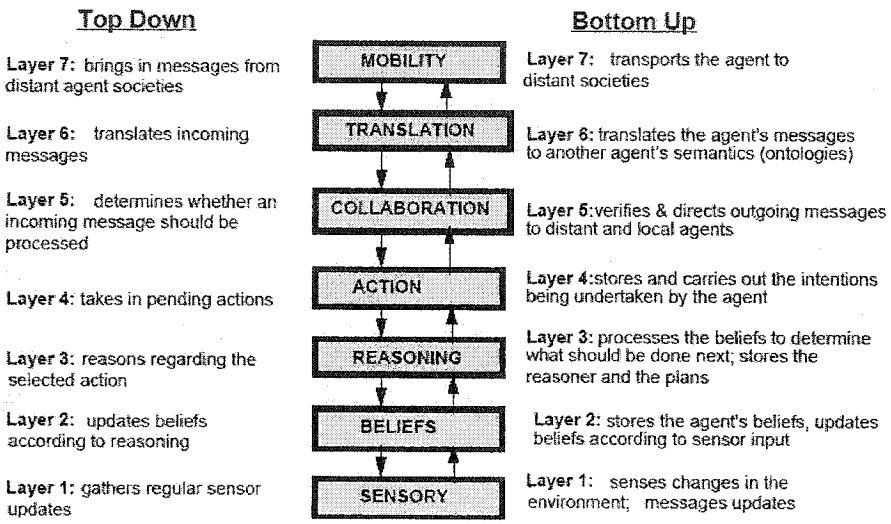


Figure 1-4 JAFIMA Agent Architecture

JAFIMA provides a layered architecture of an agent. Each layer is formed of sub-frameworks and contains design patterns. Unfortunately, it lacks support of system analysis, and the behaviors of each layer take place by a sequence according to the information flow. In TSAM, we add role models in the analysis of agent behaviours, and separate reactive behaviors from proactive behaviors. As a result, TSAM explicitly supports concurrent executions of agent behaviors.

ZEUS [31]: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems

ZEUS provides an integrated environment for the rapid deployment of distributed multi-agent systems through capturing user specification of agents and automatically generating the executable source code of the user-defined agents. In the architecture of a ZEUS

agent, a layered approach is also adopted (shown in Figure 1-5), and each of the layers is similar to JAFIMA.

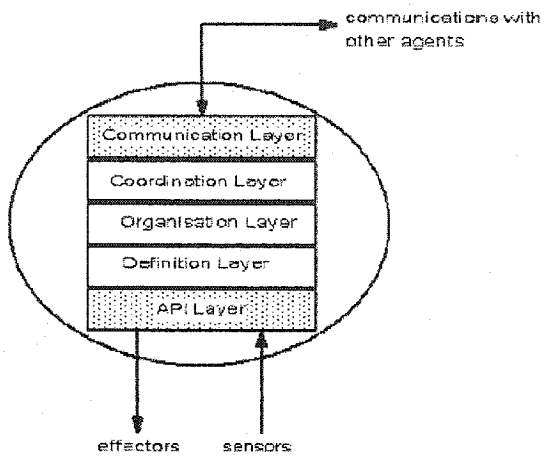


Figure 1-5 Abstract Architecture of a ZEUS Agent

Unlike JAFIMA, ZEUS provides a generic and customized tool-kit to facilitate the development of complex agent applications and automatic code generation. Because of the layered architecture of an agent, ZEUS has the same limitations in supporting concurrent behaviors of an agent as JAFIMA. In the ZEUS architecture, the sensory part is not considered, so an extra agent called Database Proxy agent is needed to deal with the application environment. On comparison, TSAM incorporates the sensory part and communication part in an agent, and does not need special agents to perform the sensory actions.

AA [28]: Agent Academy

AA, an integrated framework for constructing multi-agent applications, is implemented upon JADE [59] infrastructure. It provides an integrated GUI-based environment that enables the design of a single agent or multi-agent communities using common drag-and-drop operations. Application developers even do not need to write a single line of source code through the set of graphical tools. In terms of architecture, AA pays attention to the reasoning capabilities of an agent, and implements a "training module" to embed

essential rule-based reasoning into agents. An AA agent comprises a set of behaviors that are created by Behavior Type Design Tool in advance. These behaviors are not necessarily reactive behaviors or proactive behaviors, and AA does not consider in detail the concurrence of multi role-based behaviors of an agent. Moreover, like JAFIMA and ZEUS, AA supports message passing based agent interactions, which are totally different from the way of agent couplings in TSAM. We will discuss this point in section 1.4.

1.3.2 Platforms based Agent Programming Environment

A multi-agent system platform is a software infrastructure used as an environment for development and execution of agent systems. These platforms concentrate on providing an environment for agent programming and execution. There exist a number of platforms, which may be regarded as middleware residing between the application layer and the underlying host and network operating systems. The general criteria, according to which to evaluate the platforms, are their standard compatibilities, communication, agent mobility, security policy, availability, usability and documentations, and development issues [16]. Even though these platforms may not provide concrete agent architectures, their support services and the reuse of code are useful in building agent based systems.

In this thesis, an agent platform—JADE is selected to implement TSAM and the application system. JADE is a Java Agent Development Framework for developing multi-agent systems. It simplifies the implementation of multi-agent systems through a middleware and a set of graphical tools that supports the debugging and deployment phases.

The reasons why we select JADE as the programming environment are the following. First, JADE supports agent mobility and JADE agents can run in a dynamic environment. Second, implementing agent behaviors from a hierarchy of support classes of JADE is

simple. These classes offered by JADE are the primitive classes. Although they do not offer programmers much in terms of reusable code, they can provide a flexible mechanism for a user to implement role-based behaviors by simple class extension. Finally, JADE is a free and open source software. This allows us to change the source code by embedding other agent interaction medium, such as distributed tuple spaces.

1.4 Message Passing based Agent Interactions

The software agent paradigm strongly relies on the interaction of autonomous and cooperating processes. Currently, the common interaction mechanism is message passing. It is a high level and structured communication between a source (sender) and a destination (receiver). There are three main agent standardization groups: KQML [60], OMG's MASIF [61] and FIPA [62], which address message passing based communication among agents. KQML is a high-level, message-oriented, communication language for exchanging information independently of content syntax and ontology. The detailed descriptions of the syntax, collection of performatives, and semantics of KQML are introduced by Labrou and Finin [25]. FIPA (The Foundation for Intelligent Physical Agents) is a non-profit organization. The standard FIPA Agent Communication Language is FIPA ACL whose syntax is similar to that of KQML. MASIF is a standard for mobile agent systems. It addresses the interfaces between agent systems, and restricts the interoperability of agents based on CORBA. Today, many agent frameworks and programming platforms support the message-passing based interactions between agents. Message passing allows agents to communicate through channels or ports. However, as message passing is a lower level form of synchronization, global coordination of agents through message passing may be more complex.

Exploiting parallel and distributed systems requires programming models that deal with the coordination of large numbers of concurrently active entities. This has led to find a coordination model that effectively supports to coordinate existing (sequential or parallel) components.

Gelernter [13] proposes a shared repository based coordination model named tuple space. It is the indirect coordination model. The coordination medium is a shared data space (associative blackboard) made of tuples inserted by means of 'out' operations, and retrieved by means of 'in'/'read' operations. In TSAM, we use tuple space for agent couplings. From the surface, tuple space just supports the interactions between agents instead of message passing; however from the viewpoint of coordination, tuple space is expected to simplify the design and programming of complex coordination (dynamic coupling) among agents. The realization of the tuple space based agent coupling facilities is a separate project [64] and is performed by another student, Y. Li, who implemented the tuple space services on top of JADE. The details of how to use tuple space in TSAM will be introduced in Chapter 2.

1.5 Contributions of the Thesis Work

This thesis work is to develop an agent programming model based on tuple space that is a development framework for building distributed agent systems. It represents a considerable contribution to agent system analysis, design and implementation in the following aspects: First of all, in TSAM, we use role models to support the analysis of agent proactive behaviors. Second, we provide a generic agent architecture and an agent programming environment for programming agent behaviors consisting of three distinct types (sensory behaviors, reactive behaviors and proactive behaviors). Third, we adopt tuple spaces for agent couplings to support complex coordination among agents.

In this thesis, TSAM has been utilized to develop an agent-based e-market application in order to demonstrate the power of TSAM in design and implementation of agent based applications. As a tuple space is a centralized resource that facilitates agent coordination, we also wish to establish that it does not necessarily cost us performance loss.

Chapter 2 Tuple Space based Agent Programming Model

While Chapter1 provides an introduction and overview of the thesis, in this chapter we introduce our agent programming model—TSAM in detail. Agent oriented programming relies on the assumption that a complex distributed software system can be programmed as a set of interacting software entities, called (software) agents. Generally, agent-oriented methodologies include agent architecture and agent programming frameworks that support analysis, design and implementation of agent applications at different levels. Agent architecture is in the abstract level of agent model, while an agent programming framework is in the implementation level and supports building practical agent applications from the perspective of software engineering. TSAM is such an agent-oriented methodology that incorporates the agent architecture in the abstract level and an agent programming framework in the implementation level for building multi-agent systems. Besides, it support tuple space based agent coupling mechanism. The structure of TSAM is shown in Figure 2-1.

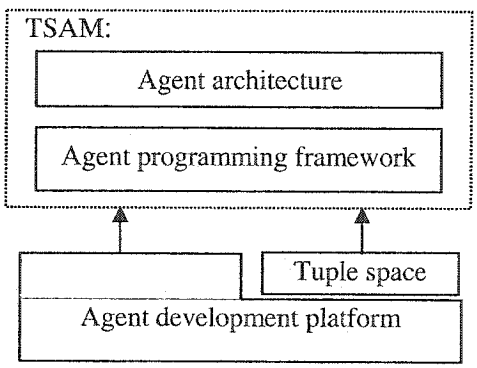


Figure 2-1 Structure of TSAM

We will explore TSAM in the following sections. In section 2.1, we present the agent architecture and its components and explain the rationale of separation of three different agent behaviors. Then, in section 2.2, we explain how the agent programming framework supports building software agents in a concrete way, which includes how to establish an agent class based on agent behavior components, and how to support the concurrent activities of an agent. In section 2.3, we describe how the tuple space based coordination model supports flexible collaborations among agents, and finally in section 2.4, we compare TSAM with other agent programming models in many aspects.

2.1 Agent Architecture

2.1.1 Current Agent Architectures

Even though different approaches to the construction of multi-agent systems impose different requirements on the individual agents, an agent is generally defined as an autonomous, collaborative and adaptive computational entity with reactive and proactive behaviors. How to construct an agent with the properties we expect of them? Agent architectures can be thought of as software engineering models for constructing the needed agents.

Many early researches on agent architecture combine the psychological and behavioral studies of human beings, and describe agent models as cognitive behaviors [27][41] of the brains of human beings. According to the features of agents, an agent is composed of at least three basic parts: sensory part, communication part and decision part, whose actions are compatible with the basic action-perception cycle of cognitive brain functions shown in Figure 2-2. The sensors perceive information from the environment, the effectors affect the environment through activities, and the part of central processing

makes decisions of what actions will be taken by the effectors according to the acquired information through sensors.

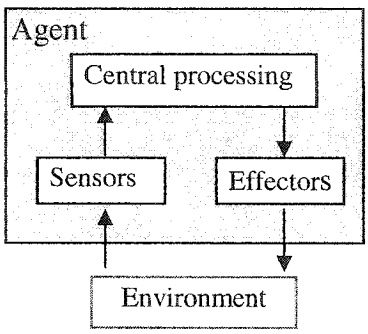


Figure 2-2 Basic Action-Perception Cycle

With the property of autonomy, an agent needs cognitive abilities to reason about complex situations and reactive abilities to respond to some changes of the environment. Therefore, in the part of central processing, an agent may have two kinds of behaviors: reactive behaviors and proactive behaviors. All the reactive behaviors are described as simple stimulus-response actions to react to the external stimulus, and the proactive behaviors need deliberate plan to achieve the goals of the agent.

Many agent architectures in the literature extend the above basic agent model with different emphasized issues. The BDI (belief-desire-intension) is one of the most adopted architecture of agents that describes agent model with mental factors like belief, preference and intension (Rao and Georgeff [34][35]). The AI oriented consideration of BDI model makes it clear to express agent behaviors. However, it is expensive to represent the intensions of agents clearly.

Hierarchical Agent Behavior Control Architecture described in [29] is a general method specifying agent behaviors. It focuses on analysis of agent behaviors and their relationships, and provides a generic specification of agent behaviors with a vertical four-layer agent structure shown in Figure 2-3. The behavior layer performs reactive activities;

the local planner layer performs regular planning to achieve agent goals; and the cooperation layer is responsible for interactions and cooperation among agents. Each layer corresponds to their knowledge bases, which are the abstractions of the environment in different levels. When an agent perceives information from the environment, if the event is just for reactive activity, the agent can do it directly. If the reactive activity is not enough, the planner layer will be activated. Moreover, when the local planner can't solve the problem, the control will move to the high layer—cooperation layer, which is in charge of the cooperation with other agents.

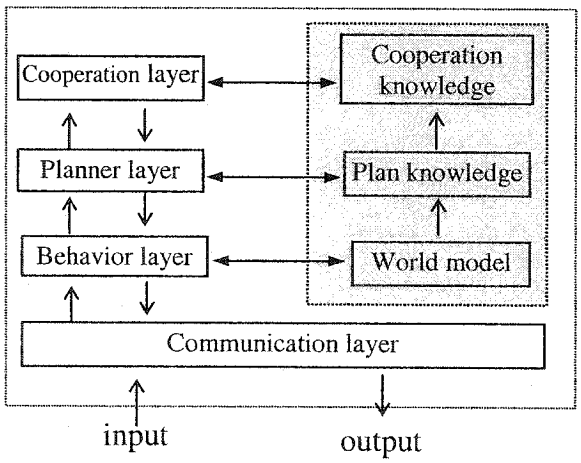


Figure 2-3 InteRRAP Structure

The layered architecture directly represents the natural rule of functional decomposition of agent behaviors. However, the interaction design within an agent is complex. Whenever there is an input, all the layers may be involved in generating an output.

The above agent architecture is based on the AI perspective, which describes agents as entities with knowledge to be able to reason in a human-like way. Agent-oriented software engineering is centered on the design of autonomous, active and interacting agents. It mainly cares about how to design flexible and interactive entities with clear definitions of agent's behaviors and their interaction protocols. Development of

distributed software is complex and it requires clear configuration and easy programming. In multi-agent based applications, the uncertainty of the dynamic environment, the complexity of coordination among autonomous agents and the multiple roles played by each agent further necessitate the agent architecture to simplify the process of development. The layered architecture does not scale well on electronic commerce, and other applications that involve a number of independently designed and operated subsystems.

In TSAM, we aim at these requirements of a software engineering process to provide different agent architecture. The following section will elaborate this architecture in detail.

2.1.2 Agent Architecture in TSAM

As an agent-programming model, the primary goal of TSAM is to provide a means for building agent applications to facilitate analysis, design and implementation of software agents in a relatively rapid and easy way. The agent architecture of TSAM is proposed as in Figure 2-4. Based on the basic agent model of action-perception cycle, the communication part and central processing part are still included in the architecture; however, they are deliberated in different ways from the traditional layered architecture for easily building a software agent.

As a software entity, an agent generally has three types of behaviors: sensory behaviors, reactive behaviors, and proactive behaviors. Accordingly, in the agent architecture, we classify an agent's behaviors into three categories, which are sensory part, reactive part and proactive part.

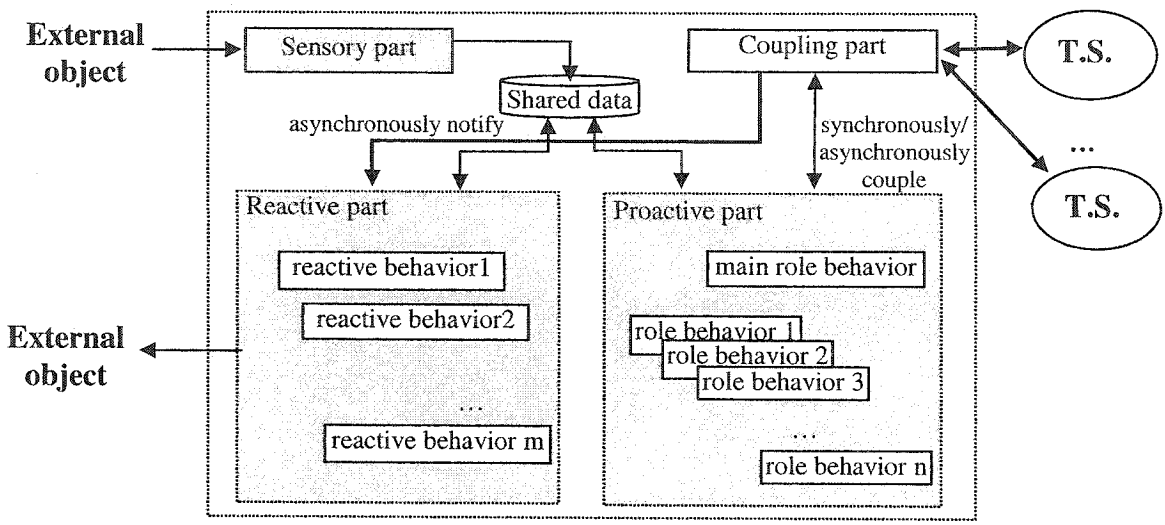


Figure 2-4 Agent Architecture of TSAM

Sensory Part:

The sensory part includes agent sensory behaviors that periodically sense the changing environment where the agent lives. The adaptability of agents enables them to readjust themselves to adapt to the environment. In this case, we assume that an agent often knows quite a bit about its environment and knows how to adapt to its environment. For example, a mobile information agent in an e-commerce application may need to sense its operating environment to decide whether to move away from some sites. Meanwhile, it also keeps sensing the status of local data resource. If it knows that the accessed data sources are problematic, it may stop searching activity immediately, and move to other places.

Reactive Part:

The reactive part includes reactive behaviors that can be regarded as stimulus-response pattern. The stimulus may be from the external objects sensed by the sensory part or from the other agents' notifications through the coupling part. Therefore, the reactive behaviors are divided into two different types. One type includes simple behaviors that are triggered by other behaviors of the agent. For example, they may be triggered by the

sensory behavior to react to the external environment, or by the proactive behaviors to react to some internal results. The other type is as asynchronous listeners to other agents to perform collaborations with other agents. The latter can be realized through reactive tuples.

A reactive behavior is a precise and deterministic action without long-term proactive protocols needed to interact with other agents. It may be a simple activity starting at once, like stopping some actions, sending alarm signals, or just triggering an agent's proactive behavior. For example, in e-market application, a seller agent as an auctioneer may have three reactive behaviors. One may be triggered by an abort event of a buyer agent to stop all transactions with the buyer agent. The second one may be triggered by a change of a price level of a product in the market to readjust its selling price for this product. The third one may be triggered by a new bidding request from a buyer agent to activate an auction behavior.

Proactive Part:

Proactive part includes agent proactive behaviors that implement the deliberate part of an agent. Proactive behaviors are task initiative behaviors that do not simply act in response to their environment, but are able to exhibit goal-directed behaviors to perform particular tasks. Collaborations with other agents to perform such a task are often needed. An agent can be defined to have multi-capabilities or to play multiple roles. As a result, each role-based behavior can be defined to perform a specific task. The main role behavior in Figure 2-4 is a proactive behavior that runs periodically as a dispatcher of other role-based behaviors. In the e-market application, for example, a seller agent can be defined to have the capabilities of advertising product prices, as well as auctioning and negotiating with buyers. In turn, a buyer agent may have at least three role-based behaviors: searching information, bidding for products, and negotiating with seller agents.

The following are the main points of the agent architecture of TSAM compared with the traditional layered architecture.

(1) Separate Sensory Part from the Coupling Part

There are two ways for an agent to learn about its outside environment. One is to sense the dynamic environment through the sensory part. The other one is to interact with other agents for a particular task through the coupling part. Some agent architecture put the two parts together in one communication module because both of them can be regarded as the same thing of interacting with the outside world of agents. However, we separate the two parts for the following reasons. First, they reflect totally different behaviors with distinguished issues, so it is much clearer to model them separately. The sensory part performs the works of periodically perceiving the environment for some specific parameters and keeping them in the internal shared database for successive stages of decision-making or reactions. On the other hand, an agent still faces one or more tasks to be solved through collaborating with other agents. The coupling part just performs this work to provide unified interfaces for couplings among a group of agents. Moreover, the sensory part and coupling part may often lead to concurrent behaviors. While an agent coupling with other agents, it may sense the environment at the same time. The independence of the two parts reflects the natures of agents' behaviors. Therefore, their separation simplifies the underlying implementation of the two parts and facilitates the design of the reactive behaviors and proactive behaviors of agents.

(2) Separate Reactive Part from the Proactive Part

Generally, an agent may combine aspects of both reactive and proactive behaviors so that it can make use of the best features of both behaviors. Some agent architecture process the two types of behaviors in the same workflow. In the layered architecture, for example, any event, no matter what behaviors it will cause, is in turn sent to each layer and is

processed by each layer. In TSAM, however, we separate the reactive part from the proactive part, define them separately, and activate them concurrently if necessary.

First of all, the separation of the two parts simplifies the design process because it supports the concurrent execution of the two types of behaviors. In traditional architectures, the actions of reactive or proactive behaviors are started by a "central scheduler". This is in fact a sequential mechanism that is not suited for the nature of agent's behaviors. Separating reactive behaviors from proactive behaviors allows a developer to manage different behaviors simply, and results in a better structured design with explicit concurrency that is more easily implemented in an agent. This is a fundamental aspect of our TSAM framework.

Second, when designing a software agent, it is important for agents to be as easily configurable and scalable as possible. This requirement needs the fact that, for building a new agent, it must be easy to establish its components through behaviors definition; for extending an existing agent, it must be simple to add new role-based behaviors to scale up the agent when it plays additional roles. The separation of reactive behaviors from proactive behaviors makes it easy to change/add/delete different role-based behaviors without interfering with other components in the architecture.

Finally, from the perspective of the life cycle of an agent, reactive behavior and proactive behavior have different effects on the state transition of an agent. Figure 2-5 shows a typical life-cycle model of an agent [59], where an agent has a basic life cycle with six states (Initiated, Active, Waiting, Suspended, on the Move and Unknown). To build a complete agent, it is necessary for the agent architecture to support the state transitions very well. Proactive behaviors can change the agent's state according to its role-based protocols. However, to avoid directly affecting the on-going execution of a proactive behavior, a reactive behavior may inform the proactive behavior its intention through the

shared object rather than directly change the agent's internal state. For example, in an e-market application, when a buyer agent moves to the location of a local market and begins to negotiate with a seller agent, the local market is down. In this situation, a reactive behavior is activated immediately to inform the proactive behavior, which then may decide to wait for the market recovery or move to another market. The different influences of reactive behaviors and proactive behaviors on agent state transition make it necessary to separate them in agent architecture.

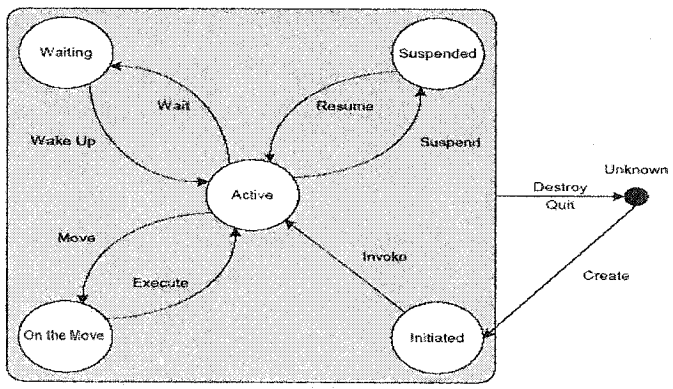


Figure 2-5 An Agent's Life Cycle

(3) Support Definition of Role-based Behaviors

In agent-based systems, a role model identifies and describes a structure of interacting entities in terms of roles that will be played by agents. The capabilities of roles represent a set of actions that are needed for an agent to achieve its tasks. In the agent architecture, the proactive behaviors are role-based behaviors that are defined according to the responsibility of roles. The start of role-based behaviors is based on couplings with the environment or other agents. Each role-based behavior can be modeled as a sequence of tasks related to a role. It involves a partial order of tasks and is mapped to one thread when running. The model of role-based behaviors is shown in Figure 2-6 similar to [15]:

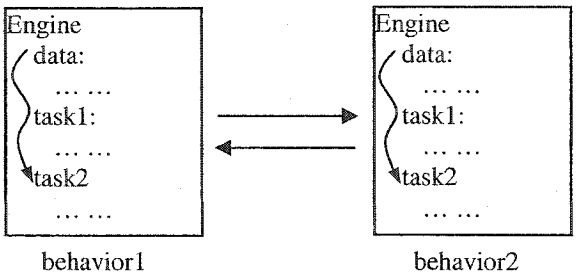


Figure 2-6 Role-based Behaviors

Using role model in the analysis and design of agent proactive behaviors is an essential part of agent-oriented software engineering. In the analysis stage, an agent is defined to play one or more roles in applications. In the design stage, each proactive behavior is defined separately in accordance with the role-related tasks. In the implementation stage, when creating an agent playing a set of roles, the corresponding predefined role-based behaviors are simply assembled into an agent class. Therefore, the definitions of role-based behaviors make the development of agent systems faster and clearer than otherwise.

2.2 Agent Programming Framework

Figure 2-7 is the class diagram of an agent in TSAM. An agent class is mainly composed of three types of behaviors: *Sensory Behavior*, *Reactive Behavior* and (Proactive) *Role Behaviors*. The class of the *Shared Data* is used in the couplings among the behaviors within an agent, while the *Coupling* class facilities the couplings between the agents. The *Coupling* class is an interface class that is implemented by tuple space services.

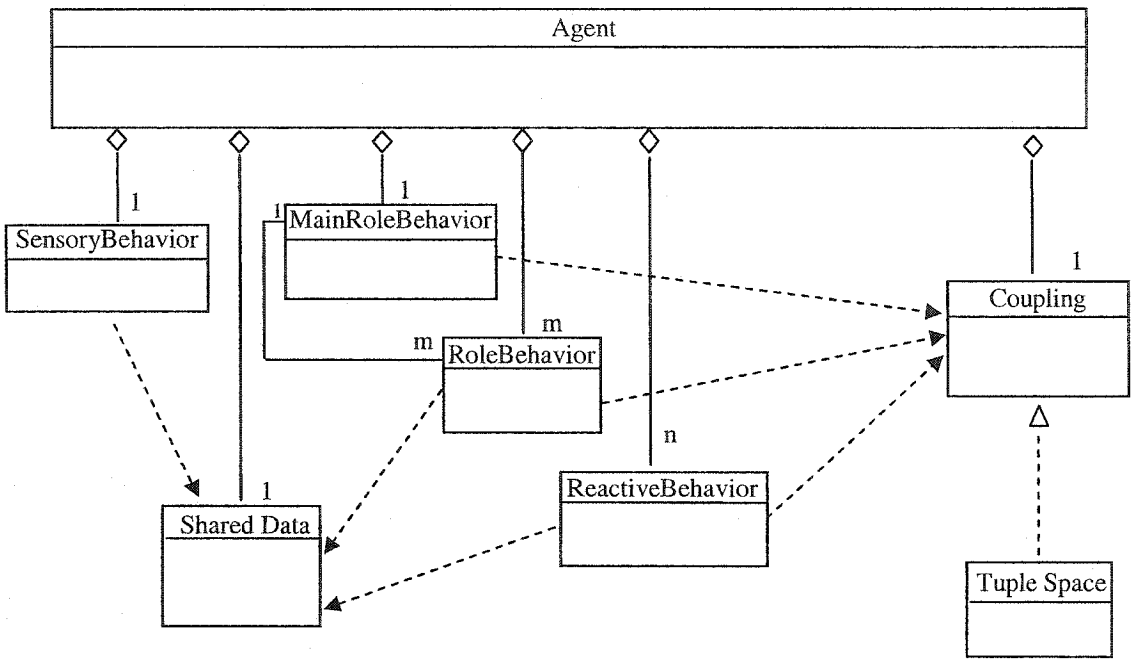


Figure 2-7 Agent Class Diagram

The main framework of class definitions used in our agent programming model is as follows:

public abstract class **TSAM_SensoryBehavior** extends JadeCyclicBehaviour{

TSAM_SensoryBehavior (Agent agent, long p); *//construction*

public void action();

public abstract void myaction(); *// perceive actions implemented by application agents*

}

public abstract class **TSAM_MainRoleBehaviour** extends JadeCyclicBehaviour{

public TSAM_MainRoleBehaviour (Agent agent, long p); *//construction*

public void action();

public abstract void myaction(); *//dispatch strategy implemented by application agents*

}

public abstract class **TSAM_RoleBehavior** extends JadeSimpleBehaviour{

public TSAM_RoleBehavior (Agent agent); *//construction*

public void action();

public abstract void myaction(); *//role based tasks implemented by application agents*

}

public abstract class **TSAM_ReactiveBehavior** extends JadeSimpleBehaviour implements IReactive{

TSAM_ReactiveBehavior (Agent agent); *//construction*

public void reactTo(AgentReactionEvent e);

public void action();

public abstract void myaction(); *//concrete reactions implemented by application agents*

}

Coupling class (interface of tuple space service){

TupleSapceID TSCreate(String name);

TupleSapceID TSFind(String name);

//synchronous access of tuple spaces

... ...

//asynchronous access of tuple spaces

... ...

// reactive primitives

... ...

}

2.3 Tuple Space based Coordination Model

2.3.1 Coordination Model in Multi-Agent Systems

Agents in a multi-agent system usually need to communicate amongst themselves, to coordinate their activities and to negotiate once they find themselves in conflict. The collaborative features of agents in multi-agent system make the coordination model a crucial factor in the design of agent systems. A common coordination in multi-agent system is that agents collaborate to achieve a goal. Coordination problems are the management of dependencies amongst activities of agents, which includes resource allocation, notification, synchronization, decomposition, and so on. In the literature, there are different coordination models for different purposes. In TSAM, however, the objective of the coordination model is to reduce design complexity and simplify the programming tasks.

Coordination models for multi-agent systems can be broadly classified into direct coordination model and indirect coordination model. Direct coordination means that agents explicitly initiate a communication via message passing and explicitly name the involved partners. The significant advantages of a direct coordination model are to permit explicit control of all interactions, which are not influenced by external entities, and to induce low overhead in the hosting execution environments [6]. However, the apparent drawbacks of direct coordination model are that the direct communication needs localizing both partners and the repeated interactions highly depend on network reliability. Some of the agent programming frameworks support the message-passing based direct coordination. For example, Agent TCL [17] provides direct communication between agents, and JADE supports the FIPA ACL compatible coordination standard.

In indirect coordination, agents interact via information spaces, like blackboard, where messages are stored and retrieved locally. A blackboard approach provides a solution to eliminate the tightly bound interaction links that many distributed technologies require during inter-process communication. It uncouples agent-to-agent interactions in time and space. This suits many application scenarios where agents do not know exactly the apriori identity of the collaborators. For example, when an auctioneer agent wants to invite a public bidding, it just publicizes the bid price of products, and does not know which bidders will participate. If the messages on blackboard are associative and if agents access the content via pattern-matching mechanism, this information space can follows Linda semantics [7] of 'in()', 'out()', read() etc. In TSAM, we use this type of indirect coordination model based on tuple space for agent couplings.

2.3.2 Why Use Tuple Space as the Coordination Model?

With the increasing complexity of applications, we need programming models to deal with the coordination of large numbers of concurrently active entities. An infrastructure of coordination model is needed to meet the complex requirements. Tuple space that supports inter-agents coordination is an attractive solution.

In the case of a complex application environment with dynamic and heterogeneous factors, tuple space can simplify the design and programming of complex coordination (dynamic coupling) among agents because of the following reasons:

First, tuple space based coordination model promotes dynamic information sharing, so that information is available to any intended agents and every agent can modify the information from the tuple space.

Second, tuple spaces free the designer from the burden of keeping track of explicit or at least implicit addressing knowledge in agent couplings. An agent needs to communicate

with a group of agents or an arbitrary member of agents rather than a particular target agent. In such cases, if the sender must name all the potential recipients, a level of abstraction is lost.

Finally, the reactive tuples of tuple space support event-driven coordination among agents with triggering corresponding reactions. The reactions are defined based on the roles an agent plays. They can access the tuple space, change its content and influence the semantics of the behaviors to achieve better control. In addition, reactions can adapt the semantics of the interactions to the specific agent environment, thus simplifying the agent programming.

2.3.3 How Tuple Space Supports Coordination in TSAM?

Some researchers have reported their work on tuple space based agent coordination. MARS [5] proposes a reactive model to manage secure access of tuples. WCL [38] proposes a monitor to react and notify waiting agents, and JavaSpace [58] implements a notifying method in object-oriented method. Even though models of reactive tuples have been proposed and some notifying mechanisms have been implemented, there is no complete agent programming model using reactive tuples reported. In TSAM, we incorporate tuple space as agent interaction medium, and support the creation of reactive behaviors through reactive tuples. The main functions of a tuple space based coordination model include:

(1) A set of programming interfaces for data sharing. Agents can synchronously/ asynchronously read/write/remove tuples from a shared tuple space to cooperate with other agents. Moreover, tuple space supports for matching logical template of tuples. This improves flexibility of the framework to support agent coupling.

(2) Asynchronous notifications of exceptional conditions via reactive tuples. The behaviors of reactive tuples can be programmed through registering reactive behaviors predefined by agents. Whenever there are some changes in a reactive tuple, its intended behavior can be triggered.

(3) An infrastructure that enables location transparent inter-agent communication. A tuple space manager manages the name service of distributed tuple spaces. It establishes a dynamic coordination environment without knowing the exact physical locations of each tuple space.

2.4 Comparison with Other Agent Programming Models

To satisfy the requirements of software engineering, a successful agent programming model may address the practical concerns of real-world applications. There are plenty of developments in agent-oriented engineering products and research projects. Different agent programming models may have different concerns on agent systems, so they have different focus on agent technologies for developing software agents. Here, we will make comparisons between TSAM and some other frameworks.

AgentBuilder [50] is a commercial product with an integrated tool suite for constructing intelligent software agents. When defining behaviors of an agent, AgentBuilder provides an Agent Manager that concentrates on creating various mental constructs including initial beliefs, initial commitments, initial intentions, capabilities and behavioral rules. In addition, the Agent Manager supports tools of adding planning and learning capabilities to an agent to build an intelligence-oriented system. Agents communicate using message passing.

Agent Building Shell (ABS) [51], developed by Enterprise Integration Laboratory, University of Toronto, provides several reusable layers of languages and services for

building agent systems. Different from Agent Builder, ABS employs a unified description language that specifies behaviors as consisting of sequential, parallel and choice compositions of actions. Constraint-based mechanisms are used to determine which actions will be executed. For coordination, it emphasizes on constraint-directed coordination based on relationships among the agents, and provides coordination language built on top of the agent communication language to support the definition, execution and validation of complex speech-act based cooperation protocols.

AgentTool [47] is a Java-based graphical development environment to help users to analyze, design, and implement multi-agent systems. Developers define high-level system behaviors graphically using the Multiagent Systems Engineering methodology (MaSE) [42], which utilizes role models in the analysis process. Different from TSAM, it has a component for message checking and routing, and a rule container represented by formal operation definitions. It does not separate the reactive behaviors from proactive behaviors.

FIPAOS [52][57], ZEUS [31] and JADE [59] are agent programming tools for building FIPA-compliant agent systems. FIPAOS is a component-oriented toolkit enabling rapid development of FIPA compliant agents. ZEUS is a 'collaborative' agent building environment written in Java. They all have agent architectures in the high level for building agent behaviors. However, the different emphases on their agent architectures lead to their different agent behavior subsystems. Different from TSAM, both FIPAOS and ZEUS have layered agent architectures. One of the important layers of a FIPAOS' agent is Task Manager that constructs agents from primitive work units called tasks. The main layer of a ZEUS agent (definition layer) represents the agent reasoning and learning abilities, which adapt to knowledge based programming. Similar to TSAM, JADE provides a behavior engine for programming agent behaviors using homogeneous

processing elements that are executed in concurrent fashion. However, JADE agent architecture does not consider the essential differences between the three types of agent behaviors. It just provides a homogeneous behavior definition and leaves the task of creating different behaviors to application developers. ZEUS considers the sensory property of agent systems, but it sets an additional agent for sensing and affecting without built-in sensory behaviors.

In comparison to other frameworks, TSAM is an agent programming model that incorporates role model in agent behavior design, supports the definition of three types of agent behaviors, and utilizes tuple space based coordination model for agent interactions. In the following chapters, these important points will be elaborated and demonstrated through an example of an electronic market system.

Chapter 3 Case Study: an E-Market Application

In this chapter, we introduce electronic market (e-market) as an agent application, and demonstrate how the agent programming model (TSAM) proposed in Chapter 2 supports the development of the e-market application. Electronic market is a software tool with trading functions that enable buyers and sellers to effectively reach their trading objectives. The main trading activities, such as online pricing, need complex cooperation and coordination process among buyers and sellers. TSAM help designers to establish flexible and dynamic protocols for the complex trading transactions in the e-market. The objective of the case study is to show how TSAM simplifies the design process and facilitates agent programming in complex collaboration through role modeling techniques and tuple space-based agent coupling facilities.

3.1 Electronic Market

3.1.1 Electronic Market

Generally, a marketplace is comprised of consumers (single buyer and groups of buyers), and merchants (single seller, stores and malls), and there are many transaction types among these participants, such as information searching, product exchanges, online shopping, etc. An e-market is a system that supports all participants in a market to perform their trading transactions through software technologies. All the participants can benefit from the e-market. From the point of view of consumers, an e-market provides services to find products and merchants rapidly, to easily purchase products and to place orders safely. From the point of view of merchants, an e-market allows them to easily

publicize their products, to sell their products while maximizing their profits and, at the same time, to satisfy consumers.

3.1.2 Trading Transactions on E-Markets

To satisfy the objectives of consumers and merchants, generally e-markets provide the following services for the participants.

(1) Product Information Search

In many industries, the main inefficiency is the lack of updated information concerning available products on markets. Relevant product information facilitates sale processes, and global knowledge of the market creates efficiency in all transactional activities. Consumers usually need the e-market to provide product information as rapidly and completely as possible. However, it is usually costly for suppliers to provide updated product information to customers. E-market supports these services at a low cost.

(2) On Line Pricing

E-markets provide pricing mechanism that can support transactions of online pricing for buyers and sellers. The main on-line pricing strategies include negotiation, on-line auction and fixed price sales.

A. Negotiation

Negotiation is a flexible way to trade. It is often used for the exchange of products and services. During the negotiation, both sides can change the price, and finally accept a negotiated price or fail to get an agreement. The typical negotiation types are one-to-one negotiation and group-buying negotiation.

One-to-one negotiation is a price negotiation between a buyer and a seller. A seller quotes a price for some goods or services. A buyer may also initiate the negotiation by proposing a start price for some products. They can come to an agreement through many

rounds of negotiation. Any side may refuse the negotiated price and terminate the negotiation process.

Group-buying negotiation is an alternative strategy for buying products. Before starting a negotiation, a buyer, as a coordinator, may collaborate with other buyers to form a group with a great amount of buying in order to get a good discount. The coordinator is a representative of all buyers in the group and is responsible for negotiating with a seller like one-to-one negotiation. When the negotiation is finished, the coordinator will inform all the buyers in the group the negotiation result. Any buyers can enter/quit the group. However, once the negotiation begins, no one is allowed to enter or quit. They have the obligation to take the result of the negotiation.

B. On-Line Auction

It is another trading scheme, where more than one buyer tries to get products through bidding. English Auction and Group-buying auction are usually used in on-line auction.

English auction is a selling, single-item, ascending-bid auction, where a set of bidders beat the current bid. Each bidder is allowed to place a bid that must increase the current price by a predefined increment. The product is sold to the bidder who placed the highest bid when the auction ends.

Group-buying auction is a selling, group-based descending-bid auction, which provides buyers an opportunity to collaborate with others to get lower prices. First, an auctioneer publicizes auction information including items, start price, and the starting/ending time. As more buyers join the group, the price drops according to a predetermined price change trajectory. In the end, everyone in the group will be charged the same final low price even if some of them indicated a willingness to buy at a higher price.

C. Fixed Price /Limited Time Sale

Fixed price selling is a limited time sale for special offers, where the price can't be negotiated. A seller posts all the items he wants to sell, along with the price, quantity available, and the starting/ending time. Any buyer can buy the products as first come first served.

(3) Order Placing and Receiving

In a traditional market, it takes time for buyers to place orders with many different suppliers, and it is also expensive for sellers to process orders. E-markets provide solutions to reduce both purchasing and processing costs. Buyers may place orders rapidly and safely, while sellers receive orders in a standardized format. Both the errors and the cost of processing orders can be reduced.

3.2 Requirement Analysis

Because of the complexity of the trading transactions between buyers and sellers, a well-designed e-market is needed to improve the efficiency of the whole market. The requirement analysis of e-markets is to present what type of an agent based application will be built, and as a result, to show how TSAM supports to easily design and implement such a multi-agent system.

(1) Dynamic Information Sharing

The dynamic shared information is a shared data area, where each participant can dynamically write data to it or read/withdraw data from it. Through dynamic information sharing, buyers and sellers can search and get the information promptly, such as check the stock levels, track deliveries or view their order records directly. They can also modify the shared information whenever they need. Hence, with dynamic information sharing, an e-market facilitates information exchange between two parties without incurring heavy cost on them.

(2) Collaboration among Participants

Collaborations among buyers and sellers are necessary in trading transactions whether in traditional markets or in e-markets. Some examples of collaborations among buyers and sellers are as follows:

Collaboration among buyers in group-buying negotiation

In group-buying negotiation, a coordinator of the buyers takes charge of negotiating with a seller. In traditional markets, it is hard to form such a group of buying. An e-market is needed to support the easily formation of a group buying and the sharing of the subsequent negotiation results among buyers.

Collaboration among bidders and an auctioneer in on-line auction

In on-line auction, collaborations exist between an auctioneer and a group of bidders. E-markets are expected to support the auction process effectively, so that each bidder may search the current price and place its bid safely, and the auctioneer can catch the bidding prices and post the current price more easily.

(3) Dynamic Relations among Participants

There are dynamic relations among market participants that are not well supported in traditional e-markets. For example, in on-line auction, the number of bidders in an auction is fixed before the start of an auction. However, it is more flexible if a bidder can join and quit from the auction dynamically. Similar situation can be conceived in a group buying where a buyer can decide to join a group at any time. To support these types of dynamic relations that may improve the efficiency and flexibility of the market, the design of an e-market should consider supporting dynamical relations among participants.

(4) Reactivity of Agents

With complex trading activities, an e-market is often faced with both unexpected and expected events. Unexpected events include changes in the external environment,

resource problems, buyer/seller failure, etc. These events may badly affect the trading transactions in markets. It is important for an e-market to support a reactive mechanism to sense and to react to these unexpected events properly without affecting the rest of community in conducting their businesses. Expected events are caused by other agents. An agent can sample them and react to them to perform collaboration with other agents for some particular role-based tasks. Therefore, reactivity of agents is an important functionality to which an e-market is needed to make transactions safely and efficiently.

In order to satisfy the above requirements, an e-market is actually built as a distributed system with multiple collaborations, concurrent activities and complex couplings among subsystems. In the recent past, agent technologies have been applied to e-market systems [26][53][54]. Using software agents adds a new and revolutionary dimension to electronic marketplaces. Agents have the distinguishing ability to automatically finish repetitive and time-consuming tasks, including searching, buying and selling products over the Internet. Here, we select a multi-agent based system to realize an e-market application, and use the proposed agent programming model (TSAM) to design and implement the example system.

In the following sections, we show how TSAM supports the design of the e-market application and simplify the design process through the following processes:

(a) Analysis of roles and their relationships in the e-market

(b) Definition of agent behaviors

(c) Design of flexible agent behavioral protocols based on reactive tuple space to support complex collaborations among agents

3.3 Role Model

Role based analysis and role model in the e-market describe agent systems in an appropriate abstraction level. In this section, we formulate and analyze roles in the e-market and present their relationships, which are the basis of design of agent behaviors in section 3.4.

3.3.1 Roles in the E-market

There are three sides (purchase side, market side, and sale side) in the application. Although each side has different benefits and is responsible for different roles, they also collaborate to get their objectives.

On the *purchase side*, two roles are involved: (a) a role of customer manager responsible for management of buying task, (b) a role of buyers to perform buying tasks. The customer manager generates buying tasks according to the consumer's requirements and assigns tasks to appropriate buyers. A buyer competes for tasks and performs the tasks through different buying strategies.

On the *market side*, two roles are involved: (a) a role of trading recorder, and (b) a role of market manager providing services to all the participants (buyers and sellers), which includes:

(1) Registration. When a buyer or a seller enters a market for the first time, it must register to the market. The market manager will record its information related to the identification, authority, credit, and quality of service.

(2) Information providing. Any participant can inquire information from the market manager.

(3) Trade monitoring. The market manager monitors the trading actions of all the participants.

On the sale side, a role of sellers is in charge of selling products to buyers.

According to the above application scenarios, we formulate the roles by describing structured interacting entities and their capabilities. Figure 3-1 is a hierarchical description of the roles in the e-market. Roles in a higher level are the abstract description of roles below. The roles at the leaf nodes are the roles that ultimately emerge to be separately mapped to agents.

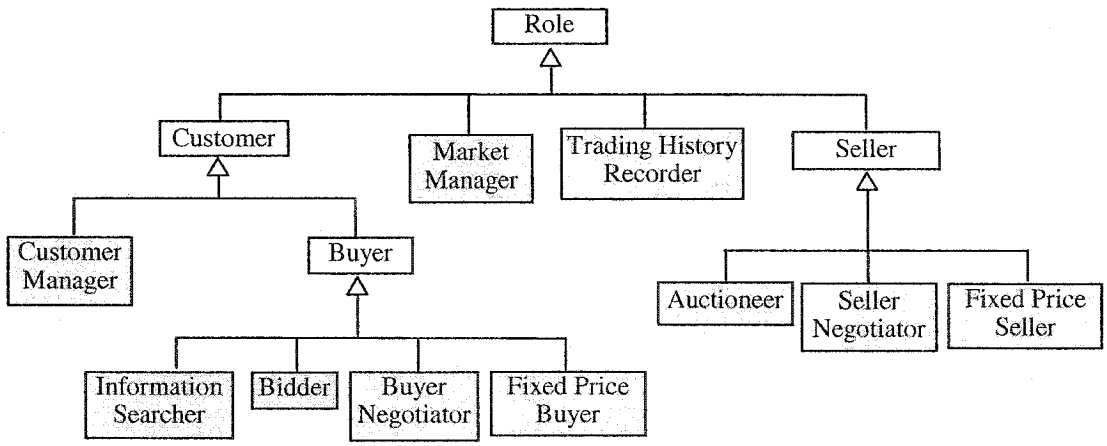


Figure 3-1 Role Tree

Each role is described by its responsibilities, knowledge and collaborators, which will be introduced in followings.

(1) Customer Manager

- *Responsibilities.* It assigns tasks to buyers, receives task results, and monitors the activities of its registered buyers.
- *Knowledge.* It knows how to create tasks, assign tasks, and monitor buyers.
- *Collaborators* are roles of Information Searcher and Buyer.

(2) Information Searcher

- *Responsibilities.* It provides service to get information from markets.
- *Knowledge.* It knows where to get products information (itinerary).

- *Collaborators* are roles of Customer Manager, Market Manager and Trading History Recorder.

(3) Bidder

- *Responsibilities.* It bids for some products through on-line auction.
- *Knowledge.* It knows how to determine its bidding price.
- *Collaborators* are roles of Customer Manager, Market Manager and Auctioneer.

(4) Buyer Negotiator

- *Responsibilities.* It negotiates with Seller Negotiators to get a good discount for products.
- *Knowledge.* It knows how to adjust the negotiated price.
- *Collaborators* are roles of Customer Manager, Market Manager and Seller Negotiator.

(5) Fixed Price Buyer

- *Responsibilities.* It buys products through public special sale.
- *Knowledge.* It knows how to buy.
- *Collaborators* are roles of Customer Manager, Market Manager and Fixed Price Seller.

(6) Market Manager

- *Responsibilities.* It manages all its participants (buyers and sellers) in a market, including register/un-register, information providing, monitoring, and authority and credit checking.
- *Knowledge.* It knows how to do the management works.
- *Collaborators* include all other roles involved in the market.

(7) Trading History Recorder

- *Responsibilities*. It records trading histories in the market and provides search results to the one who needs.
- *Knowledge*. It knows how to record.
- *Collaborators* are roles of Information Searcher and Market Manager.

(8) Auctioneer

- *Responsibilities*. It sells some specific products through auction.
- *Knowledge*. It knows the auction strategy.
- *Collaborators* are roles of Market Manager and Bidder.

(9) Seller Negotiator

- *Responsibilities*. It negotiates with Buyer Negotiators to sell products with negotiated prices.
- *Knowledge*. It knows how to adjust the negotiated price.
- *Collaborators* are roles of Market Manager and Buyer Negotiator.

(10) Fixed Price Seller

- *Responsibilities*. It sells products with fixed prices.
- *Knowledge*. It knows how to sell.
- *Collaborators* are roles of Market Manager and Fixed Price Buyer.

3.3.2 Relationships among Roles

Once the roles in an application are captured, the relationships among the roles can be formulated as well. The relationships among the roles are so important that they are the basis of the organizational structure of multi-agent systems, and the basis of agent interaction protocols.

There are two types of relationships among roles: (a) static relationships are stable, and can't be changed forever; (b) dynamic relationships are established dynamically and can

be changed at runtime. For example, a seller has a static relationship with a market manager when it registers in the market, but the relationship between a buyer and a seller may be established dynamically according to their trading needs. Figure 3-2 shows the static relations in an e-market, and Figure 3-3 shows the dynamic relations.

Static Relationships

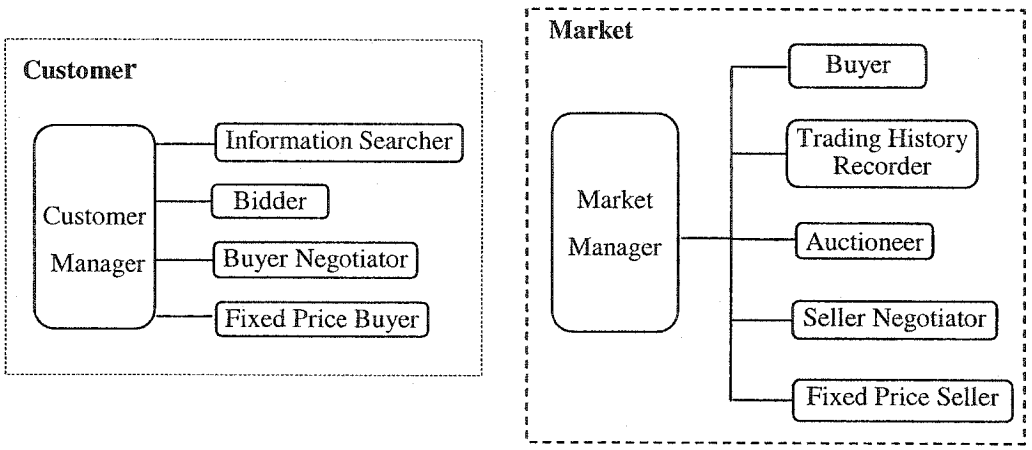


Figure 3-2 Static Relationships among Roles

In Figure 3-2, Customer Manager, Information Searcher, Bidder, Buyer Negotiator, and Fixed Price Buyer belong to a customer. They share some static information about the customer, such as customer ID and descriptions of the customer. In the market side, some information of the market is shared by all its participant roles, and the market manager provides services to all its registered participants.

Dynamic Relationships

Figure 3-3 shows the dynamic relationships among roles, where couplings among roles take place dynamically. Each of the interaction actions will be explained with action description, returned values and the parameters. The interactions between roles may be relations of one to one, one to many, many to one and many to many. The arrows show the initialization direction of interactions.

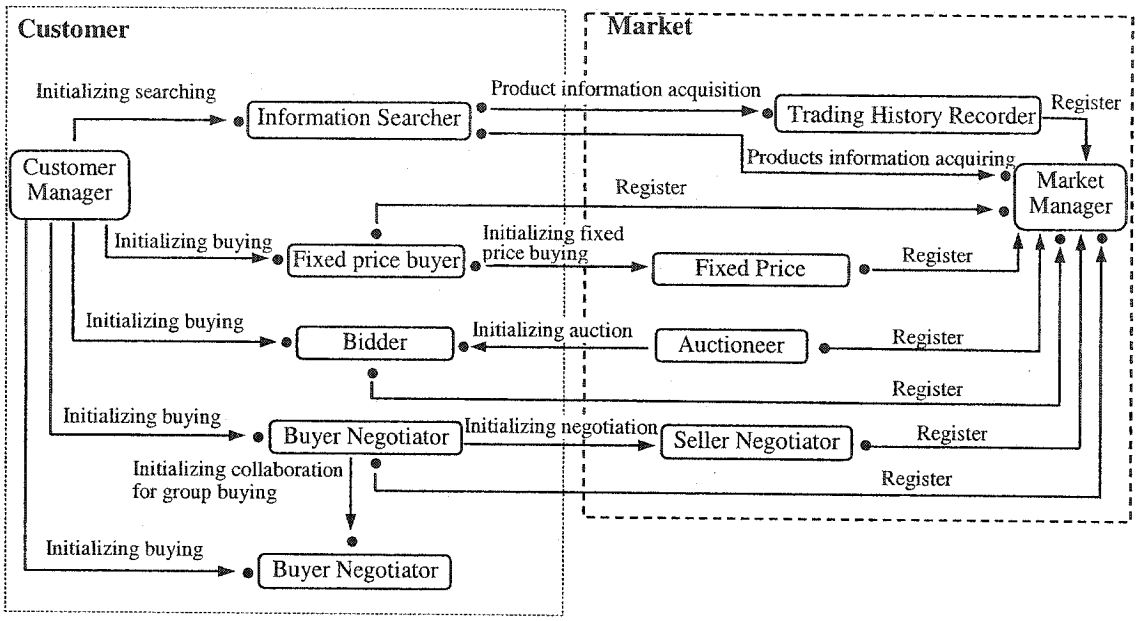


Figure 3-3 Interactions between Roles

Interaction Actions:

(1) Register

- *Description*: registration of *a*ny participant in the market.
- *Returned value:* true (success) or false.
- *Parameters:* information about identification, authority, and credit.

(2) Initializing a searching task

- *Description:* a Customer Manager assigns a searching task to an Information Searcher.
- *Returned value:* search results.
- *Parameters:* products name, products price range, and itinerary (seller's name).

(3) Product information acquisition

- *Description:* an Information Searcher asks for information from Trading History Recorder.
- *Returned value:* product information.

- *Parameters:* product name, product price range, and seller name.

(4) Initializing a buying task

- *Description:* a Customer Manager assigns buying tasks to a Buyer (Bidder, Buyer Negotiator, or Fixed Price Buyer).
- *Returned value:* buying results.
- *Parameters:* products, quantity and price ranges.

(5) Initializing fixed price buying

- *Description:* a Buyer indicates its willingness to buy some products at a special price.
- *Returned value:* acceptance/rejection.
- *Parameters:* product name and amount of purchase.

(6) Initializing auction

- *Description:* an Auctioneer posts its current price to start an auction for a product.
- *Returned value:* null.
- *Parameters:* product name and current price.

(7) Initializing negotiation

- *Description:* a Buyer Negotiator negotiates with a Seller Negotiator.
- *Returned value:* acceptance/rejection of a negotiated price.
- *Parameters:* products, quantity and negotiated price.

(8) Initializing collaboration for group buying

- *Description:* a Buyer invites other Buyers to join the group for a better discount through negotiation.
- *Returned value:* buyer name and the amount of purchase.
- *Parameters:* products and price range within which the negotiated prices may vary.

3.4 Agent Model

3.4.1 Make Agents to Play Roles

Role models are the basis of the design of agent systems. Basically, the roles that an agent plays must be identified in design. The proactive behaviors of agents are role based, and can be defined based on the individual roles, knowledge, and protocols already designed. Therefore, based on role models, the proactive behaviors of an agent are simply the totality of its assigned role behaviors.

Just as in some other previous related works [4][22], when mapping roles to agent classes, there are generally a one-to-one, one-to-many and many-to-one mapping strategies between roles and agent classes. One-to-one mapping method is a simple mapping process that defines an agent to play one specific role. However, the designer may combine multiple roles in a single agent class or map a single role to multiple agent classes for the reasons of (a) reducing couplings among agents; and (b) improving the concurrency for efficient problem solving. Here, we will consider both of the above factors and try to get a reasonable balance between them.

On the one hand, it is desirable to allocate totally different roles to different agent classes when their coupling is infrequent. This may improve the system efficiency through concurrent activities of agents. For example, the roles of Market Manager and Seller (Fixed Price Seller, Auctioneer, and Seller Negotiator) have different goals and responsibilities. They need to run concurrently even if they interact sometimes.

On the other hand, since any dependency between two roles leads to a conversation between their mapped agent classes, it is recommendable for designers to combine two roles that share frequent communications. For example, the role of Trading History Recorder periodically asks for the information from Market Manager. Using one agent

class to play these two roles will decrease the exchange of messages between agents. The same reason in mapping four roles of Information Searcher, Fixed Price Buyer, Bidder, and Buyer Negotiator in a buyer class, and mapping three roles of Fixed Price Seller, Auctioneer, and Seller Negotiator in a seller class.

Figure 3-4 presents the mapping relations between roles and agent classes in the e-market. We choose four types of agent classes to play the roles captured in the above role model.

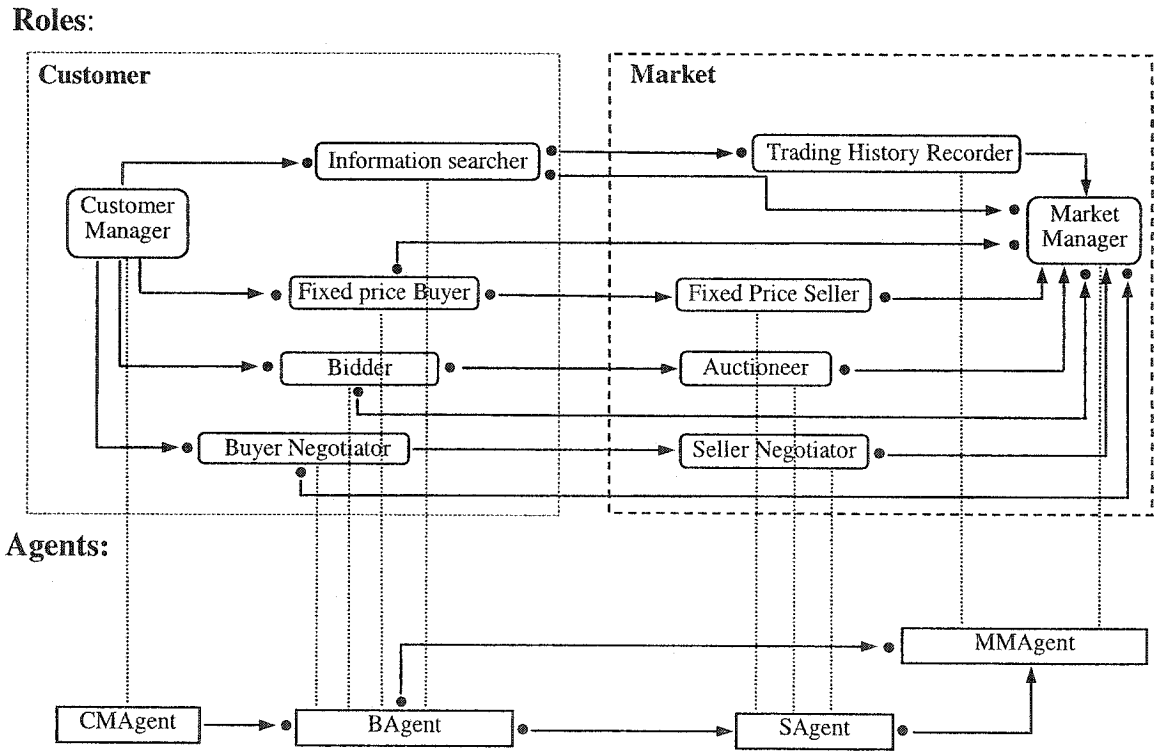


Figure 3-4 Mapping Relations between Roles and Agents

(1) *CMAgent* class plays the role of the Customer Manager.

(2) *BAgent* class is allocated four roles. When performing a buying task, a *BAgent* will first start the behavior of Information Searcher, and then choose a behavior of one of Fixed Price Buyer, Bidder, and Buyer Negotiator according to the search results by the behavior of Information Searcher. They share the buying task and the searching results through a shared internal object.

(3) *SAgent* class plays three roles. The behaviors of the roles can run concurrently and share the selling items of the seller agent through a shared internal object.

(4) *MMAgent* class is assigned two roles. The behaviors of the roles can run concurrently and share the trading records of the market through a shared internal object.

3.4.2 Agent Software Architecture

Based on TSAM, each type of agents in the e-market is designed consisting of several behaviors, among which the sensory behavior, proactive behaviors and reactive behaviors are implemented by inheriting corresponding behaviors of TSAM. In TSAM, the sensory behavior and proactive behaviors are inherited from SimpleBehavior of Jade, and the reactive behaviors are implemented by tuple space services. Figure 3-5 presents the software architecture of an e-market agent.

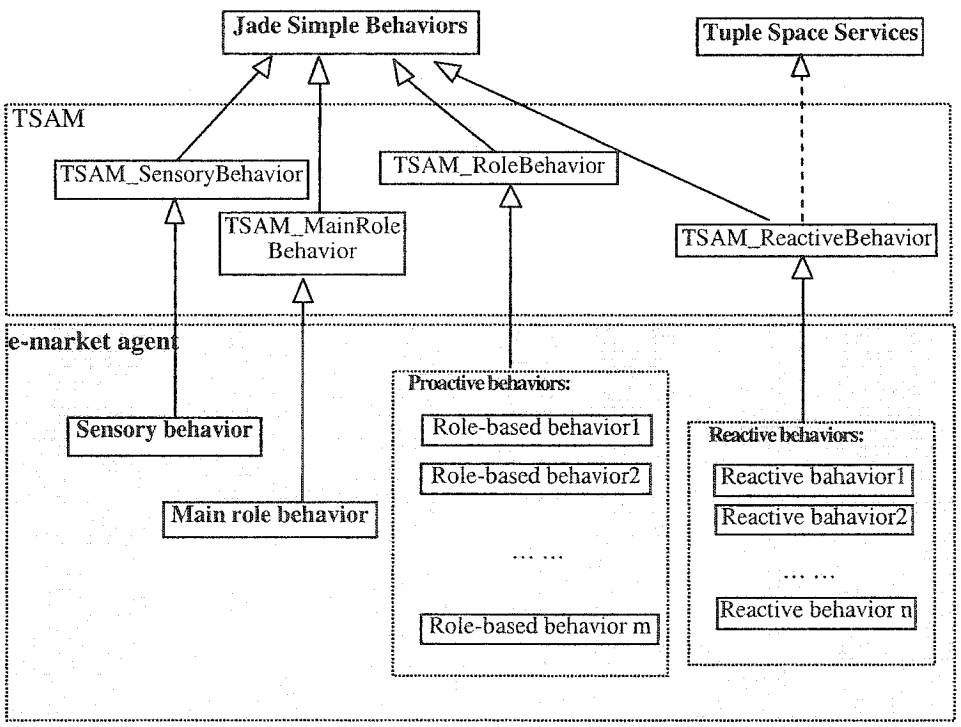


Figure 3-5 Software Architecture of an E-Market Agent

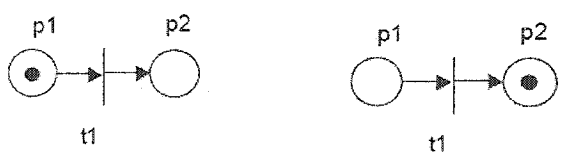
3.4.3 Agent Behavior Model

Agent behavior model includes agent inner role-based behavior protocols and agent interaction protocols. In the following section, we use Colored Petri Net (CPN) to describe agent behaviors, and especially present the interaction protocols made by agents. The objective is to show how TSAM-based agent behavior model facilitate the design of flexible behavior protocols for dynamic couplings among agents.

3.4.3.1 Colored Petri Net (CPN)

Effective modeling of complex concurrent systems requires a formalism that captures essential properties such as non-determinism, synchronization and parallelism. Multi-agent systems generally are distributed systems with many complex and concurrent processes. To model the complexity of agent systems clearly, we select CPN to describe the agent behaviors.

Petri Nets [30] are a formal and graphically appealing language, appropriate for modeling concurrent, distributed, and asynchronous behaviors in distributed systems. The Petri Net graph consists of places, transitions and arcs that connect them. Input arcs connect places with transitions, while output arcs start at a transition and end at a place. The movement of tokens enables the execution of the Petri Nets. A transition is enabled when each of the input place has a token. When the transition fires, it removes tokens from each of its input places and adds some at all of its output places. Tokens are used to simulate the dynamic and concurrent activities of systems. Figure 3-6 shows how to fire a transition in a Petri Net.



Petri Net before t1 fires Petri Net after t1 fires

Figure 3-6 Petri Nets

Colored Petri Nets [12][55] are a generalization of ordinary Petri Nets, allowing convenient definition and manipulation of data values of tokens. However, the main difference between Petri Nets and CPNs are tokens. In Petri Nets, all the tokens are the same type, which are used in state transitions. To enrich the information carried by tokens, CPNs establish a distinction between the tokens in a place. A color with structured data is associated to a token. Transitions can be fired in different manners according to the different colored tokens associated with the transitions. Different tokens in the input places may enable different transitions selectively, concurrently and possibly non-deterministically. For example, in the e-market, a tuple space can be taken as a special place with many different tuples (colored tokens) shared by agents. These tuples may enable different transitions (agent behaviors) concurrently, and as a result, change the system's state.

3.4.3.2 Agent Behavior Protocols

(1) Dynamic Information Sharing

Dynamic Information sharing is used in many scenarios of the e-market. The case of Public Special Sale is a representative use of dynamic information sharing, where all the buyer agents share the special sale information, and buy (take out) the products from the market whenever they want. The buying actions of the buyer agents are concurrent with non-deterministically modifying the shared product tuples. In Figure 3-7, we describe the behaviors of buyer agents and seller agents in Public Special Sale by CPN.

In the CPN, the big circle (place) represents a tuple space, as a shared information area with different tuples (colored tokens). The tuple space is created by a market manager agent and is shared by the participants (ABuyer/ASeller agents). In the diagram, the net is 'dynamic' as ABuyer/ASeller agents may be added/deleted at different time in Public

Special Sale. The tuples in the tuple space includes: (I) product information (pi) publicized by seller agents, (II) public special sale information (ps), (III) available number (an) modified by buyer agents, and (IV) buying result (br) placed by buyer agents.

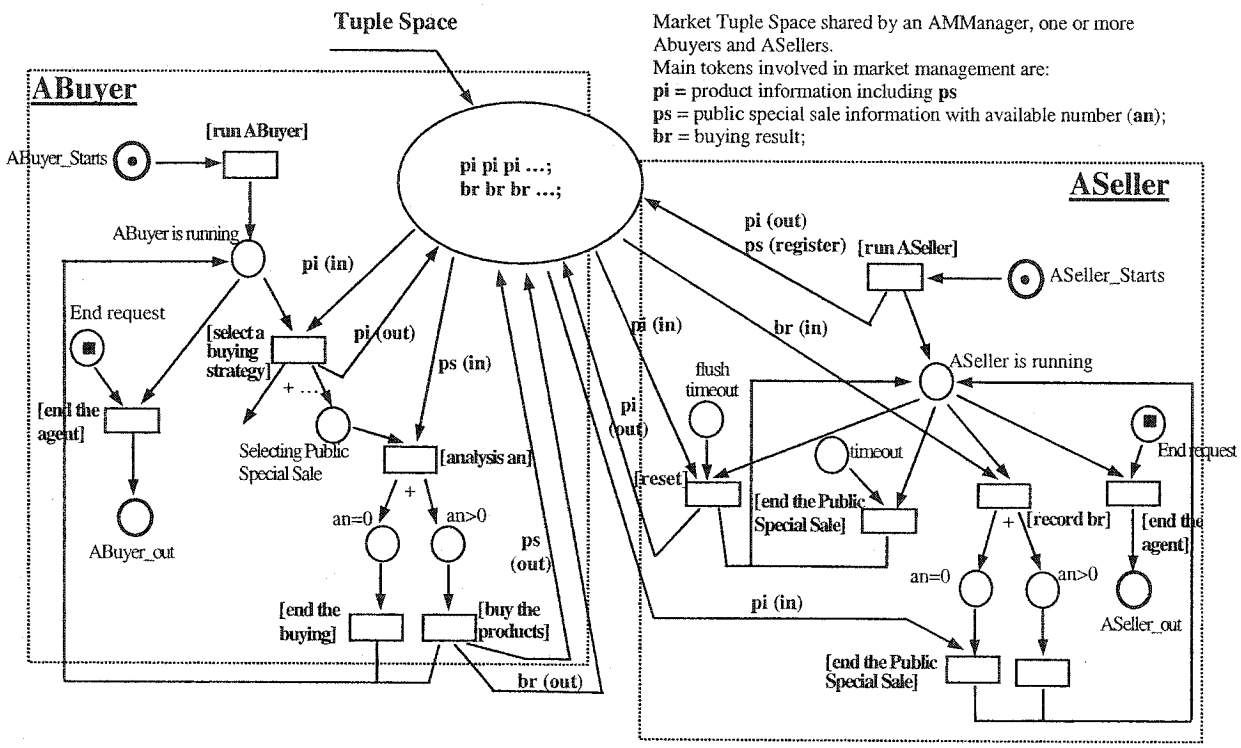


Figure 3-7 Agent Behavior Model for Public Special Sale

A ABuyer Agent in Public Special Sale

A ABuyer agent starts at running state. It may search product information on the tuple space by reading product information tuples and select to move to one of different states according to the buying strategy it selects. If the ABuyer agent decides to buy a product through Public Special Sale, it will be in the state of 'selecting Public Special Sale'. At this time, the agent will extract the corresponding public special sale information (ps) from the tuple space and move into one of two states. In one of the states, the available number is zero (an=0), in which case, the agent will end the buying and move to the running state for the next buying selection. In the other state, when the available number

is not zero (an>0), the agent will buy the product, change the available number on tuple space (ps(out)) and inform the seller agent the buying result (br(out)). Each buyer agent does not consider whether the other buyer agents buy the same products at the same time. TSAM supports first come and first served among all buyer agents.

A ASeller Agent in Public Special Sale

When a ASeller agent starts, it will proceed to do two things. One is to publicize its product information (pi) including Public Special Sale information (ps). The other is to register a reactive tuple (br) in the tuple space so that it can react to the reactive tuple of buying results (br (register)). Whenever a buyer agent buys a products and inserts a buying result tuple (br) in the tuple space, the seller agent reacts to it by removing the buying result tuple (br) from the tuple space, recording it and moving to the running state until the available number is zero. In the latter case, it will end the Public Special Sale for that product.

A ASeller agent may set a time for the special sale of some products. After the time, it will stop the Public Special Sale. A ASeller agent may also set a flush time for resetting the sale information of all its sold products by removing all relevant product information from the tuple space (pi (in)) and inserting new product information into it (pi (out)).

(2) Collaborations among Agents

Collaborations are the key for the agents in a group to move forward in their respective goals. A transaction can't be realized if collaborations among agents are not supported. In Figure 3-8, a CPN of group buying describes the collaboration of agents in a group buying.

In the CPN, the big circle represents a tuple space that stores different tuples corresponding to: (I) product information (pi) publicized by ASeller agents, (II)

invitation request (ir) publicized by the buyer coordinator, and (III) invitation answer (ia) placed by any buyer who wants to join the group. The tuple space is created by a customer manager agent and is shared by its participants (ABuyer agents). In the diagram, the net is 'dynamic' as any buyer agent may enter/quit the group purchase at different time.

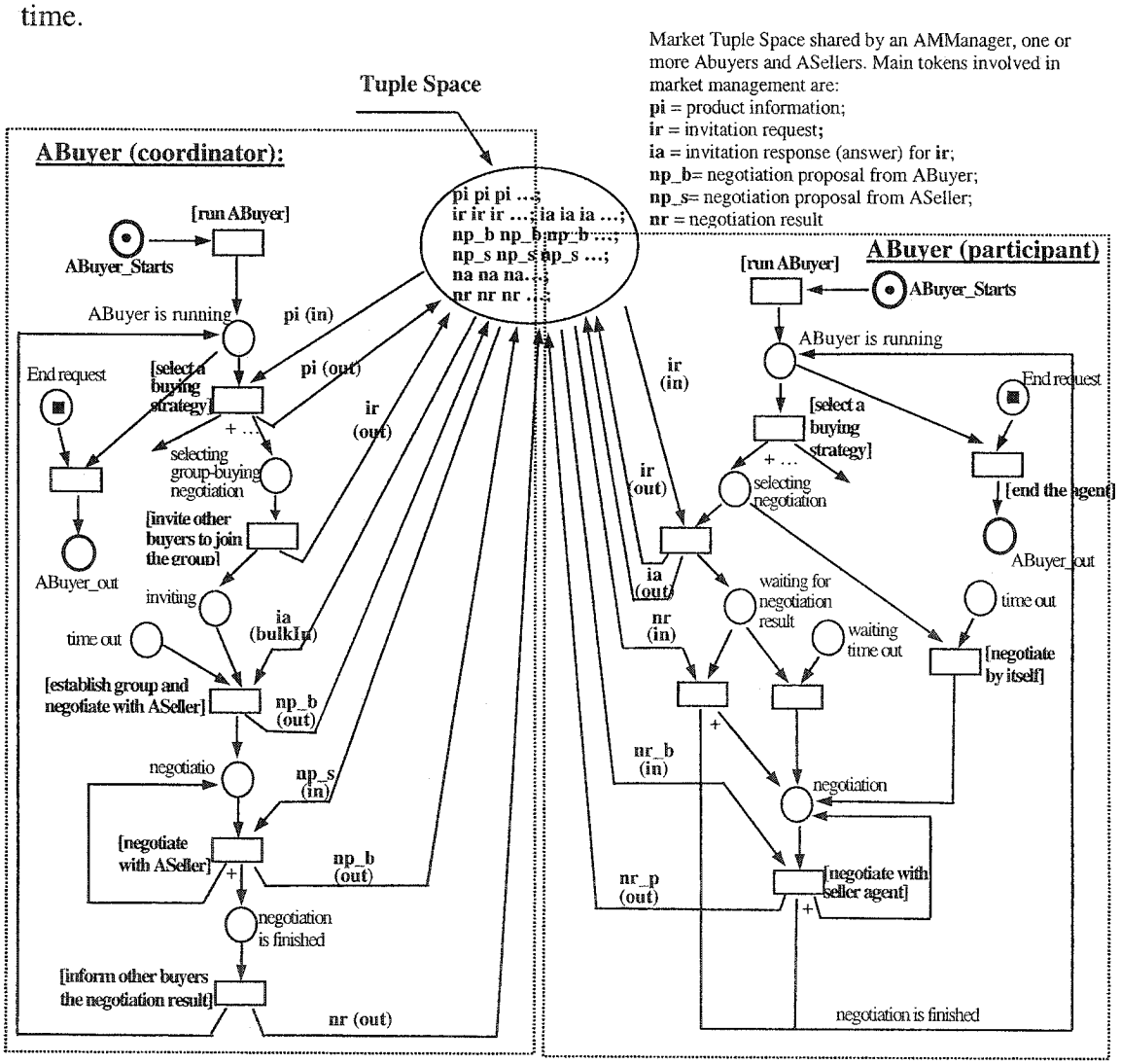


Figure 3-8 Agent Behavior Model for Group-Buying Negotiation

A ABuyer Agent (Coordinator) in a Group Buying

A ABuyer agent starts at the running state. Whenever there is product information (pi) in the tuple space, and if an ABuyer agent decides to buy through group buying, it can become a coordinator to establish a group of buyers before it negotiates with a seller

agent. As a coordinator, the buyer agent will invite other buyers to join the group buying by inserting an invitation request tuple into the tuple space (ir (out)). The coordinator does not know who will join the group in advance. When the waiting time is over, the coordinator agent will collect all the invitation answer tuples from the tuple space (ia (bulkIn)) to establish a group. Then, as a representative of the group, the coordinator agent will negotiate with a seller agent and inform all the participant agents when the negotiation is finished. Then it moves to the running state again.

A ABuyer Agent (Participant) in a Group Buying

Alternatively, a buyer agent may choose to look for a group by searching for invitation requests for a product from a tuple space. If it does not find any invitation request within a period of time, it will negotiate with the seller agent by itself. If it finds a proper invitation request, it may join the group by inserting an invitation answer tuple (ia (out)). Then it retrieves the negotiation result from the tuple space.

When the waiting time is over and the negotiation result tuple is not found, the buyer agent will negotiate with the seller agent by itself. The coordinator agent should be responsible for this situation of too long time of negotiation. If the buyer agent gets the negotiation result within the waiting time, it will move to one of two states: (a). if the negotiation is successful, the buyer agent will move to the running state again; (b) otherwise, it has to negotiate with the seller agent by itself and move to the negotiation state. When the negotiation ends, the buyer agent will return to the running state again.

(3) Dynamic Relationships among Agents

In many transactions, the relations among agents are dynamic and can be changed at run time. Here we select an English auction to show how TSAM easily supports the design of

dynamic collaboration among agents. The CPN of Figure 3-9 describes the agent behaviors in an English Auction.

In Figure 3-9, the big circle represents a tuple space containing: (I) product information (pi), (II) the current bid (cb) publicized by the auctioneer, (III) proposed bids (pb) placed by bidders and (IV) the auction commitment (ac) placed by the auctioneer. The tuple space is created by a market manager agent and is shared by all its participants. The interaction protocols are different from the FIPA English auction protocols, where the auctioneer is responsible for gradually raising the price and broadcasting the current price to all the participants. Here, the bidders are allowed to enter or quit an auction at any time and the auctioneer does not know the participants in advance. Therefore, in the diagram, the net is 'dynamic'. Any bidders can be added/deleted dynamically.

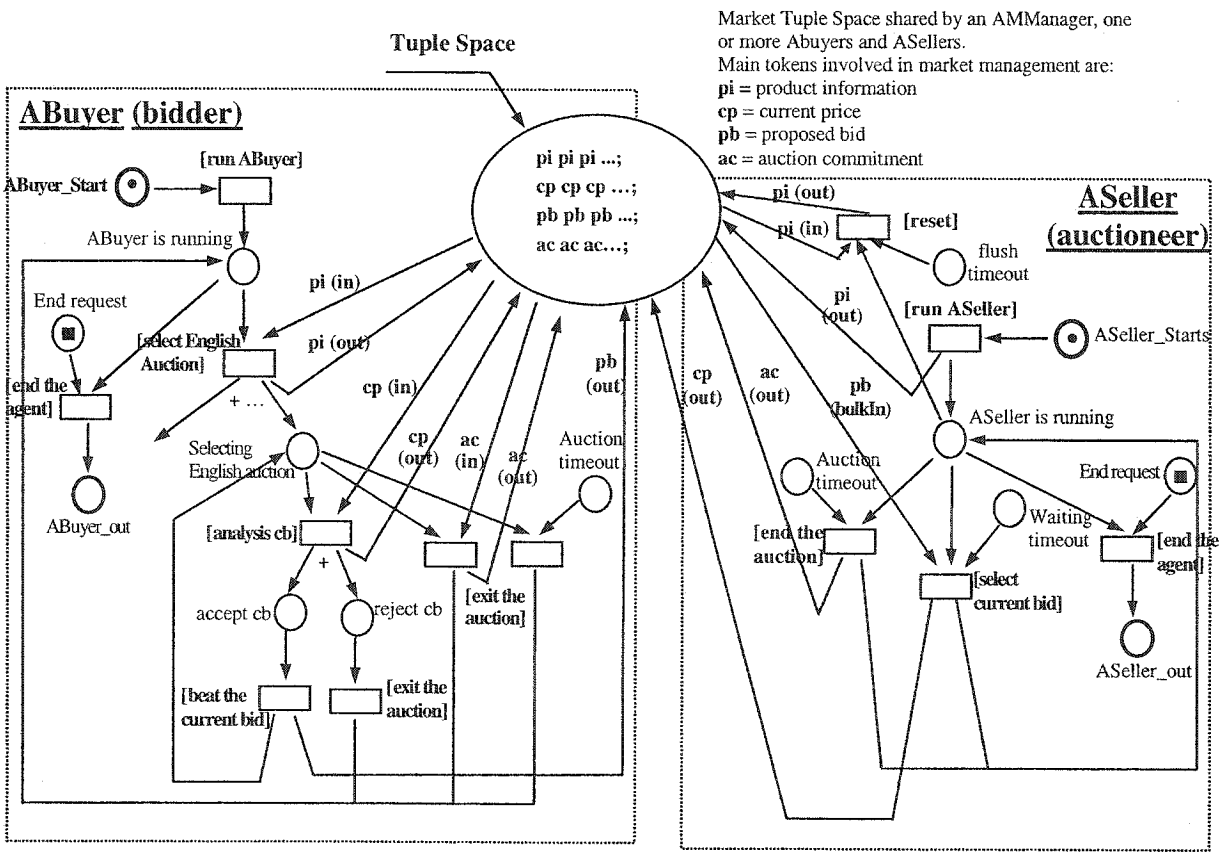


Figure 3-9 Agent Behavior Model for English Auction

ASeller Agent (Auctioneer) in English Auction

First, when an ASeller agent starts, it will publicize its product information (pi), which includes current price of a specific product in an English Auction (cp) by inserting a product tuple into the tuple space (pi (out)). In fact, it initializes an online auction as an auctioneer, but still does not know who will enter the auction beating the current bid price. Any bidder can beat the current price by proposing a higher bid. Whenever the cycle time is over, the auctioneer will extract all the proposing bids from the tuple space (pb (bulkIn)) and select the highest bid as the current price. When the auction time is closed, the auctioneer agent will select a winner with the best (highest) price and inform all the bidders the auction result (ac (out)). Then it closes the auction and moves to the running state for selling other items.

A ABuyer Agent (bidder) in English Auction

A bidder agent looks for appropriate product information in the tuple space. If it decides to buy a product through an on-line Auction, it may read the current price from the tuple space, analyze it and plan the next step. It can propose a higher bidding price and move to the English Auction state. It can also decide to give up and move to the running state again. When the bidder agent proposes its bidding price, it can obtain an auction commitment and become the winner, or fail to become a winner. In either case, the buyer agent moves back to the running state to do other possible activities.

(4) Reactivity of Agents

Tuple space based coordination model provides reactive tuples that allow an agent to react to asynchronous notifications from other agents as part of the collaboration with other agents. This type of protocols is easily designed under TSAM. For example, during the Public Special Sale, a seller agent may register a reaction with a reactive tuple of a

buying result. When a buyer agent finishes buying a product through Public Special Sale, it will write a reactive tuple of a buying result into the tuple space. Then, the seller agent reacts to the availability of the reactive tuple by activating its relevant registered reactive behavior to record this transaction. Figure 3-7 is the CPN of Public Special Sale, where the state transition of a seller agent is shown. The reactive mechanism of tuple space can improve the flexibility of collaborations among agents and make the design process simpler.

3.5 Conclusion

TSAM simplifies the design of an e-market application. In particular, role analysis and role mapping to agents provide the designer clear separation of concern in designing efficient collaboration protocols among agents. At the same time, the availability of tuple space as an interaction medium facilitates the dynamic information sharing and hence non-deterministic progress of the application. As non-determinism promotes performance (relative to static scheduling), the resulting e-market can achieve results not obtainable by static synchronous protocols.

Chapter 4 Implementation of the E-market Application

In this chapter, we present the implementation of the e-market multi-agent system. The objective of this chapter is to show how TSAM facilitates the implementation of e-market agents. To present the context of our work clearly, section 4.1 describes the system architecture of the e-market application. Section 4.2 introduces JADE platform through which we implement the e-market application. In section 4.3, we show how tuple space services are used to support programming flexible and efficient agent interaction protocols in the e-market application. Finally, section 4.4 illustrates how TSAM supports the implementation of agent behaviors.

4.1 System Architecture

To provide services for consumers and vendors to perform trading transactions, our e-market system is mainly composed of four types of agent objects: Customer Manager Agent (CManager), Buyer Agent (ABuyer), Market Manager Agent (MManager) and Seller Agent (ASeller). Figure 4-1 is a graphical representation of the system architecture. It is a distributed multi-agent system, where agents are separate entities running concurrently, and are possibly distributed on different nodes.

A CManager is a mediator between consumers and buyer agents (ABuyers). It receives purchase requests from consumers, assigns buying tasks to buyer agents and returns the purchase results to the consumers. In the e-market, stores are basic unit represented by a seller agent (ASeller). Some stores are located in the same location called 'mall' to give buyers more convenience of searching and buying. A MManager is responsible for management of such a mall (local market). Generally, malls are distributed on different locations, so MManager are distributed. The relationship between a MManager and a

ASeller is logical. As a participant of more than one mall, a ASeller may register to more than one MManager.

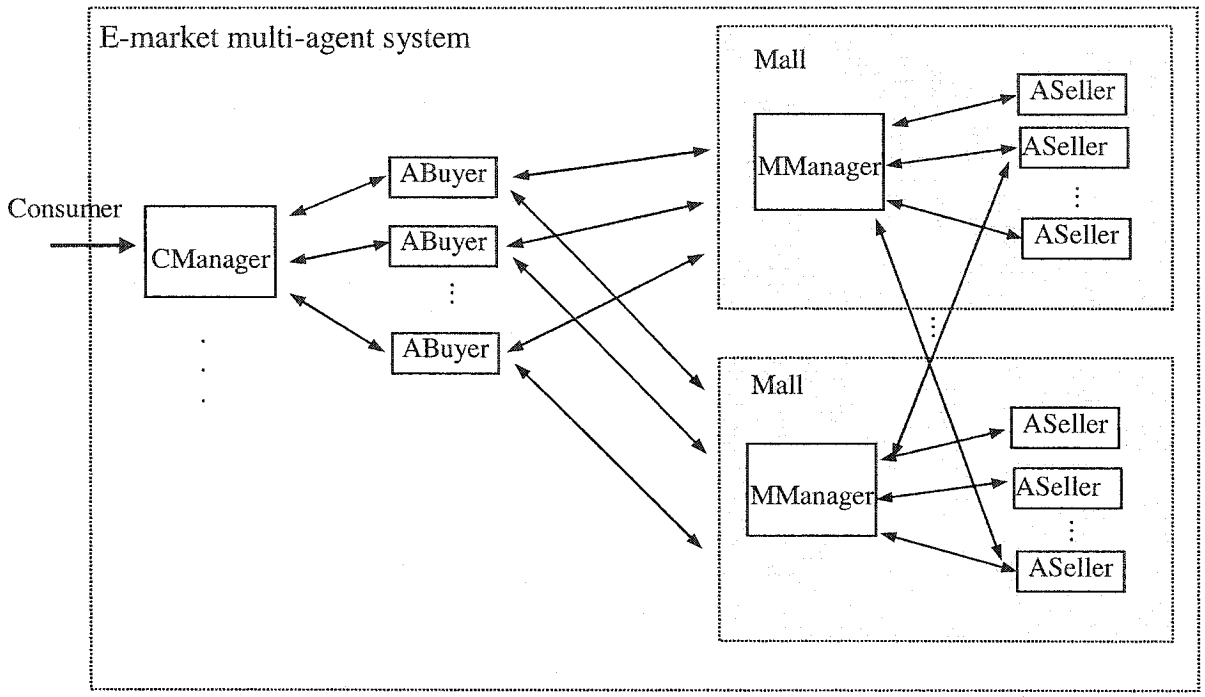


Figure 4-1 System Architecture

The architecture of the e-market is tuple space based communication system. Logically, as a shared space, one tuple space can be used by a group of agents with collaboration relations. However, the e-market is a distributed system, where multiple agents execute concurrently in different nodes, and there are different sets of tuples shared by different group of agents. Hence, physically more than one tuple space may improve the concurrency and enhance the efficiency of matching the associated tuples. In the e-market, each CManager has one tuple space shared with its registered buyer agents, and each local market shares a separate tuple space with all its participants. A buyer agent can access different tuple space to interact with seller agents. The system architecture of using tuple space is shown in Figure4-2.

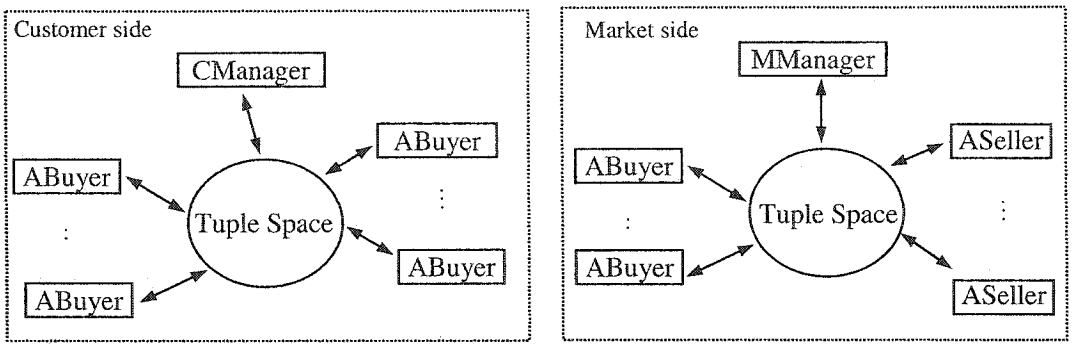


Figure 4-2 Use of Tuple Space in the E-Market

4.2 JADE Platform

As TSAM is developed on top of JADE (Java Agent Development Framework), JADE plays an important part of the e-market. JADE is a software development framework aimed at developing multi-agent systems conforming to FIPA standards. It includes two main products: a FIPA-compliant agent platform and a developing package to develop Java agents. We briefly review the key features of JADE in the following.

Execution Platform:

JADE offers a distributed agent platform that can be split among several hosts provided they can be connected via RMI. Agents are implemented as Java threads and live within Agent Containers that execute on the agent platform. A container of agents provides a complete run time environment for agent execution and allows several agents to concurrently execute on a same host.

Developing Package:

The developing package includes an agent foundation class for creating customized agents, a library of protocol skeletons for tailoring agent conversation and a suite of development tools. The tools provide runtime agent management, directory service,

message exchange debugging, agent life-cycle control, and a conversation monitor tool that graphically displays agent interactions in the form of a separate diagram.

Agent Behavior Model:

JADE supports the execution of multiple, parallel and concurrent agent activities via the behavior model. It uses one thread per agent instead of one thread per behavior to limit the number of threads running in the agent platform. A scheduler, hidden to the developer, schedules the agent behaviors in a non-preemptive way.

FIPA-Compliant Communication:

JADE supports the FIPA standard interaction protocols to build agent conversations. It provides ready-made behavior classes for agent interactions following most of the FIPA specified interaction protocols.

In the implementation of our e-market system, we extend the basis JADE agent behavior classes for the definitions of TSAM agent behaviors, but use tuple space to implement agent couplings. Tuple space does not provide predefined interaction protocols like FIPA. However, the coupling primitives provided by tuple space support the implementation of more flexible interaction protocols. Section 4.3 and section 4.4 will present the details of how the agent behaviors and agent interaction protocols are easily implemented via TSAM.

4.3 Tuple Space based Agent Coupling Primitives

The services of tuple space provide basic means of agent couplings. Table 4-1 outlines the set of primitives of tuple space services, where a *Tuple* is an ordered collection of *Fields* each of which has a type and a value associated with it, and a *Template* is similar to a *Tuple* except that its *Fields* do not necessarily have values associated with their types.

```
//tuple space management permitives
TupleSapceID   TSCreate(String name);
TupleSapceID   TSFind(String name);

// synchronous single tuple access permitives
void    out(TupleSapceID tsId, Tuple tuple);  //write tuple into tuple space
Tuple   in(TupleSapceID tsId, Tuple template); //extract tuple from tuple space
Tuple   in(TupleSapceID tsId, Tuple template, long timeout);
Tuple   read(TupleSapceID tsId, Tuple template); //read tuple from tuple space
Tuple   read(TupleSapceID tsId, Tuple template, long timeout);

// synchronous bulk access permitives, which manipulate more than one tuple at a time
TupleSet  bulkInWithoutWait(TupleSpaceID tsID, Tuple template);
TupleSet  bulkReadWithoutWait(TupleSpaceID tsID, Tuple template);

// logic-template-based synchronous access permitives
TupleSet   in(LogicTemplate logic_template);
TupleSet   in(LogicTemplate logic_template, long timeout);
TupleSet   read(LogicTemplate logic_template);
TupleSet   read(LogicTemplate logic_template, long timeout);

//logic-template and bulk based synchronous access permitives
MultipleTupleSet  bulkInWithoutWait(LogicTemplate logic_template)
MultipleTupleSet  bulkReadWithoutWait(LogicTemplate logic_template)

//asynchronous single tuple access permitives
void     asynOut(TupleSapceID tsId, Tuple tuple);
Future   asynIn(TupleSapceID tsId, Tuple template);
Future   asynRead(TupleSapceID tsId, Tuple template);

//asynchronous bulk acces permitivess, which manipulate more than one tuple at a time
Future   bulkAsynIn(TupleSapceID tsId, Tuple template);
Future   bulkAsynRead(TupleSapceID tsId, Tuple template);

// logic-template-based asynchronous access permitives
Future   asynIn(LogicTemplate logic_template);
Future   asynRead(LogicTemplate logic_template);

//logic-template and bulk based asynchronous access permitives
Future   bulkAsynIn(LogicTemplate logic_template)
Future   bulkAsynRead(LogicTemplate logic_template)

// reactive tuples permitives
RegisterID   register(TupleSapceID tsId, Tuple template, ReactionRef ref);
RegisterID   register(TupleSapceID tsId, EventChecker checker, ReactionRef ref);
void         deregister(RegisterID register_id);
```

Table 4-1 Tuple Space Primitives

These primitives can be classified into four categories: management primitives, synchronous access, asynchronous access and reactive (notification) primitives. The management primitives are used to create tuple spaces or find existing tuple spaces. The synchronous and asynchronous access primitives are used to access tuples. A

synchronous primitive blocks the process until a tuple is available, whereas, an asynchronous primitive allows an agent to access tuples without being blocked if the results are not available. Reactive primitives facilitate spontaneous couplings between agents in reactive behaviors.

In order to facilitate programming, the access primitives can be classified into three sub-categories: (i) single access, (ii) bulk access and (iii) logic-template-based access. Bulk access allows multiple tuples to be retrieved in a single operation. Logic-template-based access provides associative search of tuples via a logic-template to improve the flexibility of agent collaboration protocols.

The access of tuple spaces includes three main types of operations: (i) 'in' operations extract tuples from tuple space, (ii) 'out' operations write tuples into tuple spaces and (iii) 'read' operations read tuples from tuple spaces without withdrawal of them. We elaborate a subset of the primitives in the following part.

in(TupleSapceID tsId, Tuple template, long timeout) is a synchronous primitive to retrieve a tuple that matches the template in a tuple space. The agent will block until a matched tuple is available or timeout is triggered.

asynIn(TupleSapceID tsId, Tuple template) is an asynchronous primitive to retrieve a tuple that matches the template in a tuple space. It returns a tuple-holder object (called 'future'), which can be checked later by the agent. This enables an agent to asynchronously interact with another agent without stalling when the latter is not ready.

register(TupleSapceID tsId, Tuple template, ReactionRef ref) primitive registers a reaction (ReactionRef ref) to a tuple space. If a tuple that matches the template becomes available, the reaction referenced by 'ref' is triggered. This primitive enables the reactive behaviors of an agent to react to stimulus from other agents.

MultipleTupleSet bulkInWithoutWait(LogicTemplate logic_template) is an access primitive with bulk and logical-template facilities. It returns a set of tuples that match the logical-template involving more than one tuple space.

Based on the tuple space primitives, we implement agent interaction protocols in the e-market system. We choose the following aspects to show how the implementation of flexible agent interaction protocols is simplified using the above tuple space services than otherwise using message passing.

(1) Dynamic Sharing

Tuple space supports the concurrent and non-deterministic modifications of the shared tuples with simple operations ('out', 'read' and 'in'). For example, in Public Special Sales, a set of product tuples with Public Special Sales information in a tuple space are shared by all buyer agents. Any buyer agent can buy the product through non-deterministically removing ('in') the product tuple from the tuple space. Figure 4-3 illustrates the code fragment of a buyer agent in Public Special Sale.

In the implementation of a buyer action in Public Special Sale, three main steps are to be taken. First, it constructs a tuple template with fields and associated values of seller agent name, item name, product information and price information. Then, it retrieves a tuple that matches with the constructed tuple template from the tuple space. Because the buying actions of buyer agents are concurrent, the shared tuples can be removed non-deterministically by any buyer agent. Hence, the returned tuple may be a matched tuple or null. If the returned value is null, the buyer agent fails in the Public Special Sale; otherwise, the buyer agent can proceed with the third step successfully and informs the seller agent the buying result by inserting a reactive tuple into the tuple space. If the required quantity of the product is less than the amount available, the buyer agent must

send back the remaining quantity of the product to the market by constructing and sending a remain_tuple that is the same as the takeout_tuple except a changed amount of the product in the field of ProductInfo.

```
// actions of a buyer agent in a Public Special Sale
//step1: contructs a tuple template for Public Special Sale
  Tuple takeout_tuple=new Tuple();
  Field to_f1=new Field(TupleType.SELLER_PRICEINFO);
  Field to_f2=new Field(sellerAgent);
  Field to_f3=new Field(itemName);
  Field to_f4=new Field(ProductInfo);
  Field to_f5=new Field(PriceInfo);
  takeout_tuple.Add(to_f1);
  takeout_tuple.Add(to_f2);
  takeout_tuple.Add(to_f3);
  takeout_tuple.Add(to_f4);
  takeout_tuple.Add(to_f5);

//step2: takes out the tuple from the tuple space, and return immediately (1 ms waiting time)
  Tuple return_tuple=myAgent.in(tsId, takeout _tuple,1);

//step3:informs the seller agent by inserting a reactive tuple and put back the tuple with the
remain amount available for further buying
  if (return_tuple!=null) {
  Tuple toseller_tuple=new ReactiveTuple();
  Field ts_f1=new Field(TupleType.BUYER_PUBLICSALE);
  Field ts_f2=new Field(sellerAgent);
  Field ts_f3=new Field(itemName);
  Field ts_f4=new Field(myAgent.getLocalName()); //buyer agent
  Field ts_f5=new Field(buyingPrice);
  Field ts_f6=new Field(buyingNum);
  toseller_tuple.Add(ts_f1);
  toseller_tuple.Add(ts_f2);
  toseller_tuple.Add(ts_f3);
  toseller_tuple.Add(ts_f4);
  toseller_tuple.Add(ts_f5);
  toseller_tuple.Add(ts_f6);
  myAgent.out(tsId,toseller_tuple); //inserts a reactive tuple
  }
  if (remain-number>0) {
  Tuple remain_tuple=new Tuple();
  … …
  myAgent.out(tsId,remain_tuple);
  }
  … …
```

Figure 4-3 Code Fragment of a Buyer Agent in Public Special Sale

Tuple space makes the implementation of dynamic sharing among agents easier than message passing. Using message passing, a seller agent has to define a separate behavior to receive and deal with the buying requests from the buyer agents, and it has to consider the coordination among the behaviors for data consistency. As a result, a seller agent may become a bottleneck in the transaction. Tuple space relieves the seller agent from this work, and provides application developers simple and intuitive ways via dynamic sharing.

(2) Global Information Sharing

Tuple spaces can facilitate global information sharing for agents. The global information includes up-to-date information and persistent information. For up-to-date information, such as "global state" or other global information, they are available to all agents. For the persistent information (storage), like the results of previous calculations, they can be easily shared. Any agent who wants to post its information only uses 'out' operation to write tuples into a tuple space without indicating the recipient agents. Any agent who wants to get the global information only uses 'read' operation without knowing the exact one who modifies the information.

For example, in a group auction of the e-market, a buyer agent may join the auction any time. Often modified by an auctioneer, the on-line bidding price and the available amount of the product are shared by all the bidders. Figure 4-4 shows the implementation of a group auction. When a buyer agent wants to join a group auction, it constructs a reactive tuple of group auction request in the tuple space. As a result, the reactive tuple will trigger a reactive behavior of the seller agent (auctioneer) to receive the bidding request and publicize a changed bidding price through putting a price information tuple into the tuple space.

```
// actions of a buyer agent in a group auction
  Tuple gaRequest_tuple=new ReactiveTuple(); // constructs a reactive tuple of group
  auction request
  Field gaf1= new Field(TupleType.BUYER_GROUPAUCTION);
  Field gaf2= new Field(sellerAgent);
  Field gaf3= new Field(itemName);
  Field gaf4= new Field(buyerAgent);
  Field gaf5= new Field(quantity);
  gaRequest _tuple.Add(gaf1);
  gaRequest _tuple.Add(gaf2);
  gaRequest _tuple.Add(gaf3);
  gaRequest _tuple.Add(gaf4);
  gaRequest_tuple.Add(gaf5);
  myAgent.out(tsId, gaRequest_tuple); // joins the group auction

   // waits for the result through reading the group auction result tuple
    ... ...

   //reactions of a seller agent in the groupAuction
  String itemName=(String)(reactiveTuple.GetField(3)).getValue();
  String buyerName=(String)(reactiveTuple.GetField(4)).getValue();
  int buyingNum=((Integer) (reactiveTuple.GetField(5)).getValue()).intValue();
  int index=siTable.getCurrentIndex(itemName);
  priceInfo pri=siTable.getPriceInfo(index);
  sellingPrice=gaStrategy.getGroupPrice(pri. getCurrentNum()+buyingNum);

  // does changes on price information
  pri.setAvailableNum(availableNum-pri.getAvailableNum());
  pri.setCurrentPrice(sellingPrice);
  pri.setCurrentNum(currentNum);
     ... ...
  myAgent.out(tsId, pri_tuple); // posts the new price information
     ... ...
```

Figure 4-4 Code Fragment of Agents in Group Auction

If we use message passing to implement the group auction, some extra work has to be done. From the market side, a separate mailbox manager agent is needed to be responsible for broadcasting the price information to all the buyer agents. This adds the burdens of considering explicit or at least implicit addressing knowledge of buyer agents to programmers. From the purchase side, each buyer agent has to communicate with the auctioneer asking for the price information before it enters a group auction. This may cost extra time.

(3) Asynchronous Access

Tuple space provides an asynchronous coupling mechanism that does not enforce synchronization between the coupling partners. For example, the asynchronous access primitive, 'asyIn()', allows an agent to retrieve a tuple without being blocked In the e-market, when a seller agent sells its products through negotiation, it may use 'asyIn()' operation to retrieve the result from a buyer agent. The use of the asynchronous operation enables a seller agent to asynchronously interact with a buyer agent without waiting for the latter to become ready.

Figure 4-5 describes the use of asynchronous operation for a seller agent in negotiation. When a seller agent receives a negotiation request from a buyer agent, it starts to negotiate with the buyer agent. When the seller agent sends back a new negotiated price in the form of a negotiation-tuple, it will asynchronously retrieve the response-tuple from the buyer agent via the operation of 'asynIn()'. The returned value 'fr' is held by the tuple space. The seller agent can check the availability of 'fr' when needed. If the fr is available, the seller agent will proceed with the negotiation process. Otherwise, the seller agent may proceed with its other tasks.

```
//a seller agent's actions in negotiation
... ...
Tuple negotiation_tuple=new Tuple(); //constructs an negotiation tuple
... ...
myAgent.out(tsId, negotiation_tuple); //sends back the negotiated price to the buyer agent

Tuple response_tuple=new Tuple(); //constructs an response template
... ...
Future fr = myAgent.asynIn(tsId, response_tuple); //asynIn the response tuple

if (fr.isAvaliable()) { // checks if the response tuple is available
Tuple answer_tuple=fr.getSingleTuple(); //gets the answer tuple from fr
        ... ...
}
else block (2000);
... ...
```

Figure 4-5 Code Fragment of a Seller Agent in Negotiation

In contrast to message passing, tuple space server holds a separate thread for each of asynchronous access and keeps the context of each of the accessed tuple until it is available and is fetched by relevant agents. This simplifies the code of asynchronous couplings in the application.

(4) Bulk and Logic-Template based Access

Bulk access primitives allow multiple tuples to be retrieved in a single operation. Logical-template is a concept that defines logic operations to match multiple requirements on multiple tuple spaces to promote flexibility of associative search. The bulk and logic-template based access primitives support programmers to build more efficient codes in agent couplings.

For example, in the e-market, when a buyer agent wants to search for product information of a particular product from distributed marketplaces, the retrieved multiple tuples that match the requirements (e.g. product name, and price range) from distributed tuple spaces can be returned in a single operation of 'bulkReadWithoutWait (LogicTemplate)'. The search process is shown in Figure 4-6. In the code fragment, there are two market places associated with two separate tuple spaces (tsId1, and tsId2). A buyer agent wants to search for a product from these two tuple spaces. First, it constructs a search tuple with search requirements. Then it constructs two search templates (template1 and template2) for searching the product from each of the tuple spaces. A logic template combining the two templates is created using logic operator 'and'. The semantic of 'and' operator is that it supports retrieval of 'n' tuple spaces. Finally, a bulk synchronous access operation is used to perform the actual search. The returned future object contains a set of tuples.

```
//constructs a search tuple for information searching
Tuple search_tuple=new Tuple();
Field search_f1=new Field(TupleType.SELLER_PRICEINFO);
Field search_f2=new Field(String.class); //seller agent name
Field search_f3=new Field(String.class); //item name
Field search_f4=new Field(pro); //product information
Field search_f5=new Field(PriceInfo.class); //price information
search_tuple.Add(search_f1);
search_tuple.Add(search_f2);
search_tuple.Add(search_f3);
search_tuple.Add(search_f4);
search_tuple.Add(search_f5);

//constructs different search templates for different tuple spaces
Template template1= new Template (tsId1, search_tuple);
Template template2 = new Template (tsId2, search_tuple);

//constructs a logic template
LogicTemplate sLogicTemplate = new LogicTemplate();
sLogicTemplate.and (template1, template2);

//searches the logic tuples through bulk primitive
MultipleTupleSet multiTSet = myAgent.bulkReadWithoutWait (sLogicTemplate);
        ... ...
```

Figure 4-6 Code Fragment of a Buyer Agent in Information Search

The above use of bulk and logical template based access primitive provides efficiency and programmability in the implementation of information retrieval process. When using message passing to perform the same function, at least two things are needed: (1) a set of 'send ()' and 'receive ()' operations are used for searching each of the market places; (2) extra burdens are considered for managing the receiving channel of each of the agents. Hence the codes of using message passing will be more complex.

(5) Reactive Primitives

Reactive primitives are yet another concept introduced in tuple space based coupling mechanism that we use to implement asynchronous notification couplings among agents. Figure 4-7 shows an example of how a seller agent registers a reactive tuple through the 'register' primitive in Public Special Sale. First the seller agent constructs a reactive tuple of a buying request for Public Special Sale. Then it defines a reactive behavior to deal

with a reaction of the reactive tuple. Finally it associates the reactive tuple with the reactive behavior through 'register (tsId,reactive_tuple,RB)' in the tuple space. When a buyer agent buys a product through Public Special Sale, it will write a reactive tuple of buying request in the tuple space. At this time, the associated reactive behavior (RB) of the seller agent will be activated to record and confirm this purchase.

```
//creates a reactive tuple of a buying request forPublic Special Sale proposed by buyer agents
Tuple reactive_tuple=new Tuple();
Field rf1=new Field(TupleType.BUYER_PUBLICSALE);
Field rf2=new Field(this.getLocalName()); //seller agent
Field rf3=new Field(String.class); //itemname
Field rf4=new Field(String.class); //buyer agent
Field rf5=new Field(Double.class); //buying price
Field rf6=new Field(Integer.class); //buying number
reactive_tuple.Add(rf1);
reactive_tuple.Add(rf2);
reactive_tuple.Add(rf3);
reactive_tuple.Add(rf4);
reactive_tuple.Add(rf5);
reactive_tuple.Add(rf6);

//creates a reactive behavior reacting to buying request tuples of public special sale
Seller_RectiveBehaviour_PublicSale RB=new Seller_ReactiveBehaviour_PublicSale (this);

//registers the Reactive Behavior in the tuple space
register (tsId, reactive_tuple, RB);
        ... ...
```

Figure 4-7 Code Fragment of a Seller Agent in Registering a Reactive Tuple

The asynchronous notification among agents can be programmed easily via the reactive primitive. First, tuple space can support the location transparency of agents with reactive behaviors. Programmers do not need to consider the exact locations of reactive behaviors. They can concentrate on the stimulus-response relationship among reactive tuples and reactive behaviors. Second, the reactive primitives save time for programmers to implement asynchronous notifications. If we use message passing, an additional register agent is needed for the registration and retransmission of the triggers to respective reactions of agents.

4.4 Implementation of the E-Market Agents

In the e-market, four types of agents are created (presented in Table 4-2). Based on TSAM, each agent consists of three types of behaviors. The different behaviors of each e-market agent are also listed in Table 4-2. In the following subsections, we will elaborate the implementation of each of the agent behaviors via TSAM.

Agent Name	Functionality	Proactive Behaviors	Reactive Behavior	Sensory Behavior
CMAgent (Customer Manager Agent)	Responsible for the management of buying tasks of consumers and the registration of buyer agents.	MainRoleBehavior AssignTask	AgentRegister TaskAssign BuyerAgentAbort TaskResult	SensoryBehavior
MMAgent (Market Manager Agent)	Responsible for the management of local market and the registration of the participants of the market place	MainRoleBehavior HistoryRecorder	AgentRegister PriceInfo SellerTransaction BuyerAgentAbort SellerAgentAbort	SensoryBehavior
BAgent (Buyer Agent)	As a representative of consumer, responsible for buying products through interacting with seller agents.	MainRoleBehavior CompeteForTask DirectSearch EnglishAuction GroupAuction GroupBuying Negotiation PublicSpecialSale	BuyerAgentAbort CustomerAbort MarketAbort SellerAbort	SensoryBehavior
SAgent (Seller Agent)	As a representative of a vendor, responsible for selling products through interacting with buyer agents.	MainRoleBehavior EnglishAuction Negotiation	BuyerAgentAbort MarketAbort GroupAuction NegotiationRequest PublicSpecialSale	SensoryBehavior

Table 4-2 Behaviors of E-Market Agents

4.4.1 Implementation of the SensoryBehavior of E-market Agents

An agent has its external environment. It lives in the environment and is affected by it. In the application, the external environment is implemented as an external object that may include some resources data (e.g. factors affecting the market prices) and some

unexpected accidents (e.g. troubles in systems). The task of the sensory behavior is to check the external environment and to trigger some reactions.

A sensory behavior of e-market agents can be implemented by extending the abstract class of TSAM_SensoryBehavior. Figure 4-8 describes how to create a sensory behavior of a seller agent via TSAM. A seller agent senses the external environment through checking an environment file. If it senses that an exception will cause an abort, it will trigger a reaction to stop all its transactions and inform other agents of this situation.

```
public class Seller_SensoryBehaviour extends TSAM_SensoryBehaviour{
  public Seller_SBehaviour(Agent a, long p) {// creates a new instance of Seller_SBehaviour
        super(a,p);
        myAgent=(SellAgent)a;
    }
  public void myaction() { //supplied by application programmers
   sense (); //senses the environment object of the seller agent

   // triggers a reaction to stop all its transactions if it will abort,
   If (abortName.equals(myAgent.getLocalName())&&abortType.equals("abort"))
   {
     // deals with the exception
              … …
   }
   //checks other environment parameters
   … …
    }
}
```

Figure 4-8 Code Fragment of a Sensory Behavior of a Seller Agent

TSAM simplifies the implementation of the sensory behavior of an e-market agent. A sensory behavior is a special behavior because of its periodicity of running. Without TSAM, the management of its execution must be considered explicitly. TSAM performs these by encapsulating them in the class TSAM_SensoryBehavior. Application programmers only need to code the functions of sense and reactions in the method of 'myaction()'.

4.4.2 Implementation of the ProactiveBehavior of E-Market Agents

In TSAM, two types of proactive behaviors are created for each e-market agent: (a) one main-role behavior, (b) a set of role-based behaviors. The role-based behaviors perform role-related tasks, while the main-role behavior is responsible for dispatching the role-based behaviors. We select a buyer agent in the e-market to show how the main-role behavior and the role-based behaviors are created.

Role-based Behaviors:

In the e-market, a buyer agent plays seven roles. The corresponding seven role-based behaviors (listed in Table 4-2) are created, each of which performs different role-related tasks coded in the method of 'myaction()' of each behavior class. Figure 4-9 describes a code fragment of a role-based behavior (Buyer_PBehaviour_DirectSearch). It is implemented by extending the abstract class of TSAM_RoleBehavior. TSAM differentiates role-based behaviors from other types of behaviors in the implementation and simplifies couplings among the role-based behaviors through internal shared objects.

```
public class Buyer_PBehaviour_DirectSearch extends TSAM_ProactiveRoleBehaviour{
  /creates a new instance of Buyer_PBehaviour_DirectSearch
  public Buyer_PBehaviour_DirectSearch(Agent a, int i, String taskname) {
    super(a);
    myAgent=(BuyerAgent)a;
    index=i;
    currentTaskName=taskname;
  }
  public void myaction() { //supplied by application programmers to perform the search task
        ... ...
  }
  ... ...
}
```

Figure 4-9 Code Fragment of a Role-based Behavior of a Buyer Agent

Main-Role Behavior:

Each agent in the e-market has one main-role behavior responsible for dispatching the role-based behaviors. Figure 4-10 describes the implementation of a main-role behavior of a buyer agent via TSAM.

```
public class Buyer_PBehaviour_MainRole extends TSAM_MainRoleBehaviour{
  // creates a new instance of Buyer_ProactiveBehaviour_MainRole
  public Buyer_PBehaviour_MainRole(Agent a, long p) {
    super(a,p);
    myAgent=(BuyerAgent)a;
  }
  public void myaction() {//dispatch algorithm supplied by application programmers
   BuyerTaskTable btTable=((BuyerAgent)myAgent).getBuyerTaskTable();
   if btTable is empty { //activates a role-based behavior for competing for a task
    Buyer_PBehaviour_CompeteForTask PB1=new Buyer_PBehaviour_CompeteForTask (myAgent);
    myAgent.addBehaviour(PB1);
   }
   for (int i=0;i<btTable.size();i++) { //dispatches the  role-based behaviors
     taskState=btTable.getTaskState(i);
     switch (taskState) {
      case 0: //starts to search for the price information
          Buyer_DirectSearch PB2=new Buyer_DirectSearch(myAgent,i,taskName);
          myAgent.addBehaviour(PB2);
          exit=1;
          break;
      case 1: //is searching
          exit=1;
          break;
      case 2: //has finished searching
          btTable.setTaskState(i,3); //begins to buy
          exit=1;
          break;
      case 3://starts to buy, activates role-based behaviors to perform buying task
          startBuyingBehaviors (i,taskName);
          exit=1;
          break;
      case 4://already starts to buy and may be receive part result
          exit=1;
          break;
      case 5: // ends one buying
          btTable.setTaskState(i,-1); //sets end state
          sendBackTaskResult(i);
          exit=1;
          break;
        ... ...
     } //switch
     if (exit==1) break;
   }//for
   ... ...
  }
}
```

Figure 4-10 Code Fragment of a Main-Role Behavior of a Buyer Agent

In the code fragment, the work of the main-role behavior is to dispatch all the role-based behaviors according to the current buying task and the internal state of the agent. It is

created through extending the TSAM_MainRoleBehavior. In the method of 'myaction()', a dispatch algorithm is implemented. First, a role-based behavior of CompeteForTask is activated to compete for a new buying task. Then, a role-based behavior of DirectSearch is triggered for searching price information. Afterwards, a proper role-based behavior (PublicSale, EnglishAuction, GroupAuction, Negotiation, or GroupBuying) is started based on a selected buying strategy. During the buying, the internal state of a buyer agent is changed. Specifically, when the internal state=1, the agent is in the searching state; when the internal state=2, the agent has finished searching. The internal state=3,4,5 means the agent is in the state of start of buying, buying and completion of buying respectively.

TSAM supports application programmers to implement the proactive behaviors more clearly and easily. On one hand, both the main-role behavior and role-based behaviors are roles related. They belong to the same proactive part of an agent. On the other hand, the main-role behavior is different from the role-based behaviors. It runs continuously whereas the role-based behaviors start and stop when the task is completed. TSAM separates them with distinct implementations, so that application programmers can concentrate their efforts on the implementation of the respective role-related tasks.

4.4.3 Implementation of the ReactiveBehavior of E-Market Agents

The reactive behaviors of an agent are specific behaviors that support the asynchronous notifications from other agents. TSAM provides an abstract class TSAM_ReactiveBehavior for creating reactive behaviors. In section 4.3, we have described the use of the reactive primitive to register a reactive behavior of an agent. In

Figure 4-11, we describe how to create a reactive behavior of a seller agent in a group auction.

A reactive behavior of a seller agent is obtained by extending the abstract class of TSAM_ReactiveBehavior. A seller agent first registers a reactive tuple of a bid request and its reactive behavior in a tuple space. Whenever a buyer agent writes reactive tuple in the tuple space, the reactive behavior of Seller_RBehaviour_GroupAuction is triggered. This leads to a decrease of the current bid price and possibly the remaining amount of the product, which are shared by the buyer agents in the group. The codes in the method of 'myaction()' have been presented and explained in Figure 4-4.

```
public class Seller_RBehaviour_GroupAuction extends TSAM_ReactiveBehaviour {
  //creates a new instance of Seller_RBehaviour_GroupAuction
  public Seller_RPBehaviour_GroupAuction(Agent a) {
    super(a);
    myAgent=(SellerAgent)a;
  }
  public void myaction() {//reactions supplied by application programmers
    if (reactiveTuple!=null) {
      // reads the content of the reactive tuple
            ... ...
      //changes the current bid price and the remaining quantity of the product in PrinceInfo
            ... ...
    } //if
  }
}
```

Figure 4-11 Code Fragment of a Reactive Behavior of a Seller Agent in Group Auction

4.5 Conclusion

In this chapter, we choose an e-market application as an example to illustrate the use of TSAM in the implementation of agent behaviors. In summary, TSAM facilitates the programmability of agents in the following aspects:

(1) TSAM provides a set of abstract classes of agent behaviors that simplifies the implementation of agent behaviors.

(2) Tuple space based agent coupling primitives support programmers to easily implement efficient agent interaction protocols. However, in order to use the TSAM model, the programmer has to construct the different fields of the tuples for each primitive.

Chapter 5 Performance Test

In Chapter 3 and Chapter 4, we presented how TSAM makes the design and implementation of multi-agent systems relatively easier. In this chapter, a set of simulation experiments of our prototype system will be introduced. In these experiments, we conduct performance tests based on two agent couplings media: tuple space and message passing. The objective of the comparison tests is to show how tuple space affects the application performance compared with massage passing. Section 5.1 describes the generation of test data in the market model. Section 5.2 introduces the test plan of how to select test data and application scenarios in the experiment. The test results and their analysis are presented in section 5.3. Lastly, the conclusions of the tests are summarized in section 5.4.

5.1 Data Generating

In the e-market model of our prototype system, three types of data are included to keep the system running in a simulation environment: (1) Consumers task data, (2) Supplier product data, and (3) E-market place running data. Consumers and Suppliers are both customers of the e-market to achieve their different objectives.

5.1.1 Consumer Task

A task of consumers is abstracted as a purchase request that includes the following information:

- Task name: indicates a task of buying products. It is exclusive in buyer's task list.
- Buying strategy: is used to calculate the price in price negotiation.
- Quantity: is the number of items to buy for each product.

- Permitted high price: is the upper limit price. Any price exceeding it can't be accepted to the consumer.
- Lower limit price: is the minimal permitted buying price that can be set to zero.
- Expiry date: is the deadline of the task. After this date, the task fails if it is not finished.
- Product name, product brand, product model, manufacturer, and product functions: are requirements for each buying task.

These data are generated automatically for the simulation test in the following ways: (1) the product name, product brand, product model, manufacturers and product functions are selected randomly based on suppliers' data; (2) the quantity of each buying task can be generated as an integer in the range (1 to x) where 'x' is the quantity of product in the suppliers' data; (3) the upper limit price of each task is generated randomly in the range of (LLP, LLP+δ) where LLP is the lower limit price of each item in the suppliers' data and δ is a designated overlap rate of buying and selling prices; (4) the expiry date is generated by considering the time period of simulation.

5.1.2 Supplier Product Data

The supplier product data are the basis for any transactions in the e-market system. Each seller agent sells product items that may belong to different suppliers. In the simulation test, we put all the suppliers' data into one data source that include all the products items sold in the e-market. The data of each seller agents can be generated based on the suppliers' data source where at least the following information is included:

- Item name: is the unique identity of the selling item.
- Product name, product brand, product model, manufacturer, and product functions: are product characteristics of this item.

- Strategy selection: in the application, we developed four strategies for selling each item: Negotiation, English Auction, Group Auction and Public Special Sale.
- Lower limit price (LLP): is the minimal permitted selling price, under which no buying request can be accepted.
- Available number: is the available quantity of this item.
- Expired date: is the end time of selling this item.
- Group-buying strategy: is a selection of strategies for determining a group-buying price.

When n product items are available for selling, the related data can be generated: (1) each item name is created as 'item-no', where the 'no' can be generated as an integer value from 1 to n; (2) the product brand, product model, manufacturer and functions are generated by numbering from 1 to 5; (3) the selling strategy is generated as integer value from 1 to 4 for the four types of transactions; (4) the lower limit price is generated in the range of the average price +/- variable range; (5) the group-buying strategy is randomly selected from three predefined functions to calculate group price. Here we use three calculation functions (two exponential functions and one linear function) with the quantity of buying as the variable.

5.1.3 E-Market Place Running Data

When the e-market system runs, the following data are needed to characterize the market:

- Number of physical hosts in the system.
- Number of distributed markets, each of which can be taken as a mall.
- Number of seller agents.
- Number of buyer agents and size of group purchase.

Based on the above data, the e-market running data is generated randomly. This generated data comprises two parts:

(1) Seller's Data

Based on the number of the seller agents, a seller's data is generated. Each seller agent randomly selects product items and the available quantities from the suppliers' data. The total quantity of the item to be sold by all the seller agents equals to the total amount of the item in the suppliers' data. The selling strategy can be generated based on the suppliers' data.

(2) Agents Running Data

Before the system runs, a set of running configuration data must be determined. These data are system wide: (i) random distribution of markets (malls) and agents across the physical nodes, and (ii) random logical distribution of sellers registered in different markets.

5.2 Test Plan

5.2.1 Application Performance

For e-market applications, in highly competitive business environments, application performance is critical to the business success. Poor performance can result in lost time, lost customers and lost market revenue. Application performance represents the performance of the entire market that includes high level of user satisfaction and maximizing efficiency and productivity of the whole market.

(a) The Quality of Consumers' Satisfaction

From the point of view of consumers, the response time is the main factor to evaluate the e-market. In the simulation test, we select two parameters to express the satisfaction

quality of consumers: (i) average waiting time for consumers to get the final purchases, and (ii) possible successful number of transactions within a specified period of time.

(b) The Quality of Suppliers' Satisfaction

From the point of view of suppliers, the whole market sale is what they care about. That is the actual revenue and successful number of transactions within a specified period of time.

5.2.2 Test Data

In the simulation, we consider centralized market place, distributed market places and group purchases to demonstrate how the tuple space and message passing affect the application performance as the number of buyer agent changes.

(1) Centralized Market Place

There is only one market place in the system. All the agents buy and sell their products through this market.

(2) Distributed Market Places

There are more than one market place in the system. All the agents register and trade on different market places. Concurrent transactions happen on different markets.

(3) Group Purchase

Group purchase involves collaborations among buyer agents before they interact as a group with seller agents. The objective of group purchases is to show the different effects of tuple space and message passing on agent collaborations.

In each of the above situations, three types of application performance (response time, number of transactions and market revenue within a specified period of time) are obtained as the number of buyer agents increases. In each case, we plot two curves corresponding to results of tuple space and message passing.

5.3 Test Results and Analysis

5.3.1 Centralized Market Place

(1) Test Data

Market size; 500 items

Number of physical nodes (machines): **8**

Number of seller agents: **10**

Number of buyer agents: **10, 30, 50, 70, 100**

(2) Test Results

(**ACL**: message passing based Agent Communication Language, **TS**: Tuple Space)

number of buyer agents	performance					
	customer				market	
	aver. waiting time(ms)		trans. numbers(within 150s)		market revenue(within 150s)	
	ACL	TS	ACL	TS	ACL	TS
10	368216	439798	97	86	831258	734780
30	196975	159084	241	271	1820741	2150704
50	121170	117051	422	441	3058159	3235605
70	118824	109006	469	519	2940827	3678215
100	103215	221668	552	134	3260555	771755

Table 5-1 Performance in a Centralized Market Place with Small Market Size

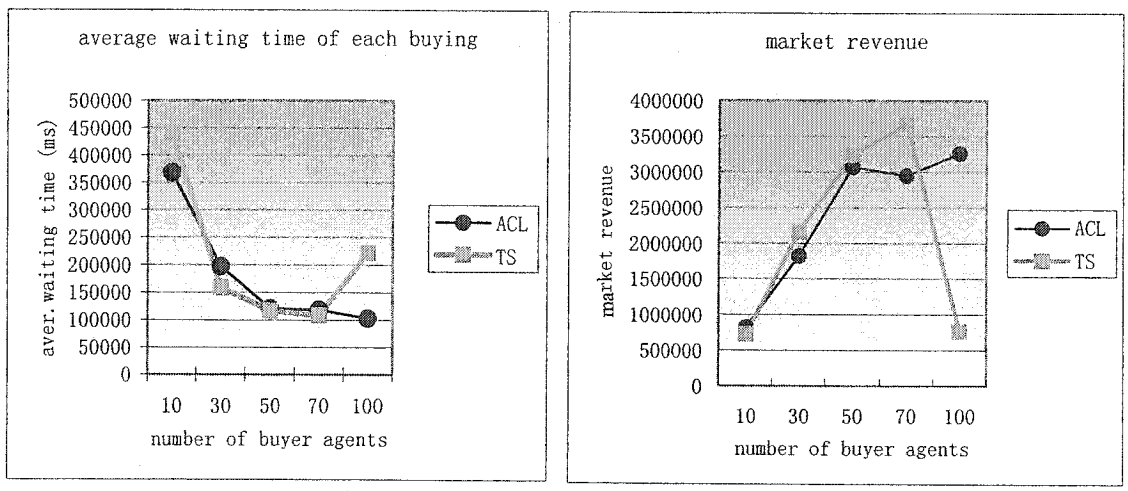


Figure 5-1 Curves of Performance in a Centralized Market Place with Small Market Size

(3) Analysis

In the case of message passing, all the agents in a container share one sending buffer, which needs to be synchronized when they send messages. When the number of buyer agents is small (smaller than 20), the couplings among agents tend to favor sequential process, in which message passing has more advantage than tuple space.

When using tuple space, all the agents in one container can access the tuple space concurrently. In addition, the bulk tuple match and reactive tuples improve the concurrent couplings among agents. When the number of agents lies in a proper range (20 to 70), the application performance of using tuple space is better (better around 13%).

However, tuple space needs to synchronize the concurrent accesses from agents. When the number of buyer agents is increased to the limit (70), the number of synchronized accesses of tuples also increases. In turn, this causes a longer waiting time. As a result, the performance with tuple spaces is not as good as the message passing.

The same test data are applied with the market size is increased to 3000 items. Table 5-2 and Figure 5-2 shows the test results.

number of buyer agents	performance					
	customer				market	
	aver. waiting time(ms)		trans. numbers(within 150s)		market revenue(within 150s)	
	ACL	TS	ACL	TS	ACL	TS
10	342944	363147	110	105	2010506	2002420
30	136877	146109	323	296	6013648	5493972
50	121515	122140	511	410	7767212	6474508
70	119370	192897	447	173	8469838	3334072
100	92007		579		8559111	

Table 5-2 Performance in a Centralized Market Place with Large Market Size

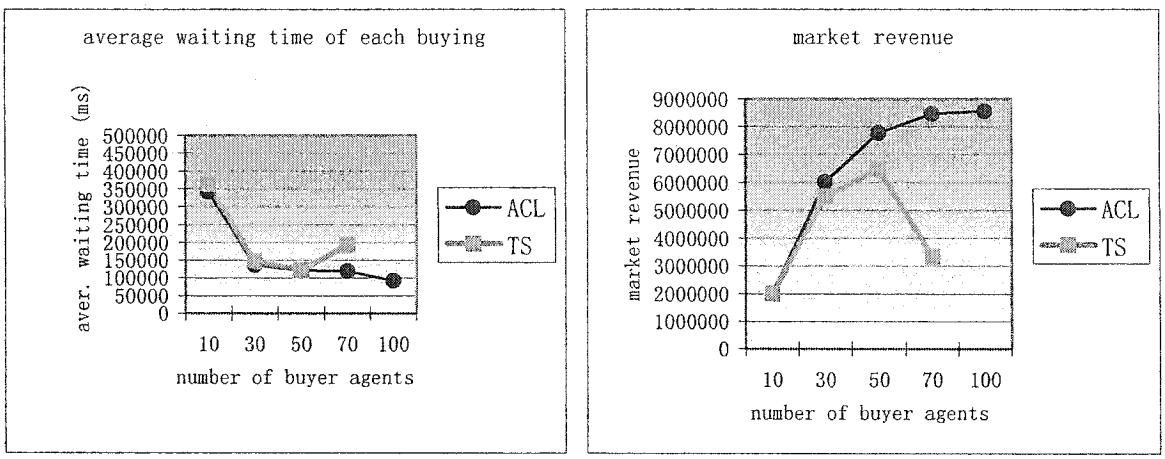


Figure 5-2 Curves of Performance in a Centralized Market Place with Large Market Size

When the market size is large, the ACL always performs better than the TS (better around 6%). The difference between the two performances is more pronounced as the number of buyer agents increases. The time spent on synchronized access to tuple space is the main factor that counteracts the tuple space positive effects. The market size also affects the breakpoint at which the tuple space performance would start to decrease. The breakpoint is at 70 (number of buyer agents) when the market size is 500, whereas the breakpoint is lowered to 50 when the market size is 3000. This result is attributed to the increases of templates in the tuple space when the market size increases. Distinct templates lead to distinct entities to be matched and this may increase the associative search time.

5.3.2 Distributed Market Places

(1) Test Data

Number of physical nodes (machines): **8**

Number of seller agents: **10**

Number of distributed market places: **4**

Number of buyer agents: **10, 20, 30, 50, 70**

(2) Test Results

number of buyer agents	performance					
	customer				market	
	aver. waiting time(ms)		trans. numbers(within 150s)		market revenue(within 150s)	
	ACL	TS	ACL	TS	ACL	TS
10	450111	413242	78	89	675435	762207
20	240706	225689	163	191	1279286	1522138
30	172531	160746	248	270	1944340	2143410
50	121803	154060	413	328	3208497	2514843
70	103485	298107	488	110	3649476	916084

Table 5-3 Performance in Distributed Market Places

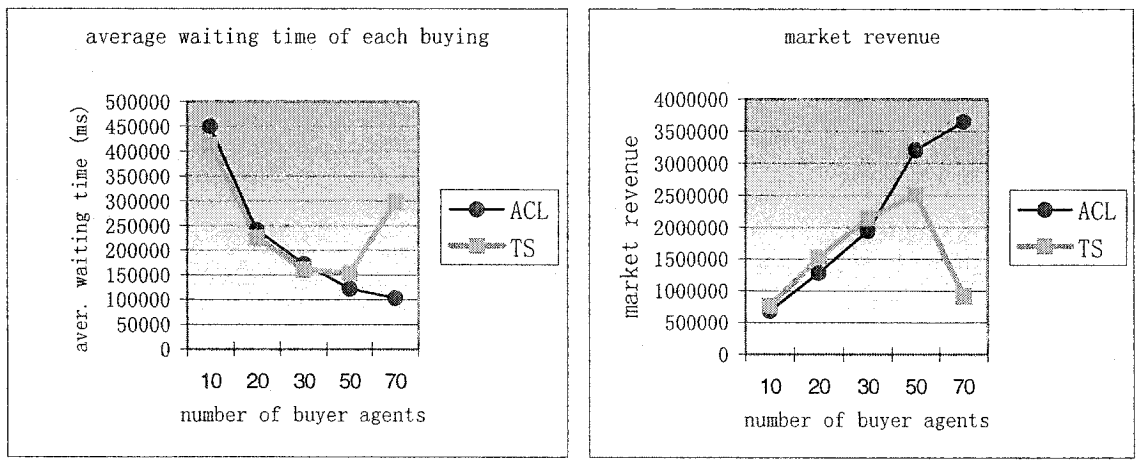


Figure 5-3 Curves of Performance in Distributed Market Place

(3) Analysis

When the number of buyer agents is not large (less than 50), the application performance of tuple space is not worse than that of ACL. As we already know that, in message passing, agents in one container send their messages sequentially. In the case of using distributed tuple spaces, the agents in a container can access the different tuple spaces concurrently. This provides real concurrent activities of agent couplings.

When the number of buyer agents exceeds 50, the performance of tuple space becomes worse because more synchronized access to the distributed tuple spaces block the accessing processes. Distributed market place seems to result in a lower breakpoint (from

70 to 50 compared with centralized market place). This may be attributed to the behavioral pattern of agents in the use of these markets and the actual concurrency that is achieved.

5.3.3 Group Purchase

(1) Test Data

Number of physical nodes (machines): **8**

Number of seller agents: **10**

Number of distributed market places: **4**

Total number of buyer agents: **30**

Group size (number of buyer agents in a buying group): **5, 10, 20, 30**

(2) Test Results

number of buyer agents in the group	performance					
	customer				market	
	aver. waiting time(ms)		trans. numbers(within 150s)		market revenue(within 150s)	
	ACL	TS	ACL	TS	ACL	TS
5	150885	135345	197	207	1625932	1844710
10	140766	117205	213	240	1914060	2041876
20	137192	113694	226	247	1888201	2261942
30	136178	104737	228	254	2010117	2326894

Table 5-4 Performance in a Group Purchase

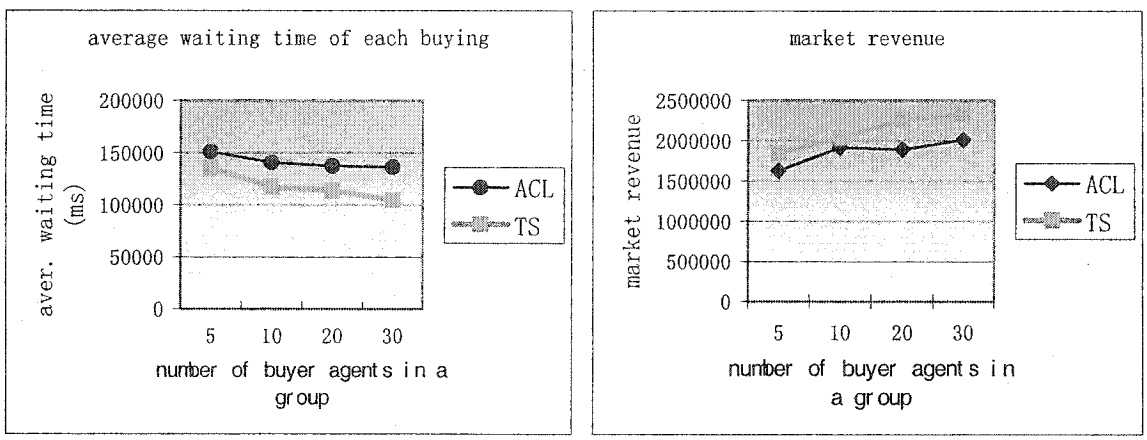


Figure 5-4 Curves of Performance in a Group Purchase

(3) Analysis

Tuple space supports the agent collaborations more efficiently than message passing in all cases of group buying. When the group size increases, more buyer agents join the group of buying. Once the group relationship is established, communication loads between agents decrease as much less number of buying transactions are involved. As a result, the performance gets better. Moreover, the characteristics of tuple space (global information sharing, bulk template match and reactive tuples) introduced in this thesis seem to have a strong positive impact on agent collaborations.

5.4 Conclusion

From the simulation test and the comparison of the test results by two (tuple space via message passing) agent coupling media, we can draw the conclusions that using tuple spaces as the agent coupling medium does not produce worse application performance. In case when distributed tuple spaces exist and when agents require tighter coordination as in group buying, tuple space has a performance advantage over message passing. However, when heavily synchronized communication is involved, there is a breakpoint, beyond which the load increases induce sharp decline in performance due to the associativity semantics a tuple space.

Chapter 6 Conclusion

6.1 Summary

Agent oriented software development is gaining importance in building complex distributed systems. In order to support simple design and implementation of agent based applications, this thesis provides a new agent based programming model — a tuple space based agent-programming model (TSAM), to facilitate the development of multi-agent systems in a systematic way.

TSAM supports the development of agent based systems in two levels: (a) in the abstract level, TSAM provides separate definitions of the three main agent behaviors namely sensory behaviors, reactive behaviors and proactive behaviors, and uses role model for analysis and design of agent proactive behaviors; (b) in the implementation level, TSAM uses tuple space as a complementary agent interaction medium, which is different from the traditional message-passing based agent coupling manners.

To validate the functions for which TSAM is particularly fitted in the development of agent based systems, the thesis has applied TSAM in developing an e-market application. Through the design and implementation of the e-market, TSAM is shown to be able to facilitate the development process, specifically support efficient agent couplings. Further, performance testing on the resulting system confirms our expectations that tuple space does not cost us more than message passing unless the system size has grown to become too large. In summary, TSAM is a practical solution for the development of complex multi-agent based applications. Specifically, we draw the following key conclusions:

(1) TSAM facilitates the design and implementation of agent based systems.

Unlike other agent model [1][8][10][[27][29][34], TSAM describes the agent architecture with three main agent behaviors and supports role based analysis in the design of proactive role-based behaviors. These simplify the design process and help designers to concentrate on the design of flexible collaboration relationships among agents. Moreover, using tuple space as the agent interaction medium simplifies the design and implementation of dynamic agent couplings. In the implementation stage, TSAM has facilitated the creation of agent behaviors, and the rich tuple space primitives (e.g. bulk and logical template, asynchronous access) simplify the coding efforts needed.

(2) Tuple space has compatible performance as message passing based agent interaction at reasonable system sizes.

The characteristics of tuple space services, namely dynamic information sharing, asynchronous access, bulk and logical template match and reactive tuples, efficiently support the concurrent couplings among agents, which make the application performance not worse than that of message passing. Distributed tuple spaces in the form of multiple market places improve performance and in some cases may even lead to better results than message passing based counterpart. Moreover, in the case of tighter coordination among agents, tuple space has an obvious performance advantage over message passing. However, each tuple space manager has to synchronize the access of tuples in the tuple space, which will affect the performance when the number of access requests increases. This can instruct the designer to select appropriate schemes according to the real application situations.

6.2 Future Work

This thesis has proposed a practical solution for developing agent-based systems. Our approach is intended to complement current practices of agent oriented engineering by

providing a general agent programming model, within which the development process can be simplified and the complex collaboration among agents can be more effectively managed. The following are some future research works.

(1) Combination of the two agent coupling mechanisms for a better performance.

The openness of ubiquitous computing presents a set of challenges for agents to engage in complex collaborations. Although we have described some of advantages of tuple space in supporting agent interactions, costs on tuple matches and heavy synchronized access are possible problems. In peer-to-peer interaction, message passing seems to be more direct and effective. Hence, a method of combining the two coupling mechanisms under different interaction protocols will get better performance.

(2) Development of agent building tool and agent running environment for TSAM that is a more general environment than JADE.

TSAM is built on top of JADE because JADE is open source written in Java that can be modified easily. However, the agent behavior classes in JADE do not offer programmers much in terms of reusable code. A next phase is to build a complete TSAM on top of the Java platform directly so that other aspects of distributed systems can be more directly manipulated and supported.

Bibliography

[1] M. S. Atkin, G. W. King and D. L. Westbrook, "Hierarchical Agent Control: A Framework For Defining Agent Behavior", *AGENTS'01*, Montreal, Quebec, Canada, May 28-June 1, 2001.

[2] F. M. T. Brazier, B. Dunin-Keplicz, and N. Jennings, and J. Treur, "DESIRE: Modelling Multi-Agent Systems in A Compositional Formal Framework", *Int. J., of Cooperative Information Systems*, vol. 6, pp. 67-94, 1997.

[3] M. Becht, T. Gurzki, J. Klarmann, and M. Muscholl, "ROPE: Role Oriented Programming Environment for Multiagent Systems", *in Proc. 4th IFCIS Conf. on Cooperative Information Systems (CoopIS'99)*, Edinburgh, Scotland, September 1999.

[4] G. Cabri, "Role-based infrastructures for agents", *8th IEEE Workshop on Future Trends Distributed Computing System*, Bologna, Italy, Oct.31-Nov. 02, 2001.

[5] G. Cabri, L. Leonardi, and F. Zambonelli, "MARS: A programmable coordination architecture for mobile agents", *Internet Computing*, vol. 4, no.4, pp. 26–35, 2000.

[6] G. Cabri, L. Leonardi, and F. Zamboneli, "The Impact of the Coordination Model in the Design of Mobile Agent Application", *in Proc. 22th Annual Int. Computer Software and Applications Conference (COMPSAC 98)*, Vienna (A), August 1998, IEEE CS Press.

[7] N. Carriero, D. Gelernter, "Linda in Context", *Communication of the ACM*, vol. 32, no.4, pp. 444-458, April 1989.

[8] S. Donikian, "HPTS: A Behaviour Modelling Language for Autonomous Agents", *AGENTS'01*, Montreal, Quebec, Canada, May 28-June 1, 2001.

[9] R. Depke, R. Heckel, J. M. Kuster, "Improving the Agent-Oriented Modeling Process by Roles", *AGENTS'01*, Montreal, Quebec, Canada, May 28-June 1, 2001.

[10] M. Fisher and C. Ghidini, "The abc of rational agent modelling", *AAMAS'02*, Bologna, Italy, July 2002.

[11] R. A. Flores-Mendez, "Towards the Standardization of Multi Agent Systems Architectures: An Overview", *ACM Crossroads - Special Issue on Intelligence Agents*, vol. 5, no. 4, ACM Press, Summer, 1999.

[12] J. M. Fernandes, O. Belo, "Modeling Multi-Agent Systems Activities through Colored Petri Nets: An Industrial Production System Case Study", *in Proc. 16th IASTED Int. Conf. on Applied Informatics*, February23-25, 1998, Anaheim, CA, pp. 17-20, 1998.

[13] D.Gelernter, "Generative communication in Linda", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, pp: 80-112, 1985.

[14] G. Gottlob, M. Schrefl, B. Rock, "Extending Object-oriented Systems with Roles", *ACM Trans on Info. Sys.*, Vol.14, No.3, July, 1996, 268-296.

[15] Z. Guessoum and J. P. Briot, "From Active Objects to Autonomous Agents", *IEEE Concurrency*, vol.7-1, pp. 68-76, 1999.

[16] N.T. Giang, D.T. Tung, "Agent Platform Evaluation and Comparison", *Institute of Informatics of Slovax Academy of Science*, June 2002.

[17] R.Gray, "Agent Tcl: A flexible and secure mobile-agent system," *4th Annual Tcl/Tk Workshop (TCL'96)*, Monterey (CA), July 1996.

[18] N. R. Jennings, P. Faratin, T, J. Norman, P. O'Brien, B. Odgers and J. L. Alty, "Implementing a Business Process Management System using ADEPT: A Real-World Case Study", *Intl. Journal of Applied AI*, vol 14, no.5, pp. 421-465, 2000.

[19] N. R. Jennings, "On agent-based software engineering", Artificial Intelligence, vol. 117, pp. 277–296, 2000.

[20] N. R. Jennings and M. Wooldridge, "Agent-Oriented Software Engineering", Handbook of Agent Technology, AAAI/MIT Press.

[21] E. A. Kendall, "Role model designs and implementations with aspect-oriented programming", *in Proc. ACM Conf. Object-Oriented Systems, Languages, and Applications*, Denver, Colorado, United States, 1999, pp. 353-369.

[22] E. A. Kendall, "Role models – patterns of agent system analysis and design", *BT Technol. J.*, vol. 17, no.4, October 1999.

[23] E.A.Kendall, P.V.Murali Krishna, C.V.Pathak, and C.B.Suresh, "An Application Framework for Intelligent and Mobile Agent Systems", *ACM Computing Surveys (CSUR)*, vol. 32, March 2000.

[24] J. Lind, "Issues in Agent-Oriented Software Engineering", *in Proc. 1th International Workshop on Agent-Oriented Software Engineering (AOSE 2000)*, 2000.

[25] Y. Labrou and T.Finin, "KQML as an agent communication language", in Bradshaw.J., Software Agents, The MIT Press, 1997.

[26] P. Maes, R. H. Guttman, A. G. Moukas, "Agents That Buy and Sell", *Communications of the ACM*, 1999, pp 81-91.

[27] H.A. Mallot, "Behavior-Oriented Approaches to Cognition: Theoretical Perspectives", *Theory in Biosciences*, vol.116, pp. 196-220, 1997.

[28] P. A. Mitkas, D. Kehagias, A. L. Symeonidis, and I. N. Athanasiadis, "A Framework for Constructing Multi-Agent Applications and Training Intelligent Agents", *in Proc. 4th Int.Workshop on Agent-oriented Software Engineering (AOSE-2003)*, 2003.

[29] J. P. Muller and M. Pischel "Modelling reactive behaviour in vertically layered agent architectures," *Intelligent Agent: Theories, Architectures, and Language (LNAY vol. 890)*, pp. 261-276. Springer-Verlag: Berlin, Germany, 1995.

[30] T. Murata, "Petri Nets: Properties, Analysis and Applications", *in Proc. of the IEEE*, vol.77, no.4, pp.541-580, April 1989.

[31] H. S. Nwana, D. T. Ndumu & L. C. Lee, "ZEUS: An Advanced Tool-Kit for Building Distributed Multi-Agent Systems", *in Proc. 3th Int. Conf. on Autonomous Agents (Agents'99)*, 1999.

[32] C. Petrie, "Agent-Based Software Engineering", *Agent-Oriented Software Engineering, Lecture Notes in AI*, Springer-Verlag, 2001, pp. 58-76, Stanford Networking Research Center, Stanford, CA 94305-2232.

[33] S. Poslad, P. Buckle, and R. Hadingham, "The FIPA-OS agent platform: Open Source for Open Standards", *Nortel Networks*. Manchester, UK. April 2000.

[34] A. S. Rao and M. P. Georgeff, "Modelling rational agents within a BDI architecture", *in Proc. Int. Conf. on Principles of Knowledge Representation and Reasoning*, San Mateo, Kaufmann, 1991, pp. 473-485.

[35] A. S. Rao and M. P. Georgeff, "BDI Agent: From Theory to Practice", *ICMAS-95*, San Francisco, USA, June, 1995.

[36] M. J. Raphael and S. A. Deloach, "A Knowledge Base for Knowledge-Based Multiagent System Construction", *National Aerospace and Electronics Conference (NAECON)*, Dayton, OH, October 10-12, 2000.

[37] D. Riehle and T. Gross, "Role Model Based Framework Design and Integration", *in Proc. 1998 Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, ACM Press, 1998, pp. 117-133.

[38] A. Rowstron and S. Wray, "Run-Time System for WCL", *Internet Programming Languages*, Springer-Verlag, Lecture Notes in Computer Science 1968, 1999, pp.78-96.

[39] Y. Shoham, "Agent-oriented programming", *Artificial Intelligence*, vol. 60, no.1, pp. 51-92, March 1993.

[40] Y. Shoham, "Rational Programming", http://cs.stanford.edu/»shoham, June 5, 2003.

[41] G. Wagner "A Logical and Operational Model of Scalable Knowledge-and Perception-Based Agent", *in Proc. 7th European Workshop on Modeling Autonomous Agents in a Multi-Agent World: Agents Breaking Away*, Einhoven, The Netherlands, 1996, pp. 26-41.

[42] M. F. Wood, S. A. Deloach, "An Overview of the Multiagent Systems Engineering Methodology", *in Proc. 1th Int. Workshop on Agent-Oriented Software Engineering (AOSE 2000)*, 2000.

[43] M.Wooldridge and N.R. Jennings. "Intelligent agents: Theory and practice," *The Knowledge engineering Review*, vol.10, no. 2, pp.115-152, 1995.

[44] M. Wooldridge, "The Gaia Methodology for Agent-Oriented Analysis and Design", *Autonomous Agents and Multi-Agent Systems*, vol. 3, pp. 285-312, 2000.

[45] H. Xu, "Multi-Agent System and Agent Based Software Engineering", *Advanced Software Engineering*, December 11, 2003.

[46] F. Zambonelli, N. R. Jennings, M. Wooldridge, "Organisational Abstractions for the Analysis and Design of Multi-Agent Systems", *1th Int. Workshop on Agent-Oriented Software Engineering (AOSE 2000)*, 2000.

[47] Agent Tool: http://www.cis.ksu.edu/~sdeloach/ai/download-agentool.htm

[48] Aglet community http://aglets.sourceforge.net/

[49] Ajanta: http://www.cs.umn.edu/Ajanta/

[50] Agent Builder: http://www.agentbuilder.com/

[51] Agent Building Shell: http://www.eil.utoronto.ca/aac/abs/index.html

[52] Agent Technology Research: http://www.emorphia.com/research/overview.htm

[53] Autonomous Systems of Trade Agents in E-Commerce (ASTA): http://www.cwi.nl/projects/ASTA/

[54] e-AEC Research at Stanford: http://e-aec.stanford.edu/index.htm

[55] Colored Petri Nets: http://www.daimi.au.dk/CPNets/

[56] Grasshopper – the agent platform: http://www.grasshopper.de/

[57] FIPA-OS: http://fipa-os.sourceforge.net/

[58] JavaSpace: http://www.cdegroot.com/cgi-bin/jini/JavaSpace

[59] JADE: http://jade.cselt.it

[60] KQML: http://www.cs.umbc.edu/kqml

[61] MASIF—the Object Management Group's Mobile Agent System Interoperability Facility: http://www.omg.org/

[62] Publicly Available Implementations of FIPA Specifications: http://www.fipa.org/

[64] Yu Li, Master of Computer Science Thesis in Progress, Dept. of Computer Science, Concordia University.