Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master Thesis

Abstracting agent cooperation protocols in agent oriented programming by collective operations

エージェント指向プログラミングにおける集団操作を利用した
エージェント協調の抽象化

Nguyen Tuan Duc
グェン　トアン　ドゥク

Supervisor: Professor Ikuo Takeuchi

January 2009

Abstract

Multiagent system has been recognized as one of the next generation software architecture since it can be used for modeling a wide-range of complex problems as well as utilizing the massive power of distributed parallel computer systems. It has these abilities because of its two main characteristics, namely autonomy and cooperation. Each agent in a multiagent system acts as an autonomous entity which is able to reasoning about and adapt to any changes in the computing environment. Furthermore, by exchanging useful knowledge about the current situation, agents are able to cooperate with each other to achieve the overall goals. Many programming languages and frameworks have been proposed for programming autonomous intelligent software agents. Unfortunately, implementing cooperation protocols between agents is still a challenging task to create intelligent multiagent systems because there is a lack of models, languages and frameworks for description and execution of agents' cooperative actions. Therefore, in this thesis, we concentrate on the cooperation model of agents in multiagent systems by proposing methods for simplifying the description of cooperation protocols as well as designing language and building runtime system to support cooperative actions of agents.

This thesis proposes a new approach for abstraction and realization of agent cooperation protocols, that is using collective operation, a concept in parallel distributed programming, as cooperation primitive. To show that cooperation can be smoothly integrated into agent oriented programming, we designed and implemented a new agent oriented programming language called Yaccai which supports the description and execution of these cooperation primitives while maintaining the autonomous computational model of agents. Collective operations are implemented as language constructs of Yaccai. Many complex cooperation protocols can be straightforwardly mapped to collective operation constructs in Yaccai.

To prove the powerfulness of our model, we implemented the FIPA agent interaction protocols using message passing APIs (send, receive and collective operations) supported in our language. We show that using collective operations, many protocols in FIPA interaction protocol set are not only simple for coding but also intuitive to understand.

To evaluate the performance of cooperation protocols implemented using collective operations, we carried out many experiments on the Vacuum Cleaner simulation problem with different computational resources and problem settings. Our experiments show that collective operations allow agents to effectively cooperate to solve sophisticated problems of Distributed Artificial Intelligence.

概要

マルチエージェントシステムは次世代のソフトウェアアーキテクチャのひとつであり、さまざまな種類の複雑な問題のモデリング手法として知られている。マルチエージェントシステムが強い問題解決能力を持つのは、自律性と協調性という 2 つの特徴を持っているからである。個々のエージェントは自律的に計算環境の変化に適応し、達成したいゴールに向けて行動している。更に、マルチエージェントシステムの中の複数エージェントが通信、同期をして、知識を交換し、協調しながら問題を解決していく。すなわち、マルチエージェントシステムは並列分散計算機環境の巨大な計算能力を出せるソフトウェアモデルである。既存研究では、自律エージェントの記述手法として、エージェント指向プログラミング (AOP) をはじめ、多くのフレームワークが提案されている。しかし、マルチエージェントでの協調記述を支援する言語やフレームワークがまだ十分でないため、マルチエージェント開発はまだ困難な作業である。そこで、本研究は、エージェント協調記述を簡単にする手法とマルチエージェント開発のためのプログラミング言語と実行環境に注目する。

本論文はマルチエージェント協調プロトコルを簡単な集団操作を用いて構成することによりエージェントの協調を抽象化し、協調プロトコルを見通しよく、かつ検証しやすくすることを提案する。すなわち、協調のプリミティブとして、並列分散プログラミングでよく知られている集団操作を利用する。また、協調と自律というエージェントの 2 つの排他的な性質を 1 つのモデルに柔軟に統合することが可能であることを明らかにするために、Yaccai という新しいエージェント指向プログラミング言語を設計し、それを効率よく実装するための新しいエージェントの実行と通信モデルを提案する。集団操作は設計した言語のプリミティブであり、複雑な協調プロトコルをこれらのプリミティブを利用するだけで記述できる。更に言語の実行時システムを利用して分散マルチエージェントシステム全体を容易に構成、実行できる。

また、本研究が提案した集団操作の集合の記述力や表現力を評価するために、いくつかの複雑なエージェント協調プロトコルを記述した。その中で最も重要なプロトコル集合として、FIPA の全部で 11 個のエージェントインタラクションプロトコルを我々の集団操作で簡潔に記述できることを確認した。

更に、集団操作で記述した協調プロトコルの協調性能を調べるために、Vacuum Cleaner 問題をソフトウェアマルチエージェントシステムを用いて複数の計算機環境と問題設定でシミュレーションした。その実験結果から提案した集団操作によって効率よくエージェントの分散協調問題をモデリングできることが明らかになった。すなわち、本システムが分散人工知能の諸問題を解決するためのプラットフォームとして有効なことを明らかにした。

Acknowledgements

I am extremely grateful to my supervisor, Professor Ikuo Takeuchi, for all the guidance and support he has given me. Without his insightful comments on the overall direction and stimulating suggestions about the evaluation method, this research would not have been possible. His state-of-the-art Japanese and English have helped me a lot not only in writing papers, presenting slide shows but also in everyday conversations. I am enjoying his infinite source of humor that he delivers in every of his talks, comments and even in his emails. This has made the research life more interesting and allowed me to relax even in great pressure of completing a long thesis.

I am grateful to the thesis committee members, especially Professor Masami Hagiya and Professor Hiroshi Esaki, for their precious comments and suggestions to make my thesis better.

I would like to thank Dr. Koichi Sasada for his valuable comments on my research, and in particular, on my presentations and thesis. I really appreciated his kindness as he provides us with many interesting books about system programming, and of course, about Ruby.

I would also like to thank Dr. Yoshinori Tanabe and Dr. Takefumi Miyoshi for their suggestions on this work and particularly for their guidance on presentation and thesis writing. The figure about Agent oriented programming model in this thesis is a piece of art that they created for me.

I would like to express my gratitude to all other members in my laboratory for their encouragement and support for my research as well as daily life. Their guidance about Akihabara and delicious restaurants around the Daibiru is wonderful and amazing.

Finally, I wish to thank everyone else for their support and encouragement that help me to complete this thesis.

Contents

Chapter 1 Introduction 1
- 1.1 Background and Motivation 1
- 1.2 Contribution of this Work 2
- 1.3 Organization of this Thesis 3
- 1.4 Font face convention in this Thesis 3

Chapter 2 Multiagent systems and agent cooperation protocols 4
- 2.1 Multiagent systems 4
- 2.2 Agent oriented programming 5
- 2.3 Agent cooperation protocols 5

Chapter 3 Related work 7
- 3.1 Agent oriented programming languages 7
- 3.2 Multiagent cooperation models 8
- 3.3 Agent-oriented methodologies for grid applications 9

Chapter 4 Abstraction of agent cooperation protocols 10
- 4.1 Building blocks for cooperation 10
- 4.2 Set of essential collective operations 12
- 4.3 Building cooperation protocols from collective operations 15
- 4.4 Example implementation of some cooperation protocols 17

Chapter 5 Communication and execution model 22
- 5.1 Agent execution and communication model 22
- 5.2 The Yaccai programming language 25

Chapter 6 Evaluation 29
- 6.1 Expressiveness of collective operations as agent cooperation primitives 29
- 6.2 Performance of cooperation protocols implemented with collective operations 32

Chapter 7 Conclusion and future work 39
- 7.1 Conclusion 39
- 7.2 Future work 39

Publications 41

References 42

Appendix A Implementation of some FIPA protocols using collective operations 45
- A.1 Implementation of the FIPA Recruiting Interaction Protocol 45
- A.2 Implementation of other FIPA Interaction Protocols 45

Appendix B Grammar of the Yaccai programming language 47

List of Figures

2.1 A software multiagent system 4
2.2 The agent oriented programming model 5

4.1 Essential collective operations 13
4.2 Implementation of an auction protocol by collective operations 15
4.3 Cooperative problem solving using collective operations. 17
4.4 FIPA Contract Net Interaction Protocol (CNET) 18
4.5 Implementation of FIPA CNET protocol using collective operations . . . 19
4.6 FIPA Recruiting Interaction Protocol 20
4.7 Implementation of FIPA Recruiting protocol 20

5.1 Communication model for a multiagent system 23
5.2 Agent execution model (execution of agent's reasoning layer) 24
5.3 Excerpt of Yaccai grammar 26
5.4 A simple multiagent system definition 28

6.1 Implementation of Contract Net Interaction Protocol (CNET) in Yaccai 30
6.2 Implementation of Contract Net Interaction Protocol (CNET) in JADE 31
6.3 Homogeneous Vacuum Cleaner problem 32
6.4 Average score of 5 times of simulation on `map_dense` 34
6.5 Average score of 5 times of simulation on `map_sparse` 34
6.6 Average score of 5 times of simulation on `map_sparse` with "explore" command. 35
6.7 Heterogeneous Vacuum Cleaner problem 36
6.8 Average number of move commands of heterogeneous agent teams 37
6.9 Communication cost that cooperation takes (average number of messages an agent sent for cooperation) 38

A.1 Implementation of the FIPA Recruiting Interaction Protocol 46

List of Tables

4.1 Collective operations 12

5.1 Differences between the proposed model and MPI 23
5.2 Belief-base operations 27

Chapter 1

Introduction

1.1 Background and Motivation

The popularization of parallel distributed computing infrastructures such as cluster and grid has lead to many software applications which contain many components that communicate with each other in sophisticated cooperation protocols with complex communication patterns. Furthermore, the distributed nature of these infrastructures requires the operating programs be able to cope with unpredictable changes in the environment and adapt to the situation in an appropriate manner. Multiagent system (MAS), the software architecture that consists of many autonomous intelligent agents which interact by sending/receiving messages of several types, is a suitable paradigm for modeling these kinds of software applications. Multiagent system can be used for modeling problems which are difficult or impossible for a single monolithic program to solve. The main characteristics that give multiagent systems this powerful problem solving ability are autonomy and cooperation. Each agent in a multiagent system is an autonomous intelligent entity that can react to events and adapt to the environment. On the other hand, cooperation is a very important process in multiagent systems because it associates agents in a collaborative team to achieve the main objective of the system: reaching the overall goals. Agents cooperate with other agents by sharing knowledge and exchanging useful information to solve the problem. Therefore, description of cooperation protocols is a crucial requirement for programmers to realize intelligent software multiagent systems.

The autonomous behavior of agent can be rigorously modeled using agent oriented programming (AOP) [1], a natural successor of object oriented programming (OOP). In AOP, an agent is modeled as an autonomous entity which listens to events from the environment and from other agents and determines actions to do by itself. After having the action in mind, the agent will perform the action to change the environment. This cycle of sense-reasoning-act helps the agent to reach its goals and the cycle is called "reasoning cycle". Agent oriented programming allows one to precisely describe agents' knowledge about the world and automatically generate reasoning cycle.

On the other hand, cooperation has become an important topic in multiagent system research. There are many studies on cooperation model for agent such as the joint-intentions model [2], teamwork [3] or distributed cooperation model ECM [4]. However, there is a lack of effort for integrating cooperation into agent oriented programming. This causes difficulties for programmers to create multiagent systems using agent oriented programming languages and frameworks because it is tedious to describe agent's reasoning cycle separately from its interactions with other agents.

A general problem of existing cooperation models for autonomous agent is that they do not pay attention to the distributed nature of multiagent system. Some cooperation models such as joint-intentions model [2] or teamwork model [3] assume there are some

shared memory locations that agents can use to share their mental state and goals, but this can not be simply assumed for distributed environments. Other models like the Group Situation based Cooperation model [5] assume that there are some global coordinators which can be a central to control all other agents. This makes the model error-prone and creates bottleneck in a large-scale environment.

Distributed cooperation models such as ECM [4] do concentrate on the distributed requirement of multiagent systems but they lack support for the autonomous computational model of agents. They often model the interaction of processes in multiagent systems but can not capture the autonomous actions of agents. Some models even defer the description of agent's reasoning cycle to the users.

The cooperation process of agents can be viewed as the process of synchronization and passing messages between agents in order to coordinate and sharing knowledge about the outside environment. Therefore, it is important to investigate the application of message passing models into the description of cooperation process in multiagent systems. Yet there is little attention on method for constructing cooperation, coordination protocols from simple communication primitives (such as sending, receiving, broadcasting messages among the agents).

These reasons have motivated me to study methods for realizing complex cooperation protocols of agents from simple message passing primitives and integrating these methods into agent oriented programming.

1.2 Contribution of this Work

The main contributions of this work are as follows:

- **It proposes a new approach for abstraction and realization of sophisticated agents' cooperation protocols by using simple cooperation primitives called collective operations.**
  Collective operations are operations that are simultaneously executed by many agents, such as broadcasting messages among an agent group or finding some global properties of the entire multiagent system. We show that using only these primitives, many sophisticated cooperation protocols can be implemented. Using collective operations as cooperation primitives we were able to simplify the description of several complex cooperation protocols in the FIPA's agent interaction protocol set. Many protocols can be straightforwardly mapped to collective operations in an intuitive manner. More importantly, collective operations raise the abstraction level of agent cooperation protocols and make the description of these protocols well-structured. Therefore it is easier to debug and verify the behavior of the code that implements the protocols.
- **It proposes a new execution and communication model for agents in a multiagent system to smoothly integrate cooperation into agent oriented programming.**
  The integration of cooperation into agent oriented programming is not simple because in AOP, agent needs to be an autonomous, independent entity while cooperation involves sharing of knowledge and collaborating with each other. We have designed and implemented a new agent oriented programming language named Yaccai which supports the description of collective operations as cooperation primitives. Collective operations are implemented as language constructs of Yaccai. The underlying communication model of the Yaccai runtime system provides full support for message passing APIs while maintaining the autonomous computational model of agents.

- **It investigates the powerfulness of collective operations as cooperation primitives and the ability to apply these operations in the description of real-world multiagent systems.**
  We show that, global knowledge, a crucial element in realization of intelligent agents, can be easily derived using collective operations. We have implemented the Vacuum Cleaner simulation problem using collective operations in Yaccai and carried out many experiments with different computational resources and problem settings. The results confirmed that collective operations help the agents effectively cooperate with each other to reach the overall goals.

Our final target is making the description of multiagent systems easier, more intuitive and well-structured. We hope that the contributions of this work will create a paradigm shift from agent oriented programming (AOP) to multiagent oriented programming (MAOP).

1.3 Organization of this Thesis

The rest of this thesis is organized as follows. Chapter 2 provides some fundamental concepts about multiagent systems, agent oriented programming and agent cooperation protocols. We review related previous work in agent oriented programming, multiagent cooperation and agent oriented approach for grid applications in Chapter 3. We present our first proposal about method for modeling agent cooperation protocols using collective operations in Chapter 4. We also demonstrate the expressiveness of our model using many examples right after the proposal in the same chapter. Then, in Chapter 5, we explain our second proposal about communication and execution model for agents to support the syntax and semantics of collective operations as shown in Chapter 4. Chapter 6 evaluates our proposals from many aspects. Finally, in Chapter 7, we give the conclusion and discuss about future research direction.

1.4 Font face convention in this Thesis

In this thesis, collective operation names are written in typewriter type font. For example, `Reduce`, `Bcast` or `Gather` are collective operation names.

Chapter 2

Multiagent systems and agent cooperation protocols

In this chapter, we provide fundamental concepts of multiagent system, agent oriented programming and agents' cooperation. These concepts are essential for understanding of this thesis.

2.1 Multiagent systems

As introduced in Chapter 1, a *multiagent system* (MAS) is a system that consists of many autonomous intelligent *agents*. Each agent can be any entity that has the ability of reasoning about current situation of the environment and taking appropriate actions. Examples of agents contain human beings (a kind of perfect autonomous intelligent agent), robots and intelligent software programs.

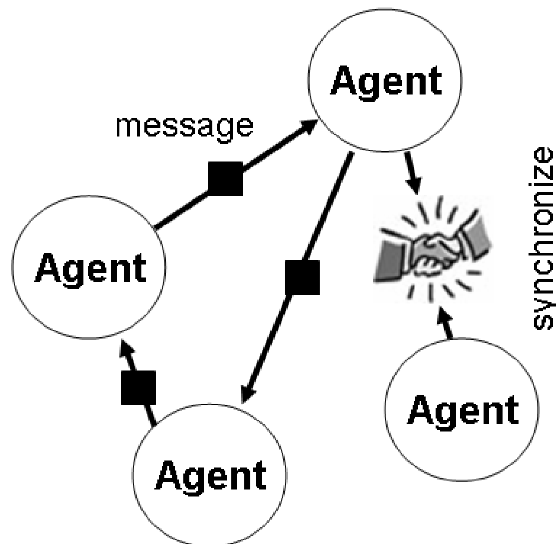


Fig. 2.1. A software multiagent system

This research focuses on software agent, that is, software program which has intelligent reasoning mechanism and can function in a complex computational environment such as computer cluster or grid. Multiple programs (processes of execution, agents) that interact by communicating or synchronizing with each other to solve some common problems form a software multiagent system. The term "multiagent system" mentioned in this thesis has

this meaning except when explicitly specified. A multiagent system that contains four agents is depicted in Figure 2.1.

2.2 Agent oriented programming

Yoav Shoham has proposed a new programming paradigm called agent oriented programming (AOP) [1] for description of autonomous agents. In this programming paradigm, each agent is considered as an object (or actor in Actor model [6]) whose state contains components such as beliefs (belief about the world, the environment, about itself and about other agents), desires (goals), capabilities and intentions. An agent rationally maintains the consistency between its beliefs by itself and therefore it is said to be "autonomous". As illustrated in Figure 2.2, an agent has knowledge about the world (the environment) as its *belief* and the goal to perform as its *desire*. To perform a goal, an agent needs some plans (capabilities) as the recipe for achieving the goal and committed plans are called *intentions*. Belief, desire and intention (BDI) form the state of the agent (which is called *mental state*) and the cycle of sense-reasoning-act (as shown in Figure 2.2) is the "reasoning cycle" of the agent.

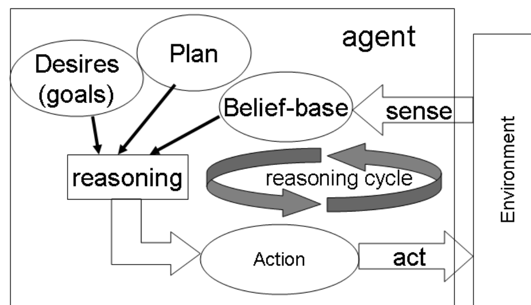


Fig. 2.2. The agent oriented programming model

An agent oriented programming language allows one to precisely model agent's mental state using its language constructs and the language runtime system automatically generates reasoning cycle.

2.3 Agent cooperation protocols

Cooperation is known as the process of synchronization and sharing knowledge to achieve a goal or to reach an agreement between agents in a multiagent system (see Figure 2.1). Cooperation is a crucial process in multiagent system because without cooperation the system is simply a set of separated agents and has no ability of collaborating to reach the goal. The protocols that describe cooperation process between agents are called cooperation protocols. There are many standard agent cooperation protocols (such as the FIPA Contract Net Protocol), but many non-standard or customized protocols also exist. Therefore, it is required that a multiagent platform must provide the ability of realizing an arbitrary cooperation protocols.

There are two important aspects of a cooperation process, namely, data movement pattern and computation pattern. Data movement refers to the communication process in the cooperation protocol while computation pattern refers to the actual computation functions that are executed to process the communicated data. This research mainly

deals with the problem of abstracting communication patterns in multiagent systems. In addition, it also provides mechanism for describing some complex computation patterns (e.g., reduction).

Chapter 3

Related work

3.1 Agent oriented programming languages

Many agent oriented programming languages have been proposed, such as AgentSpeak(L) [7], Golog/ConGolog [8], 3APL [9, 10, 11] or BOID [12]... These languages focus on formal model of agent, they provide formal semantics and support for model checking, proof of correctness of the system. Most of these languages base on BDI logic [13] as they provide tools for query belief base, reasoning about goals and plans. Applications of these languages are generally to formal problems, to reasoning about the behaviors of agent rather than to real world problems. They don't provide structures for describing the society of agents, hence it is impractical to use these languages to develop multiagent systems.

Another group of agent frameworks have been designed for real-world applications [14, 15, 16]. These frameworks are more practical than frameworks for formal reasoning of agent systems. They support many program constructs to easily describe agent behavior. Partly because of the extended constructs, these languages are lack of the agent oriented aspects, the formal BDI model for reasoning about agent behavior. The SPARK agent framework [17] is an attempt to combine both practical and formal aspects to an agent programming language. Still, there is little support for agent organization in these languages. Developing a multiagent system with these languages is difficult, because agents usually need communication, synchronization and they form organizations (groups) in the system.

One of the first attempt to create a language for multiagent systems is the 2APL programming language by Dastani et al. [18, 19]. It provides constructs for describing sets of agents that form a multiagent system as well as many intellectual constructs for modeling mental state of individual agents. Moreover, 2APL supports the communication between agents by allowing the specification of "communication actions" (send, receive, ...) and sharing environments among agents. However, there are some disadvantages which may make the development of multiagent systems in 2APL tedious. First, 2APL actually requires MAS to be described in 2 different languages (although these languages form a unique language called "2APL"). One language is for description of individual agents with many intellectual constructs for modeling agent's mental state. Another language is for description of the entire multiagent system. Despite the fact that these 2 languages can be recognized by a unique interpreter, programmers must use completely different constructs for the description of multiagent systems and of individual agents: the language for creating multiagent system is very simple and lacks support for dynamic generation of agent's groups (in fact, this language is nearly the same as a simple shell-scripting language that forks many agents simultaneously). Second, the communication patterns that 2APL supports are very simple (i.e., peer-to-peer) and there is no support for complex

computational patterns (e.g., reduction or gathering) (although the agents may share their computation results in a shared variable of an environment). Our language is similar to 2APL because it allows the description of the entire MAS as well as individual agents but it does not require 2 different languages (common constructs such as *for-loop* or *while-loop* can be used everywhere) and it allows the dynamic creation of agents as well as agent groups. More importantly, the set of powerful cooperation primitives (collective operations) integrated in our language supports the description of complex communication and computation patterns.

Thus, existing agent oriented programming languages focus on the internal representation of agent, that is, how to make an agent to be autonomous, how to express the mental state (the intra-agent aspects) of an agent efficiently and easily using constructs provided by the language. They do not pay enough attention to the inter-agent aspects (i.e., communication and cooperation between agents) and abstraction of cooperation between agents. This causes many difficulties in realizing multiagent systems using these languages because cooperation is a very important requirement in MAS. Different from these languages, our cooperation model and AOP language focus not only on the autonomy (the intra-agent aspects) but also on the cooperation of agents (the inter-agent aspects). We do provide constructs for modeling mental state of agent but simultaneously support the communication and cooperation between agents in the system. Therefore, our language and cooperation model are appropriate for developing multiagent systems.

3.2 Multiagent cooperation models

Cooperation model is one of the most important topic in multiagent system research. There are many attempts, such as KQML [20] or FIPA ACL [21], to standardize the format of messages exchanged by agents when agents participate in a cooperation protocol. Unfortunately, these frameworks only show a method for improving interoperability between multiagent systems and they simply do not concern about the methodology for implementation of cooperation protocols.

Cooperation models such as joint-intentions model [2] or teamwork [3, 22] allow the description of global goal for the entire multiagent system and coordination scheme is automatically derived from the team's goal. But it is difficult to ensure the autonomy of each agent because all agents have the same goal and mental state.

Michael Schumacher proposed a model for inter-agent coordination, called ECM [4] and a programming language to specify agent hierarchy. Agents participate in many agent societies, called "blops". Each blop is a group of agents, in which agents can easily communicate with each other and even broadcast messages when they want. However, ECM and its languages do not support the description of mental state of agent, agent itself needed to be specified by another programming language. Our model is similar to ECM in the way of modeling agent groups and supporting for broadcast operation, but it adds extra abstraction to the communication pattern of agents using collective operations. Moreover, by introducing collective operations into the cooperation model, we were able to abstract not only communication patterns, but also sophisticated computational patterns (e.g., reduction, gathering of data, ...).

Recently, there are some cooperation models based on situation calculus [23, 5]. For instance, the requirement/service [23] model allows an agent to send request to another agent (service agent) and the service agent will serve the request. However, it is difficult to describe cooperation schemes with complex actions to synthesize the solution from sub-problem solutions since the cooperation here is restricted in the request and response actions. The Group situation model [5] defines cooperative group with situations

(state of the group at a time) and allows specifying cooperation process for the group. Cooperation process describes the method to achieve the goal situation (i.e., what each agent needs to do). The model is intuitive for describing cooperative systems but it is difficult to generate real execution code from the model, especially code for distributed multiagent systems because the cooperation process is a centralized process which needs to be executed by a master agent (when the cooperation process failed, the system will not function). Our model guarantees that the cooperation process is decentralized which promises the system's fault tolerance. Moreover, arbitrary complex cooperation protocols can be described in our language because the wide range of cooperative actions can be specified by collective operation primitives.

3.3 Agent-oriented methodologies for grid applications

Agent technology has been applied to grid computing to efficiently manage computational resources [24, 25, 26], cope with complex problems such as fault tolerance [27, 28] and support complex inter-process communication [29]. Moreover, many agent-oriented software engineering methodologies have been proposed and used for modeling grid applications [24, 30]. These frameworks are related to our work in the sense that they make the implementation of distributed multiagent systems easier.

MAGE [24] and CoordAgent [25] use the autonomous nature and mobility of agent to efficiently utilize computing resources of grid environment. Unfortunately they have no mechanism to support the realization of communication patterns of agents. The users (programmers) have to create communication schemes from scratch, or they have to use classical methods (C/C++ native programs) to describe their problems. Thus, these frameworks are only a wrapper over the classical programs to adapt to grid environment. Nothing about agent's autonomy and cooperation is utilized to make the problem simpler to solve in these frameworks.

Agent-oriented modeling frameworks for analysis and design of grid applications as described in [30] or [31] provide methodologies to model hierarchies of agents and agent groups, but like the above frameworks, they do not pay attention to agents' cooperation and communication.

The Concordia framework [29] proposed a method for modeling distributed multiagent systems that supports collaboration between agents. Many communication and computation patterns that are supported by our system are also available in Concordia. For instance, in Concordia, an agent can freely broadcast messages to a group of agents. Concordia also supports the master-slave distributed problem solving technique. However, Concordia is not flexible enough to model an arbitrary problem using multiagent system because its functions are restricted to an inadequate communication primitive set (such as broadcast) as well as computational patterns (like master-slave distributed problem solving). Our framework allows the realization of an arbitrary cooperation protocol by providing a powerful cooperation primitive set (as described in the next section). Thus, our proposal is flexible and generalized for modeling sophisticated cooperation problems using multiagent system.

Chapter 4

Abstraction of agent cooperation protocols

In this chapter, we present our first proposal: using collective operations as cooperation primitives for abstraction of agent's cooperation protocols. First, we show basic idea to make the description of complex cooperation protocols easier and well-structured by encapsulating communication patterns in collective operation, a concept in parallel distributed programming. Then, we describe essential collective operations that are very important for description of agent's cooperation protocols. Next, we discuss in detail about method for creating cooperation protocols from these cooperation primitives. Finally, we illustrate the powerfulness of the proposed cooperation primitives by realizing many sophisticated cooperation protocols using collective operations.

4.1 Building blocks for cooperation

Cooperation in multiagent systems involves the sharing of mental state (belief, desire, intention) of agent and exchanging solutions of sub-problems. The main issues one has to cope with to build an intelligent cooperation protocol include division of the goal into smaller tasks, synthesis of solution from sub-problem results, coordination of agents for avoiding unhelpful interactions and maximizing effectiveness [32]. To make the above issues easier and more intuitive, we propose collective operations as building blocks for realization of cooperation schemes. Collective operation is a concept in the Message Passing Interface (MPI) [33], a parallel distributed programming interface, in which many processes simultaneously participate into the execution of some procedure. For example, broadcasting message among agents in a group is a collective operation where the initiating agent may send message to its neighbors and then these neighbors may forward the message to other agents in the group until all agents received the message. MPI defines many collective operations such as `Bcast` (broadcast), `Gather`, `Scatter`, `Reduce` ... as tools for synchronization and coordination of processes/threads of execution. The processes that participate into the collective operations must be in a same group called a "communicator" in MPI (thus, communicator is an agent group in our model). Since collective operations are used to describe the synchronization and coordination of processes in distributed parallel programs, they are suitable for modeling cooperation of agents.

There are two specific cooperative problem-solving activities that are likely to be present: task sharing and result sharing [32]. Task sharing takes place when a problem is decomposed into smaller sub-problems and allocated to different agents. Result sharing involves agents sharing information relevant to their sub-problems. Collective operations

fit very well to the description of these activities. For example, gathering data from all agents (the `Gather` operation) can be applied for collecting result of sub-problems to an agent. On the other hand, the `Scatter` operation (scattering data to all processes in a group) is appropriate for task sharing because after decomposing the problem into smaller tasks, they need to be delivered to agents which can solve the tasks. We will discuss detail about algorithm for constructing cooperation schemes using collective operations in Section 4.3.

Programming with collective operations is much simpler than with only point-to-point operations because collective operations provide much functionality in just a function call. This fact was revealed in parallel programming in the past [34], but in this thesis, it is the first time collective operations are proposed to be cooperation primitives for realizing agent cooperation protocols. Actually, implementing cooperation protocols between agents is similar to programming parallel distributed program by message passing. Therefore, our proposal is a natural application of message passing model into agent oriented programming model. However, there are some challenges that we need to overcome to maintain the autonomous computational model of agent while integrating these collective operations into AOP. In message passing models like MPI, collective operations must be invoked in parallel by all processes in the same group (communicator). In agent oriented programming, this should be avoided because each agent is an autonomous independent process and it is difficult for programmers if they have to invoke collective operations in many agents as the same time. Our model allows the invocation of collective operations by just one agent, other agents will automatically participate into the operations.

There are many advantages when we use collective operations to describe cooperation protocols:

- The description of cooperation protocol will be well-structured with collective operations because collective operations provide much more functionality than low level *send/receive* primitives do. The code written with collective operations will be more intuitive, easier to debug and simpler for verification because the relation between collective operation with low level *send/receive* primitives is similar to the relation of **goto** statement and programming constructs in structured programming (e.g., *if-then*, loop constructs, . . .) [34].
- Using collective operations, global knowledge of the entire multiagent system - a crucial element for intelligent multiagent systems - can be easily derived. Global knowledge is knowledge that involves the entire multiagent system, for instance, the minimum value of a particular property of agents. Data that is distributed across many agents may be considered as global knowledge because gathering of the data involves communication of many agents. Collective operations fit very well for collecting of this kind of data because they are designed for distributed programming. We will give some examples about deriving global knowledge in the following section.
- Cooperation protocols that are described using collective operations can be easily optimized and therefore can achieve high performance than the implementation not using collective operations. This is because collective operations can be implemented as language constructs or in library by experts that have good understanding about the underlying system. Furthermore, some collective operations (such as broadcast) may have supports from hardware [34].

Therefore, utilizing collective operations for description of cooperation protocols between agents may lead to an extraordinary improvement in the quality of the protocols' implementation.

4.2 Set of essential collective operations

In this section, we describe the set of collective operations that are essential for implementing cooperation protocols between agents. First, we give an overview of the set of collective operations to be used as cooperation primitives in our model. Then we discuss about the reason for including these collective operations in detail.

4.2.1 Overview

Collective operations that are likely to be used for building cooperation protocols include `Barrier`, `Bcast` (Broadcast), `Reduce`, `Gather` and `Scatter` and are briefly described in Table 4.1 and Figure 4.1. This set of collective operations is a part of the Message Passing Interface (MPI) [33], a popular parallel distributed programming library. In addition to these collective operations, we also support *send-receive* (point-to-point) operations because as mentioned by Gorlatch, Send-Receive are "considered harmful", but they are still needed in some situations, just like **goto** has not gone away despite the success of structured programming [34].

Table. 4.1. Collective operations

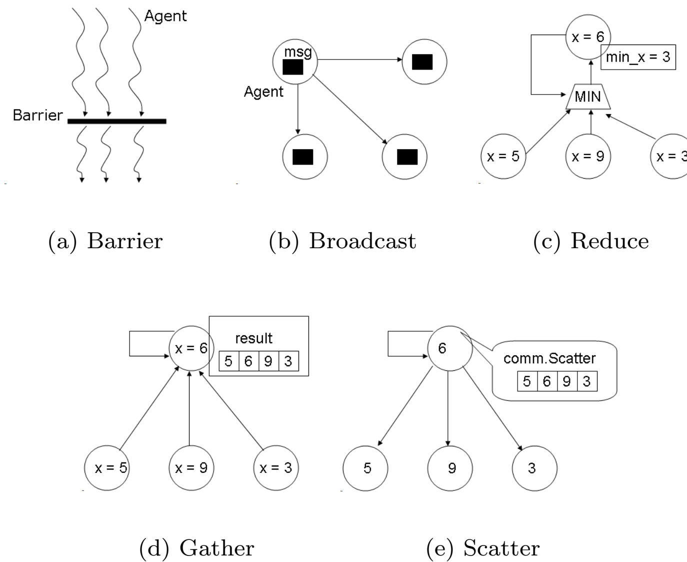
Operation	Meaning	Example
`Barrier`	Synchronizing all agents in the communicator	comm.`Barrier`(*barrier_name*);
`Bcast`	Broadcasting message *msg* to all agents in comm.	comm.`Bcast`(*msg*);
`Reduce`	Evaluate *expression* at each agent and combine the results by applying *operator* to an accumulator value and each element in turn	comm.`Reduce`(*expression*, *operator*);
`Gather`	Evaluate *expression* at each agent then gather the results to an agent	comm.`Gather`(*expression*);
`Scatter`	Scatter the array of messages to all agents in comm	comm.`Scatter`(*array_of_msg*);

We use the concept of "communicator" in MPI to model agent group. An agent can participate into a communicator by invoking a method (or calling a function) like this:

comm = GetCommunicator(*comm_name*, *msg_listener*);

where *comm_name* is a string which is used to identify the communicator (the result of this call is a reference to the communicator object ("comm"), upon which, collective operations can be invoked). Other agents can join this communicator by invoking the above method with the same communicator name. The second parameter, *msg_listener* is the user defined plan (procedure) for processing messages when there are messages from other agents in the communicator that need to be delivered to the agent (it is a callback function which is called by the underlying system). After getting reference to a communicator (the variable "comm" in the above code), an agent can invoke collective operations on that communicator as shown in Table 4.1. By this way, an agent can dynamically create groups of agents (by creating communicators) or join to other groups. This is an essential requirement for modeling complex agent societies.

Collective operations in normal parallel distributed programming library such as MPI [33] must be invoked in parallel by all processes that are participating in the operation. This requirement ensures the efficiency for the execution of collective operations



(a) Barrier (b) Broadcast (c) Reduce

(d) Gather (e) Scatter

Fig. 4.1. Essential collective operations

but it causes difficulty in maintaining the autonomy of each process since it requires all processes invoke the operation at the same time. The proposed system allows collective operations to be invoked by just one agent and other agents will automatically participate in the operations.

There are three types of collective operations in our model: operation for querying data, operation for informing something and operation for barrier synchronization. Operations for querying data are `Reduce` and `Gather` operations. In these operations, the invoking agent wants to know some data from target agents, it requires the target agents to evaluate an expression (specified as the parameter for the operation) and supply the result as parameter for some operator. The action that the target agents take to response to this kind of request is evaluating the expression and this is automatically done by the framework. The programmers do not need to concern about the action of target agents for this kind of operation. The second type of operation (`Barrier` operation) is used to synchronize all agents in a group. The programmers only need to invoke the barrier method with an identical barrier name to synchronize all agents in a group, the actual synchronization is done by the underlying system. The last type of operation is operation for informing some events or knowledge to other agents. `Bcast`, `Scatter` operations are in this group. In these operations, the message (data) from the invoking agent will be passed to the method *msg_listener* of the target agents and it is the responsibility of the programmers to specify which action should be done when the message arrived. By describing *msg_listener* method, agents with reactive reasoning capability can be easily created (i.e., the agents can react to some events that are received from other agents or from the environment).

An agent can leave from a communicator by invoking the method "Leave" on that communicator:

comm.Leave();

Messages in the communicator will not be passed to the message listener any more after the agent leaved the communicator (even though, the agent may participate in the message passing process of the execution environment and act as a router to forward messages to

another agent in the system without awareness of the programmer). Therefore, an agent can freely participate to and leave from a communicator. This provides the programmers the ability to dynamically create/dissolve agent groups and makes the multiagent systems' modeling process easier.

4.2.2 Barrier synchronization

The simplest operation in the operation set is `Barrier` operation, it is illustrated in Figure 4.1(a). It is an essential operation for agent cooperation protocols because in many protocols we need to synchronize the action of agents. For example, in cooperative distributed problem solving, in which many slave (worker) agents solve subproblems and report the result to a master agent for synthesizing the final solution, barrier can be used to wait for slave agents to finish the solution of subproblems. The master agent invokes the barrier operation with a specified barrier name and workers invoke the operation with the same barrier name to notify that they finished the assigned task.
In our model, barrier operation can be invoke as follow:

comm.`Barrier`(*barrier_name*);

where "comm" is a reference to a communicator and *barrier_name* is a string for identification of the barrier operations (it needs to be globally understood by all agents).

4.2.3 Broadcast and scatter

The `Bcast` (broadcast) and `Scatter` operations are used to distribute data across many agents in just one function call (as illustrated in Figure 4.1(b)(e)). These are very important operations for cooperation protocols because cooperation often involves sharing knowledge and exchanging messages. By invoking these operations, the invoked agent can "push" data to other agents (e.g., informing some useful information). The syntax of broadcast and scatter operations is described in Table 4.1.

Note that the syntax and semantics of these operations in our model are different from those in MPI. In MPI, an agent can not invoke broadcast operation alone, the programmers need to invoke the operations in parallel in all agents. In the proposed model, these operations are invoked by just one agent (the agent who wants to push data to other agents). The response action of target agents (the agents that data arrives) can be various because it depends on the *msg_listener* method specified by the programmers. This makes our model more flexible and expressive for describing cooperation protocols.

4.2.4 Reduce and gather

The last type of collective operation is operation for querying data from other agents. These are `Reduce` and `Gather` operations and are illustrated in Figure 4.1(c)(e). These operations make the description of cooperation protocols much more easier in many cases because they abstract a complex communication and computation pattern.

A typical pattern of agent cooperation protocol is the pattern in which several agents contribute their data (information, knowledge, message, ...) to a decision process, the final decision will be determined by a coordinator. This pattern is precisely modeled by operations like `Reduce` or `Gather`. For example, in an auction protocol, where several agents (bidders) propose their price and a coordinator (the auctioneer) decides the final winner based on the proposed prices, `Reduce` operation can be very effectively used: the auctioneer uses reduction on prices with MAX operator to determine the winner. Other protocols such as FIPA's Contract Net Protocol or Recruiting Interaction Protocol can be easily described using `Reduce/Gather` operations.

The syntax of reduce operation in our model is slightly different from MPI:

comm.`Reduce`(*expression*, *OP*);

where *OP* is the operator to be applied and *expression* is a string representation of an expression to be evaluated at each agent to supply data (parameters) for the *OP* operator. Since the operation is invoked locally by an agent, *expression* needs to be transmitted to all agents before the actual evaluation process occurs. That is the reason why *expression* is normally wrapped in the operator "@{}", the operator for converting an arbitrary expression into a literal string (to be sent by the underlying network) in the Yaccai language described in the next chapter. The semantics of the *expression* in gather operation is the same at the *expression* in `Reduce` operation.

Unlike in MPI, the *expression* in our model is a kind of code for mobility. The code for *expression* actually travels from one agent (the invoker) to other agents (the target agents). This kind of code mobility simplifies the data querying process and therefore is important for cooperation protocols.

`Reduce`, `Gather` operations are very appropriate for deriving global knowledge of a multiagent system as stated in Section 4.1. For example, a Vacuum Cleaner agent can know how many agents are in idle state by invoking the following reduce operation:

comm.`Reduce`(@{(**belief query** idle)[0]}, @{Sum});

where "Sum" is an operator of 2 operands which returns the sum of these operands. The belief-base query expression is evaluated at each agent and returns a collection (contains only one element) that is 1 if the agent in idle state and 0 otherwise. The operator "Sum" is applied to the result set in a particular order (the order of the application depends on the reduce algorithm, such as tree-like or linear algorithm).
Another method to achieve the same goal is using the gather operation to gather all idle state of other agents:

comm.`Gather`(@{(**belief query** idle)[0]});

The gather operation returns a collection contains values representing idle state of all agents in the communicator "comm".

4.3 Building cooperation protocols from collective operations

As stated in Section 4.1, cooperation of agents may be built from collective operations. Each collective operation is a building block for constructing cooperation schemes. In this section, we show how the process of reaching agreement and cooperation can be described by using collective operations as primitive actions.

4.3.1 Reaching agreement

```
procedure auction( ) {
   bidders = GetCommunicator( "bidders" );
   bidders.Bcast( proposal );
   winner = bidders.Reduce( @{price}, @{MAX_ID} );
   bidders.Send( winner, inform_win );
   bidders.Recv( winner, confirmation );
}
```

Fig. 4.2. Implementation of an auction protocol by collective operations

As stated by Wooldridge, "the ability to reach agreements (without a third party dictating terms) is a fundamental capability of intelligent autonomous agents - without this

capability, we would surely find it impossible to function in society" [32], reaching agreement is a very important process in multiagent systems. Therefore, the ability of building protocols for reaching agreements is very important for every multiagent system platform.

In our model, many-to-one negotiation is straightforwardly implemented by `Bcast/Scatter` and `Reduce/Gather` operations. For example, in the auction scenario, the auctioneer (the agent who wants to sell goods) uses `Bcast` operation to broadcast the proposal to all bidders (the collection of agents that want to buy the goods), then it can use `Reduce` or `Gather` operation to collect the bidding result from the bidders. In English auctions, the auctioneer may want to know the agent that bids the highest price to allocate goods to that bidder (first-price auction). In this case, it can directly use `Reduce` operation on the agent group with the "MAX_ID" operator:

winner = bidders.`Reduce`(@{price}, @{MAX_ID});

to identify the winner (the expression inside the "@{}" operator is evaluated at each agent in our language). Figure 4.2 shows the pseudo-code for the auction scenario using collective operations.

Many-to-many negotiation can also be done with collective operations such as `AllGather`, `AllReduce` and `AllScatter` (these operations are defined in MPI), but this work has not investigated them elaborately, this should be done in the future.

4.3.2 Cooperation using collective operations

The process of cooperative distributed problem solving contains of 3 stages: 1) Problem decomposition, 2) Sub-problems solution and 3) Solution synthesis [32]. Sub-problem solution is an issue related to the specific problem that can only be solved when we know about the domain of the problem. For problem decomposition and solution synthesis, we can use collective operations to achieve many complex strategies. For instance, a master agent can distribute tasks to slaves by putting tasks into an array and invoking `Scatter` operation. Each agent has an integer identifier (called "rank") so the master can use this identifier to arrange the task array such that the specified agent will receive the desired job (when scattering an array of data, we send each element to an agent in order of their ranks). This strategy is useful when the master has known the capability of each agent so it can assign appropriate task for each agent.

Even when the master does not know the capabilities of each agent, it may use `Bcast` and `Gather` operations to determine which task should be assigned to which agent. It simply broadcasts each task to all agents and ask the agent for the ability of solving the task (by using `Gather` operation). When it has response from all agents, it can assign task as desired.

Solution synthesis (result sharing) can be achieved by `Reduce/Gather` operation. The `Gather` operation is used when the solution of the problem directly equals to the set of sub-problem solutions (for example in QuickSort, the sorted list is the concatenation of sublists (in an appropriate order)). The `Reduce` operation provides a powerful function of deriving global solution from sub-problem solutions. For instance, in the auction problem above, the winner may be easily found by `Reduce` operation with "MAX_ID" operator. Another example, by using SUM operator the master can directly find the distance of the route that is synthesized from many sub-routes that slave agents found.

The `Barrier` operator is useful when the master needs to wait until all agents have completed solving the sub-problem. In this case, the master invokes barrier operation by providing a barrier name (identifier), other agents also invoke the operation with the same barrier name to inform the master about the completion of sub-problem solution.

The skeleton of cooperative problem solving implementation by collective operations is provided in Figure 4.3. In the figure, "Solvable" is a hash table that maps a task into an

```
procedure solve( ){
  for each task in TaskSet {
    workers.Bcast( task );
    Solvable[task] = workers.Gather( @{is_solvable} );
  }
  task_array = create task assignment
                  array from Solvable;
  workers.Scatter( task_array );
  solution = workers.Reduce( @{result}, OPERATOR );
}
```

Fig. 4.3. Cooperative problem solving using collective operations.

array of boolean values whose element i is true if the agent with rank i can solve the task.

4.4 Example implementation of some cooperation protocols

In this section, we show that many complex cooperation protocols in the set of FIPA Agent Interaction Protocols (FIPA-IP) [35] can be easily and intuitively described using collective operations.

4.4.1 FIPA Interaction Protocols (FIPA-IP)

FIPA Interaction Protocols (IPs) are the agent cooperation protocol set proposed by the Foundations of Intelligent Physical Agents (FIPA) [*1] and has been approved as IEEE standard. There are 11 protocols in this set [*2], including 9 standard-status protocols and 2 experimental-status protocols [35]. These protocols deal with pre-agreed message exchange protocols for FIPA Agent Communication Language (FIPA ACL) messages and play an important role in the development of multiagent systems. These protocols range from simple protocols for message exchange (such as FIPA Request Interaction Protocol) to very complex protocols that involve sophisticated computation and communication patterns for reaching agreements (e.g., Auction Protocols, Contract Net Protocol, Recruiting Interaction Protocol, . . .).

In this work, we propose to implement FIPA IPs by using collective operations to obtain well-structured and efficient cooperation protocols. We have pseudo-implemented all of these 11 protocols by collective operations, many of the implementations lead to more readable and intuitive code.

The following subsections show some example implementations of FIPA IPs by collective operations in pseudo-code, the other protocols that are not shown here are available in the Appendix A.

4.4.2 Contract Net Protocol (CNET)

FIPA Contract Net Interaction Protocol (CNET) [36] is a famous protocol for multiagent systems. In this protocol, an agent (the Initiator) takes the role of the manager which wishes to have some task performed by one or more other agents (the Participants) and

[*1] http://www.fipa.org
[*2] As of January 2009

further wishes to optimize the cost to perform the task (the price). The representation of this protocol is given in Figure 4.4 which is based on extensions to UML1.x [37].

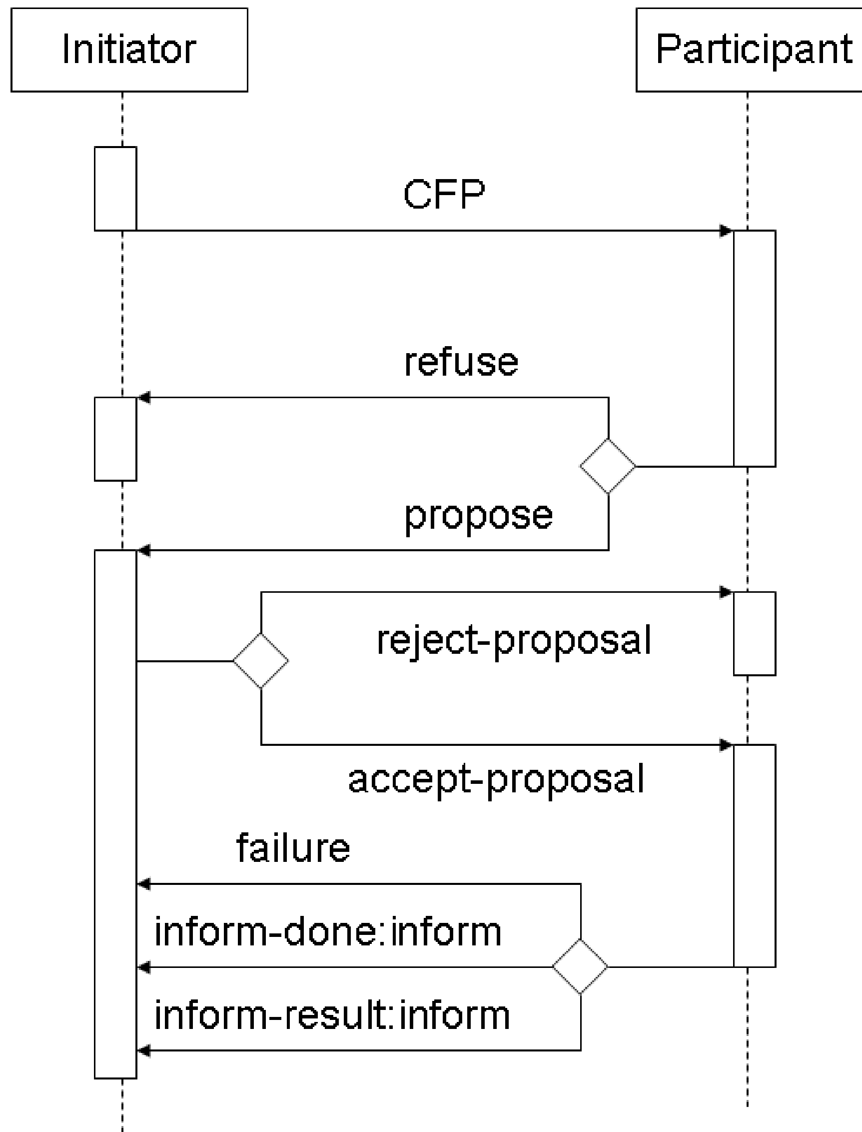


Fig. 4.4. FIPA Contract Net Interaction Protocol (CNET)

Using collective operations, this protocol can be easily described. First, the Initiator broadcasts its task to all participants using `Bcast` operation. When received the task from the Initiator, each participant decides whether it should propose a price or not based on its capability. If the participant can not accomplish the task, it should set its price to infinity. The Initiator then uses `Reduce` operation on all proposed prices to determine the best price (in this case, the minimum price). The pseudo-code for an implementation of this protocol using collective operations is shown in Figure 4.5.

As shown in the code in Figure 4.5, the implementation of FIPA CNET is very simple and intuitive to understand because of two reasons: 1) `Bcast` operation simplifies the announcement of the task (therefore, the Initiator does not need to repeat sending the task), 2) `Reduce` operation abstracts the algorithm for finding best price so the Initiator

```
1  // Initiator
2  plan cnet( task )
3      comm = get communicator that contains all participants ;
4      comm.Bcast( CFP (task, my_id) );
5      winner_id = comm.Reduce( @{belief query proposal}, @{MIN_ID} );
6      comm.Send( winner_id, winning_inform );
7  end plan
8
9  // Participants
10 plan processMsg( msg )
11     if msg is a CFP then
12         initiator, task = get initiator and task from msg;
13         if can_perform?( task, initiator) then
14             proposal = evaluateTask( task, initiator );
15         else
16             proposal = INFINITY;
17         end if
18         add proposal to belief-base;
19     elseif msg is a winning_inform then
20         task_result = perform_task( task );
21         add task_result to belief-base ;
22     end if
23 end plan
```

Fig. 4.5. Implementation of FIPA CNET protocol using collective operations

does not need to concern about how to gather all prices from participants and how to find the minimum value of these prices. Thus, in this example, collective operations abstract not only communication pattern but also computation pattern. The expression "price" that is evaluated remotely at each participant simplifies the data collecting process of the Initiator.

4.4.3 FIPA Recruiting Interaction Protocol

FIPA Recruiting Interaction Protocol [38] is a complicated protocol which is designed to support recruiting interactions in mediated systems and in multiagent systems. In this protocol, an agent (the Initiator) requests the Recruiter agent to find agents that can solve the Initiator's task. The recruiter (a form of broker) agent then finds and assigns the task to appropriate agents and requests these agents to forward result of the task to designated target (the target is determined by the Initiator). The protocol is depicted in Figure 4.6.

The implementation strategy for FIPA Recruiting protocol is shown in Figure 4.7. When received proxy request from the Initiator, the Recruiter finds appropriate agents (the agents that can potentially solve part of the task - the "service agents") and broadcasts the task to these service agents, as shown in Figure 4.7(a). The service agents that can really contribute to the solution of the task perform the requested actions and participate into a new communicator ("comm1"), which is created by the Receiver agent (the agent that will receive the result of the task). The Receiver then performs a `Gather` or `Reduce` operation in the newly created communicator ("comm1") to aggregate all results. The pseudo-code for this progress is shown in Appendix A.

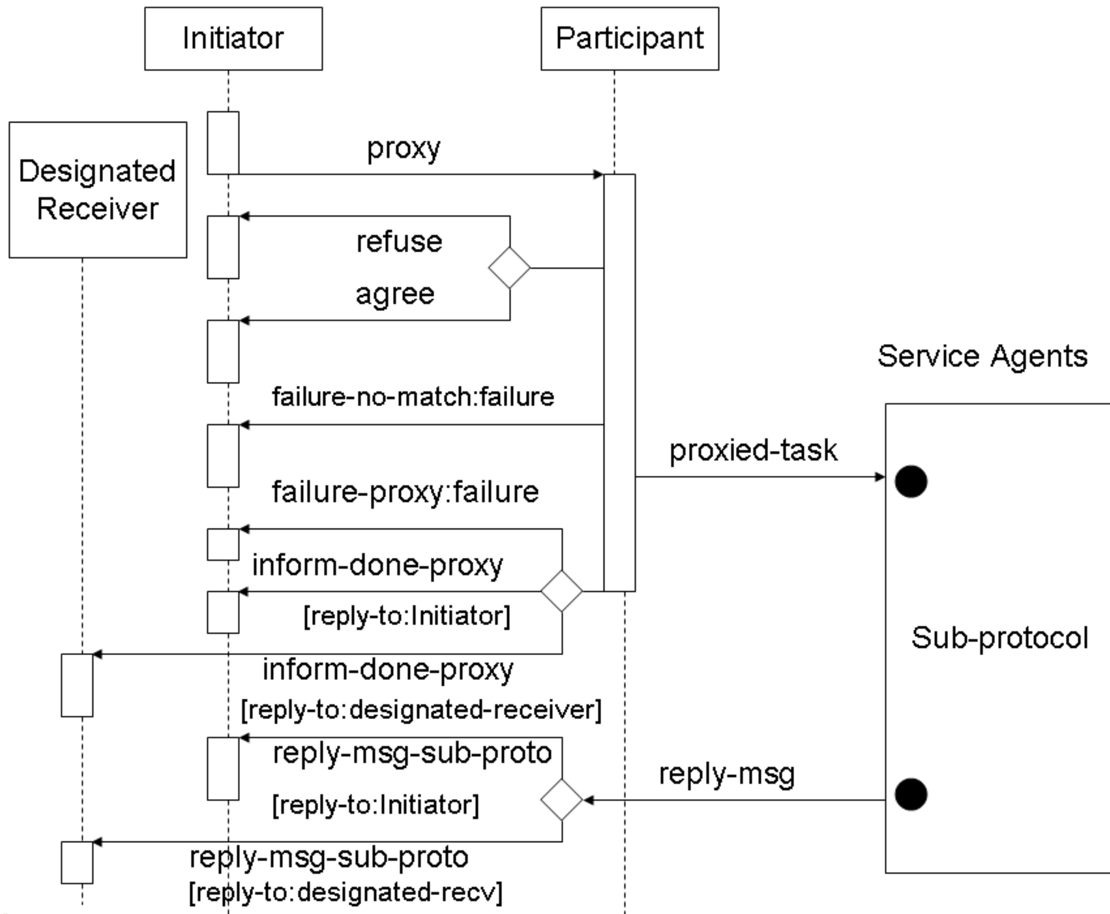
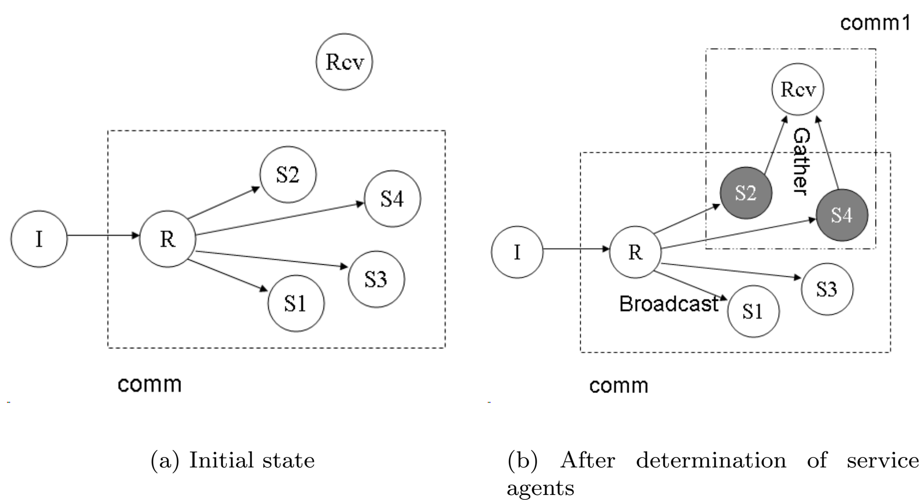


Fig. 4.6. FIPA Recruiting Interaction Protocol



(a) Initial state

(b) After determination of service agents

Fig. 4.7. Implementation of FIPA Recruiting protocol.
(R denotes the Recruiter, I: Initiator, S: service agent, Rcv: Receiver)

By dynamically creating communicators, we can easily perform some filter functions on the set of agents (e.g., the communicator "comm1" in Figure 4.7 is used to get all agents that contributed something to the task achievement process). Therefore, the description of complicated FIPA Recruiting protocol is drastically simpler.

Chapter 5

Communication and execution model

In this chapter, we present our second proposal: a new execution and communication model of agent which allows the implementation of collective operations as well as smoothly integrating cooperation primitives into agent oriented programming model. First, we overview the layered model for each agent and discuss the advantages of this model. Then, we present the programming language named Yaccai that we have designed and implemented to support the execution of the proposed model. We briefly describe syntax of the language and the runtime environment we have implemented to support the execution of multiagent systems written in Yaccai.

5.1 Agent execution and communication model

5.1.1 Communication model

As mentioned in Section 4.1, the integration of cooperation primitives into agent oriented programming is not easy because agents need to be autonomous and independent from other agents while collective operations involve simultaneous execution of some code by all agents. To maintain the autonomy of agents, we need to ensure that the programmers can describe each agent independently from others. Therefore, collective operations may be invoked locally by just one agent, but other agents need to participate into the operations.

In our model, we allow collective operations to be invoked by just one agent locally. This semantics of collective operations is different from semantics in MPI because in MPI, collective operations can not be invoked locally by one agent, they need to be invoked by all agents in parallel. The parallel invocation of collective operations (in MPI) allows these operations to be easily and efficiently implemented. However, it does not guarantee the autonomous computational model of agent, because all agents must participate into the invocation (despite the fact that the final result can be aggregated to just one agent!). The implementation that supports the semantics in our model is not trivial because a locally invoked operation must then be evaluated globally by all agents.

To accomplish this goal, we propose a new communication model for agents, as shown in Figure 5.1. In this model, each agent is divided into two layers: the agent reasoning layer and the message passing layer. The agent reasoning layer (the upper part in Figure 5.1) contains user's code for reasoning process of the agent. The programmers only need to write code for this layer to reflect agent's capabilities. On the other hand, the message passing layer (the lower part in Figure 5.1) contains code for message passing and routing. This code is provided by the runtime system and programmers do not need to worry about this part. This communication model can be viewed as the abstraction of the model in MPI as it hides the message passing layer of MPI under the agent reasoning layer. In MPI, an agent (the receiver) must *actively* participate into a collective operation that is initiated by

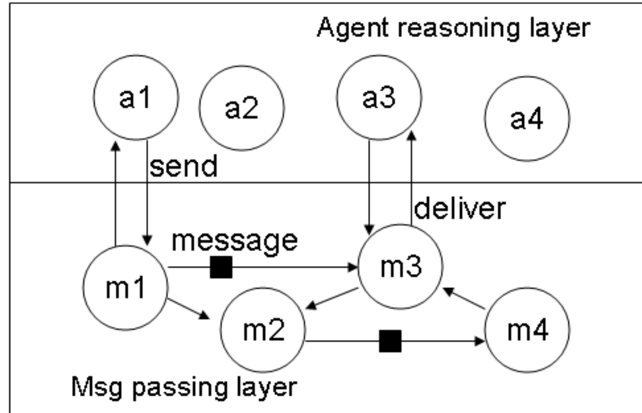


Fig. 5.1. Communication model for a multiagent system.
(the circle "a_i" denotes the reasoning layer, the circle "m_i" represents the message passing layer of agent i)

another agent (even the result of the operation is not related to the desires of the receiver). Thus, an agent needs to pay attention to something that is neither related to its goals (or at least, to it long-term goal) nor in its intentions. This not only causes waste for the effort of programmers, but also is incompatible to the autonomous computational model of agent (in which agents independently and autonomously taking actions). By adding this abstraction, the proposed model allows an agent to *passively* participate into collective operations. The receivers (the agents that need to participate into the operations, not the invoker) simply listen for message from the invoker. The invoker (initiator of a collective operation) sends messages to the receivers to inform about the operation and the data that the operation needs. The receivers can then process the request in background, and therefore, the agent reasoning code of an agent is not disturbed by collective operations that are invoked by other agents. This leads to the ability to invoke collective operations locally at an agent. Table 5.1 gives a comparison between the proposed model and MPI.

Table. 5.1. Differences between the proposed model and MPI

	MPI	Proposed model
Code	All processes have same code	Can be different for each agent
Collective operations syntax	Invoked by all processes (same code)	Invoked by an agent
Can leave from a communicator	No	Yes
Participate into collective operations	Actively	Passively
Reasoning code needs to handle collective operations?	Yes	No

The separation of an agent into 2 layers leads to some advantages. First, it ensures the autonomy of agent because agent's reasoning code can be executed independently in one thread of execution while message passing can be executed in other threads. Thus, the execution of agent's reasoning cycle is not mixed with the execution of message passing, except when explicitly stated by the programmers (by invoking communication primitives to allow an agent to perform the "communication actions"). Second, it supports the im-

plementation of collective operations with different semantics from other message passing models (as described in the previous paragraph). Finally, it allows the optimization of message passing process to be done in library, but not by the agent's programmers. The programmers can enjoy the optimization of message passing without any effort.

5.1.2 Agent execution model

An important issue that we need to cope with to integrate cooperation and communication into agent oriented programming is the description of two kinds of reasoning process in an agent: pro-active and reactive reasoning. Pro-active reasoning refers to the long-term reasoning process of agent. It is the process in which an agent actively determines its goal by itself and takes actions to achieve the goal. On the other hand, reactive reasoning is the process in which an agent reacts to some events or messages from the environment or from other agents. In our execution model, we separate pro-active reasoning and reactive reasoning by using a scheduler which interrupts the pro-active reasoning cycle if needed (i.e., when some messages come or some events occur). The execution model of our system is shown in Figure 5.2.

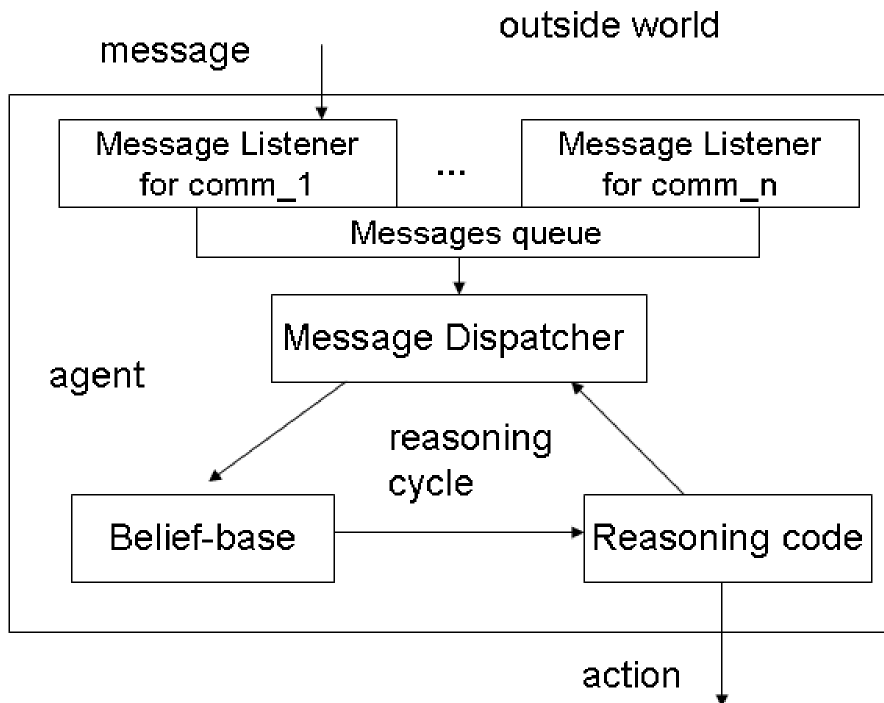


Fig. 5.2. Agent execution model (execution of agent's reasoning layer)

As shown in Section 4.2.1, when an agent wants to participate into a communicator, it invokes a static method called "GetCommunicator" and provides the message processing plan (a callback procedure) as a parameter. Once this statement is executed, the runtime system will create a new thread (which is different from the thread that executes the pro-active reasoning cycle) to listen to messages from the communicator and store incoming messages into a messages queue. In software agent, information about the outside world is often sent as message to agents so an agent "senses" the outside world by listening for messages. The message listener (message processing plan defined by the programmers) can be viewed as a sensor of the agent to sense the outside world. In Figure 5.2, the

message dispatcher (which runs in the same thread with the pro-active reasoning code) fetches messages from the messages queue and delivers to appropriate agent's plan (the message processing plan). The message processing plan may update the belief-base and after the plan is finished, the command scheduler of our runtime system will return control to the pro-active reasoning code (the main reasoning plan for the agent).

This execution model has many advantages. Firstly, it separates the message processing plans from the main plan (pro-active reasoning code). The separation brings an image that main plan and message processing plans are executed in different threads of control and message passing can be seen as asynchronous (in fact, messages are stored in messages queue and message processing plans are given control by our scheduler when the scheduler interrupts the main plan). Secondly, since the execution of the two types of plan (main plan and message processing plan) is actually done by one thread (the main thread of the agent which normally executes the pro-active reasoning code), many intricacies of concurrent programming such as race condition can be avoided. The underlying system automatically maintains consistency of data and synchronizes threads if needed. Finally and most importantly, this execution model is very suitable for modeling autonomous pro-active agent. The message processing plan is responsible for reactive reasoning while pro-active reasoning code could be described in main plan. By this way, programmers can easily model autonomous agent with pro-active reasoning and reactive reasoning capabilities.

5.2 The Yaccai programming language

We have designed and implemented a new agent oriented programming language called Yaccai (Yet Another Concurrent Cooperating Agent Infrastructure) to support the execution of the proposed cooperation primitives.

5.2.1 Overview of Yaccai

Yaccai is an agent oriented programming language which supports not only the autonomous computational model of agent but also the cooperation process in multiagent systems. The language provides constructs for defining classes and agent classes like in normal object oriented programming languages. Agent class is a special type of class that is representing an agent. An agent class contains definitions of methods and plans (plan is similar to a procedure to achieve a goal). Yaccai uses the concept of "communicator" in MPI to model agent groups and provides collective operations as methods of a communicator object. Yaccai wraps each agent (an instance of an agent class) into a separated process which can be executed on a remote host. An agent can uses the **create** operator to create a new agent, the newly created agent may run in a different machine from the machine that invoked the **create** operator. Moreover, Yaccai provides an integrated belief-base for each agent. An agent can use this belief-base to store its beliefs (knowledge about the world, the goal it is pursuing, etc.). An advanced belief-base query language with similar syntax to the syntax of LINQ [39] is also integrated in Yaccai. The mental state of an agent is therefore easily described and the reasoning cycle (cycle of sense-reasoning-act) is automatically realized when the programmers specified a special plan with name "act".

Figure 5.3 shows the brief grammar of Yaccai. The syntax of plan declaration contains "precond" and "maintains" clauses which represent the pre-condition and maintain-condition (condition that must be maintained during the execution of the plan) respectively. The **create** operator allows an agent to fork a new agent (a new instance of an

```
⟨translation_unit⟩ === ⟨include_stmt⟩ * ( ⟨class_decl⟩ | ⟨agent_class_decl⟩ )*
⟨include_stmt⟩ === include ⟨string_literal⟩ ;
⟨access_type⟩ === public | protected | private
⟨class_decl⟩ === ⟨access_type⟩ class ⟨identifier⟩ [ extends { ⟨identifier_list⟩ } ]
                 {
                     ( ⟨method_decl⟩ | ⟨member_var_decl⟩ )*
                 }
⟨agent_class_decl⟩ === ⟨access_type⟩ agentclass ⟨identifier⟩ [ extends { ⟨identifier_list⟩ } ]
                 {
                     ( ⟨method_decl⟩ | ⟨member_var_decl⟩ | ⟨plan_decl⟩ )*
                 }
⟨member_var_decl⟩ === ⟨access_type⟩ [ static ] ⟨identifier⟩ [ = ⟨expression⟩ ]
                 ( , ⟨identifier⟩ [ = ⟨expression⟩ ] )* ;
⟨stmt_block⟩ === {
                 ⟨statement⟩ *
             }
⟨method_decl⟩ === ⟨access_type⟩ [ static ] function ⟨identifier⟩ ( ⟨identifier_list⟩ )
                 ⟨stmt_block⟩
⟨plan_decl⟩ === ⟨access_type⟩ plan ⟨identifier⟩ ( ⟨identifier_list⟩ )
                 [ precond ( ⟨expression⟩ ) ]
                 [ maintains ( ⟨expression⟩ ) ]
                 ⟨stmt_block⟩
// forks an agent at a host specified in ⟨expression⟩
// ⟨identifier⟩ must be an agentclass name
⟨agent_allocation_expr⟩ === create ⟨identifier⟩ ( ⟨expression_list⟩ ) [ @ ⟨expression⟩ ]
// returns a string literal represents the ⟨expression⟩
⟨stop_eval_expr⟩ === @ { ⟨expression⟩ }
```

Fig. 5.3. Excerpt of Yaccai grammar

agentclass) as well as specify the destination host at which the newly created agent is executed. The "stop evaluation operator" (@{*expression*}) returns a literal string which represents the *expression* inside the operator (the *expression* inside the operator is not evaluated, but is recognized as merely a string; this operator is somewhat similar to the quote operator in Lisp). This operator is used for two purposes, the first one is creating a string representation of an expression to send to other agents via the network for remote evaluation, the other one is returning a variable name for binding result of the **match** statement which matches a string with a specified pattern. The full grammar of Yaccai can be found at Appendix B.

5.2.2 Modeling reasoning cycle and belief-base

Yaccai provides a built-in belief-base which agent programs may use to store agent's beliefs. It also provides language constructs for querying and managing the belief-base based on ideas borrowed from LINQ (language integrated query)[39]. The execution of agent can be briefly described in a loop: sense the environment, update its belief-base, determine action to perform and take action, action may change the environment and the agent needs to sense the environment again.

5.2.2.1 Belief-base:

In Yaccai, each agent has its own independent belief-base. There is no common belief-base for the entire multiagent system. This ensures the independence of each agent because the agent does not implicitly share belief-base with other agents in the system. It is important for agent to share knowledge with others, but unlike joint-intentions model where agents automatically (implicitly) have common knowledge, in our model, an agent needs to explicitly query for other agents' knowledge if it wants to know information about those agents. The information that agent received from querying other agents may be stored into its own belief-base for later query or update. Belief-base operations are briefly described in table 5.2. Object can be stored into belief-base by using "**belief fact**"

Table. 5.2. Belief-base operations

Operation	Example
add fact	**belief fact new** {x = 1, y = 2};
query	**belief query** {x, y};
projection	**belief query** {x, y} **as** p{x};
conditional query	**belief query** {x, y} **as** p **where** p.x == 1;
remove	**belief remove** {x, y};
conditional remove	**belief remove** {x, y} **as** p **where** p.x == 1;
join	(**belief query** {x, y} **as** p) **join**
	(**belief query** {y, z} **as** q) **on** p.y == q.y;

expression. Object may be referred to by a pointer or directly created with operator **new**. We also support anonymous object (instance of implicitly declared class). The first example in Table 5.2 will add an object with 2 member variables {x, y} into the belief-base (an anonymous object with two members x = 1, y = 2 is created). The class name or list of member variable names (in case of anonymous class) may be used to identify the class. We call the identifier for a class as "class signature". In the above example, class signature is "{x, y}". The expression "**belief query** *class_signature*" will query the belief-base for all objects that are instance of class described by *class_signature* (as seen in the second example in Table 5.2). The result set of belief-base operations is represented by a collection in our language.
Condition expression (expression after keyword "**where**") may be used in the belief query and belief remove operation to filter objects affected by the operation as described in the forth example in Table 5.2. Moreover, join and projection operations can be applied on the result set of belief query operation. The third example in Table 5.2 will projects the object with two member variables {x, y} onto an object with only one member {x}.
The syntax of belief-base operations is not new because it is introduced in LINQ[39]. But it is the first time these operations are applied to manage the belief-base of agent in an agent oriented language where database query operations are frequently used by the

program.

5.2.2.2 Reasoning cycle

Reasoning cycle of agent is automatically realized when agent program defines a special plan with name "act". A plan is similar to a method in object oriented language, but it is defined inside agent class and has different semantics. In AOP, a plan is considered as a recipe to achieve a goal. Plan usually consists of pre-condition, maintain-condition and plan's body where statements are declared. Pre-condition is condition that needs to be true when the plan is invoked. During the execution of plan, maintain-condition has to be true. When maintain-condition becomes false, the execution of plan is canceled and plan becomes false (i.e, the goal that the plan is trying to achieve could not be accomplished).

```
1  agentclass HelloAgent {
2      public m_comm;
3      public function HelloAgent() {
4          m_comm = Environment.GetCommunicator(
5              "World", MsgListener );
6      }
7      public plan MsgListener( msg ) {
8          id = -1;
9          match msg.Value with {
10             "Hello from", @{id}, "at", @{addr} → {
11                 belief fact new {rank=id, host=addr};
12             }
13         }
14     }
15     public plan act() {
16         myRank = Environment.GetRank();
17         m_comm.Bcast( "Hello from " + myRank +
18             " at " + Environment.Hostname() );
19     }
20 }
21
22 class SimpleMAS {
23     public static function Main() {
24         a1 = create HelloAgent() @ "localhost";
25         a2 = create HelloAgent() @ "somehost.com";
26     }
27 }
```

Fig. 5.4. A simple multiagent system definition

Fig. 5.4 shows an example of a multiagent system definition in Yaccai. Once the agent is created, the constructor will be called and then the plan "act" will automatically be executed. Each agent is mapped to a process (including process in remote host). Each agent in the example simply broadcasts a "Hello" message and its identifier (rank) to others. When an agent received "Hello" message, it stores the host address of the sender into belief-base.

Chapter 6

Evaluation

In this chapter, we evaluate the proposed approach from two aspects: expressiveness of collective operations in the description of agent cooperation protocols and performance of cooperation protocols that are implemented using collective operations. For expressiveness of collective operations, we investigated the simplicity and clarity of cooperation protocols implemented with these cooperation primitives, as well as the powerfulness of the primitive set. For performance of cooperation, we carried out experiments with the Vacuum Cleaner simulation problem and evaluated the performance of Vacuum Cleaner agent team with different computational resources and different problem settings.

6.1 Expressiveness of collective operations as agent cooperation primitives

In this section, we evaluate our model in the aspect of expressiveness, that is, how agent cooperation protocols can be easily and intuitively described with collective operations. First, for simplicity and clarity, we implemented the Contract Net Interaction Protocol (CNET) in Yaccai and compared with the implementation in JADE (Java Agent DEvelopment Framework). Then, for powerfulness of the proposed collective operations set, we pseudo-implemented all of the 11 protocols in the FIPA Interaction Protocols set using collective operations.

6.1.1 Simplicity and clarity

As we have mentioned in Section 4.1, collective operations make the code that implements agent cooperation protocols more readable and well-structured. To make this claim convincing and clear, we give here the comparison between the implementation of the Contract Net Protocol in the proposed programming language, Yaccai, and in JADE [40], a famous agent development framework based on Java [*1].

As described in Section 4.4.2, in Contract Net Protocol, the Initiator wishes to have some task performed by the Participants and further wishes to optimize some criteria (in this case, maximizing a proposal value). The excerpt of code for implementing CNET in Yaccai and in JADE is shown in Figure 6.1 and Figure 6.2, respectively.

In Yaccai, the Initiator code can be read as "broadcast the CFP (call for proposal) to all agents in the communicator "comm" (line 3 in Figure 6.1), and then use `Reduce` operation with "MAX_ID" operator to find the proposal with maximum value (line 5)"

[*1] The executable code of the implementation in Yaccai can be found at Yaccai's homepage: http://www.nue.ci.i.u-tokyo.ac.jp/%7Educ/yaccai , the code in JADE is available in the JADE example code at http://jade.tilab.com/download.php

```
1  // Initiator
2  // my_rank represents the identifier of the agent
3  comm.Bcast( “FIPA_CONTRACT_NET Initiator ” + my_rank + “ ” + task );
4  Environment.Sleep( 5000 );
5  result = comm.Reduce( @{new YValIdPair( proposal, my_rank )}, @{ColOps.MAX_ID} );
6  // accept only the best proposal (the agent whose rank is result.Id )
7  comm.Send( result.Id, “PROPOSAL_ACCEPT Initiator ” + my_rank + “ ” + task );
8  for( i = 1; i <= nResponders; i += 1 )
9      if( responder_ids[ i ] != result.Id )
10         m_comm.Send( responder_ids[ i ], “PROPOSAL_REJECT Initiator ” + my_rank + “ ” + task );
11
12
13 // Responders’ message listening plan for “comm”
14 // proposal, my_rank are instance variables
15 match msg.Value with
16         “FIPA_CONTRACT_NET”, “Initiator”, @{id}, @{task} → {
17             proposal = evaluateAction( id, task );
18         }
19
```

Fig. 6.1. Implementation of Contract Net Interaction Protocol (CNET) in Yaccai

(the Responders simply set their own proposal value in an instance variable (for remote query from the Reduce operation) when they received the CFP as shown in line 17). The “MAX_ID” operator takes two operands which are 2 pairs of integers. “MAX_ID” returns the pair whose value of the first element is bigger than the one in the other pair (in the example, the operator returns a pair of (proposal, my_rank) which has the bigger proposal value).

The Initiator code in JADE needs to create an ACL message whose protocol is set to “FIPA_CONTRACT_NET”, loop through the receivers and add them to the target of the message (these operations are equivalent to broadcasting CFP to all agents) (lines 1–6 in Figure 6.2). Next, the Initiator manually gathers all proposals and implements the algorithm to find maximum proposal (“bestProposal”) (lines 15–26), this part is equivalent to reducing all proposed values to find the best one. Since JADE does not provide remote evaluation ability, the Responders have to manually set the proposal value (line 41) and create the reply message (lines 42-44).

The code in Yaccai is more readable than in JADE because it reflects exactly what the programmers want to do while the code in JADE can not be recognized without thinking. More importantly, the implementation with Yaccai is well-structured because it wraps all destinations into a communicator and wraps the process for gathering all proposals and the algorithm for finding best proposal in an operator. The frequently used operators (such as MAX, MIN, SUM, . . .) can be implemented in library and the programmers do not need to re-implement them. The implementation in Yaccai takes 120 lines of code (from scratch) while the implementation in JADE takes 140 lines even it is using the specific library implementing FIPA CNET (therefore, the total code in JADE is many times bigger than the code in Yaccai).

6.1.2 Powerfulness of the proposed cooperation primitive set

With the proposed cooperation primitive set (`Barrier`, `Bcast`, `Reduce`, `Gather`, `Scatter`), we were able to implement all protocols in the FIPA Interaction Protocol set [35]. Many protocols in this set can be directly mapped to some collective operations, as shown in

```
// Initiator
ACLMessage msg = new ACLMessage(ACLMessage.CFP);
for (int i = 0; i < args.length; ++i) {
  msg.addReceiver(new AID((String) args[i], AID.ISLOCALNAME));
}
msg.setProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET);
msg.setReplyByDate(new Date(System.currentTimeMillis() + 10000));
msg.setContent("dummy-action");

addBehaviour(new ContractNetInitiator(this, msg) {
  protected void handleAllResponses(Vector responses, Vector acceptances) {
    int bestProposal = -1;
    AID bestProposer = null;
    ACLMessage accept = null;
    Enumeration e = responses.elements();
    while (e.hasMoreElements()) {
      ACLMessage msg = (ACLMessage) e.nextElement();
      if (msg.getPerformative() == ACLMessage.PROPOSE) {
        ACLMessage reply = msg.createReply();
        reply.setPerformative(ACLMessage.REJECT_PROPOSAL);
        acceptances.addElement(reply);
        int proposal = Integer.parseInt(msg.getContent());
        if (proposal > bestProposal) {
          bestProposal = proposal;
          bestProposer = msg.getSender();
          accept = reply;
        }
      }
    }
  }
}

// Responder
MessageTemplate template = MessageTemplate.and(
  MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET),
  MessageTemplate.MatchPerformative(ACLMessage.CFP) );

  addBehaviour(new ContractNetResponder(this, template) {
    protected ACLMessage prepareResponse(ACLMessage cfp) throws
    NotUnderstoodException, RefuseException {
      int proposal = evaluateAction();
      ACLMessage propose = cfp.createReply();
      propose.setPerformative(ACLMessage.PROPOSE);
      propose.setContent(String.valueOf(proposal));
    }
```

Fig. 6.2. Implementation of Contract Net Interaction Protocol (CNET) in JADE. (source: The JADE framework http://jade.tilab.com/)

Section 4.4 and in Appendix A. This confirms the powerfulness of the collective operation set. In fact, this set of collective operations is Turing-complete, as proved by Gorlatch et al. in [41], but the proof is beyond the scope of this thesis.

6.2 Performance of cooperation protocols implemented with collective operations

In this section, we evaluate the performance of cooperation protocols implemented with collective operations. We have implemented the Vacuum Cleaner simulation problem in Yaccai: each Vacuum Cleaner team has many agents (each agent is a Vacuum Cleaner) that cooperate with each other using many different protocols. The performance of each cooperation protocol is evaluated based on the amount of garbage that the team collected or the time needed to clean all garbage.

6.2.1 Homogeneous agents

6.2.1.1 Problem settings

Vacuum Cleaner problem is a famous problem in AI in which each vacuum cleaner is an autonomous entity (human being, robot, . . .) that patrols around a space to search and clean dirt (or to collect garbage). Each entity may work independently or cooperate with others. The problem is that what actions each entity should take to totally clean the space as soon as possible and maintain the space in the cleaned status (i.e., when a new dirt appeared, how to discover the position and clean the dirt as quick as possible). In this experiment set, we simulate the Vacuum Cleaner problem using software multiagent teams with homogeneous agents created with Yaccai. Each Vacuum Cleaner is represented as a software agent whose capabilities are move and clean. All agents have the same capabilities so the problem is called "homogeneous Vacuum Cleaner problem". The simulation server constructs a virtual space which is a grid of $m \times n$ cells and contains many cells that have dirt as shown in Figure 6.3. Agent can only move up, down, left or right (one step for each cycle) in the virtual space and it receives information about the current position (e.g., contains dirt or not) from the server. The agent can only clean a unit of dirt in each cycle by sending a "clean" command; a cell may contain many units of dirt.

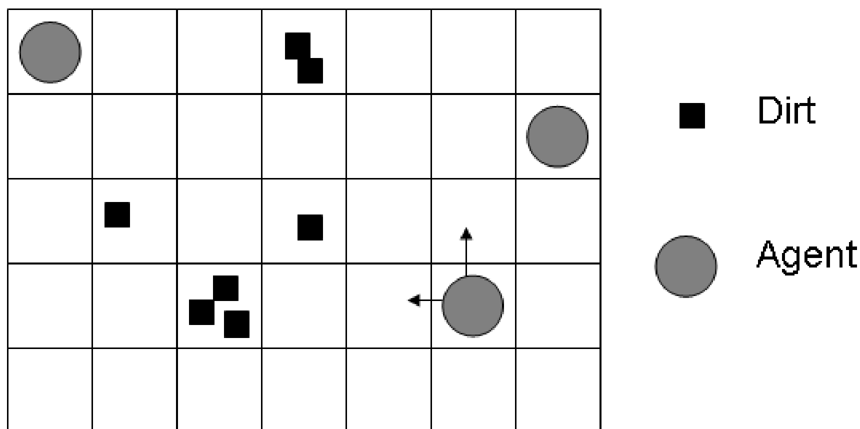


Fig. 6.3. Homogeneous Vacuum Cleaner problem

The server is written in C# while the agent is written in our language (complete source

code for the agent can be viewed at the Yaccai's homepage[*2]). The agents communicate with the server using TCP sockets. At each cycle, the server waits until all agents have issued a command, then it advances the simulation one step (an agent can only issue one command for each cycle). Communication between agents is done by using collective operations. A game lasts for 1000 cycles; at each cycle, the server reports the score of the game by the following formula:

$$Score = \frac{Total\ units\ of\ dirt\ cleaned}{Total\ units\ of\ dirt} \times 100 \tag{6.1}$$

In the first experiment, we created a multiagent system which contains many simple agents: the agents do not cooperate with each other, they only send/receive commands and information from the simulation server. The agents simply scan the virtual space by moving horizontally first then go up/down one row when they could not move in horizontal direction and change the horizontal direction from left-to-right to right-to-left and vice versa. When an agent discovered a cell that has dirt, it immediately sends "clean" commands until the cell is cleaned.

In the second experiment, we created a multiagent system which contains agents that cooperate using `Bcast` operation. The agents use the same strategy in the first experiment to move around the virtual space. When an agent found a cell has dirt, it will broadcast the position of the cell and the units of dirt contained there to all other agents. When received message from other agents, an agent will store the information into its belief-base and determine if it should go to the cell to clean or not. The heuristic to determine to go or not is simple: if the agent is in idle state and the number of units of dirt is bigger than 4 times of the distance between current position to dirt's position, the agent will go to dirt's position. On the way to the dirt, if the agent found another place contains dirt, it will clean the place immediately (and broadcast the position to other agents). When an agent successfully cleaned a position, it also broadcasts the information to other agents. If the agent is not in idle state, it simply stores the cell in its belief-base and when it becomes idle, it will query the belief-base for dirty cells and go to clean. An agent will remove the dirt's position from its belief-base when it received the clean message from other agents.

In the third experiment, we created a multiagent system which contains agents that use similar cooperation scheme to the agent in the second experiment except that when an agent found dirt, it uses `Reduce` operation to know the idle agent nearest to it. Then it sends the information about the cell to that agent only (not broadcast to all agents).

We executed the simulation with 2 scenarios (maps): `map_dense` contains large amount of dirt that broadly distributed across many cells in the virtual space (the map has many cells that contains dirt, so it is dense of dirt), `map_sparse` contains large amount of dirt that comparatively concentrated on a region in the virtual space; each map is a grid with fixed size 20×30 cells. The experiment is carried out on a cluster of 20 machines (Intel Pentium 4, 2.8GHz, 2GB RAM, Linux 2.6.18) connected by Gigabit ethernet to guarantee that each agent is executed on a different host. The score is the average score of 5 times of simulation reported at the end of each simulation (i.e., at the cycle 1000).

6.2.1.2 Result for homogeneous agent teams

Figure 6.4 and Figure 6.5 show the score for the non-communication agents, broadcast agents and reduce agents with 2 maps: `map_dense` and `map_sparse` respectively. When the number of agents increased, the score also increased (the simulations on `map_sparse`

[*2] http://www.nue.ci.i.u-tokyo.ac.jp/%7Educ/yaccai

with 16 broadcast and reduce agents reached the highest possible score (100) after about 900 cycles, i.e., agents do nothing in about 100 cycles left). Standard deviation is very small (as shown in the figures) because we do not use any random parameter. We also performed the simulation with 32 agents run on 16 hosts (each host has 2 agents), but we do not show the teams' score in the graphs because for `map_sparse` they reached the highest score too early before the last cycle.

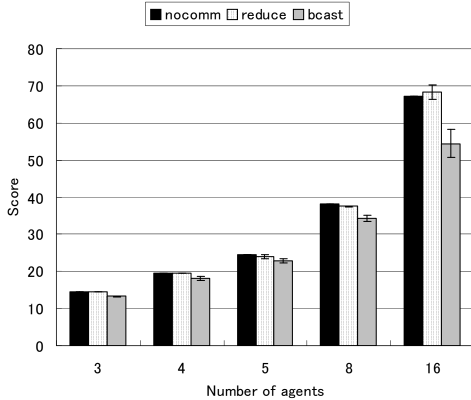


Fig. 6.4. Average score of 5 times of simulation on `map_dense`

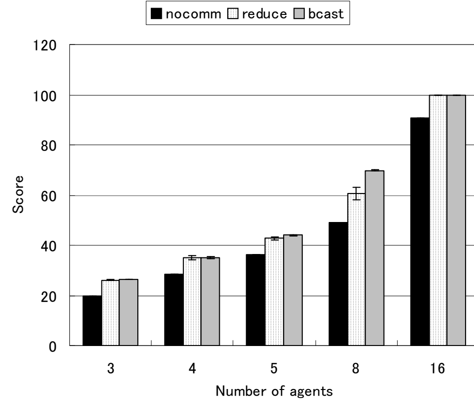


Fig. 6.5. Average score of 5 times of simulation on `map_sparse`

In `map_dense`, the broadcast agents do not perform very well because they can not scan entire the space to find dirt and do too many useless moves. The broadcast agents seem to concentrate on a place at each time (because when received broadcast message they often go to the dirt place if they are in idle state or when they become idle, they will query the belief-base for the cell and go to clean). The reduce agents have the same performance with non-communication agents because when discovered a position has dirt, an agent does not broadcast message to all agents, only one agent is affected by the message. When there are many places with dirt, the agents are busy (not in idle state) and they will not react to messages from other agents immediately so the behavior of the team is similar to non-communication team. In the `map_sparse` map, the situation is different. The broadcast team has the best performance because they do not spend a lot of time in idle moves. They can concentrate on a dirt place immediately when an agent found the dirt place. The reduce team also has relatively good performance because when agents are in idle state, they will react to messages from other agents immediately so the behavior is similar to broadcast team. Agents team without cooperation has poor performance in this situation because agents have to find dirt independently and do many useless moves to scan the virtual space.

The result confirms that collective operations are good for cooperating agents: the agents that use collective operations to cooperate obtained high score when cooperation is important to the problem. In situations where cooperation becomes very important (e.g., in the `map_sparse`), collective operation is very effective because it simplifies the description of cooperation of agents. Even in situations where cooperation is not important (e.g., in the `map_dense`), the agent team uses collective operation may also achieve good performance if it uses appropriate cooperation scheme to reduce the risk of "over-cooperating" (i.e., too concentrated on a region).

6.2.1.3 More about performance of cooperation protocol for homogeneous agents

In the experiments described above, we can see the impact of cooperation by comparing the score that each team obtained, especially in `map_sparse`. In order to make this impact more clearly, we slightly modified the problem settings and investigated the performance of agent teams with `map_sparse`. In these experiments, we modified the server so that it does not automatically send the information about dirt contained in the current cell to the agents. If an agent wants to know the information about the dirt contained in the current cell, it needs to consecutively send 5 "explore" commands to the server. When received "explore" commands in 5 consecutive cycles from an agent, the server will send back to the agent information about current position of the agent: does the cell contain dirt or not, and the unit of dirt (if the cell has). The 3 teams, no communication team, broadcast team and reduce team, are exactly the same as in the above experiments except that when in idle state (i.e., not cleaning or moving to a position where the agent believes that the position has dirt), an agent always explores the current cell after a move. This effectively causes the cost of finding dirt expensive and in turn it makes the value of cooperation more precious.

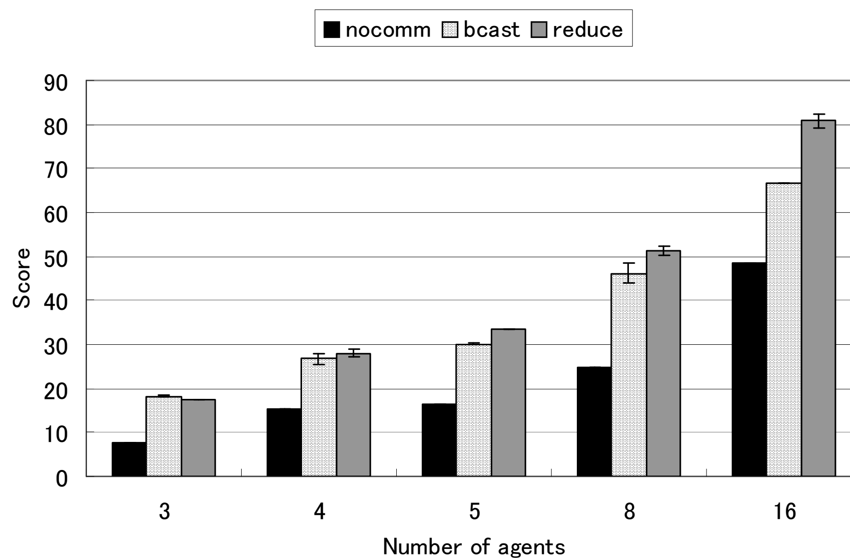


Fig. 6.6. Average score of 5 times of simulation on `map_sparse` with "explore" command.

We carried out these experiments with the modified Vacuum Cleaner server and modified Vacuum Cleaner agent teams on the PC cluster as described in Section 6.2.1.1. The results of these experiments are shown in Figure 6.6. The teams that use `Bcast` and `Reduce` operations obtained very good score compared to the team without cooperation.

6.2.2 Heterogeneous agents

6.2.2.1 Problem settings

In this experiment set, we simulate the Vacuum Cleaner problem with heterogeneous agents. The virtual space is the same as in Section 6.2.1.1, except that the size of the grid is different for each experiment and there are many types of dirt in the space. Each agent can only move up, down, left, right as described in the first experiment set (without "explore" command) in Section 6.2.1.1. However, the capability of cleaning dirt is different: each

agent can only clean a type of dirt as shown in Figure 6.7. The evaluation criteria in these

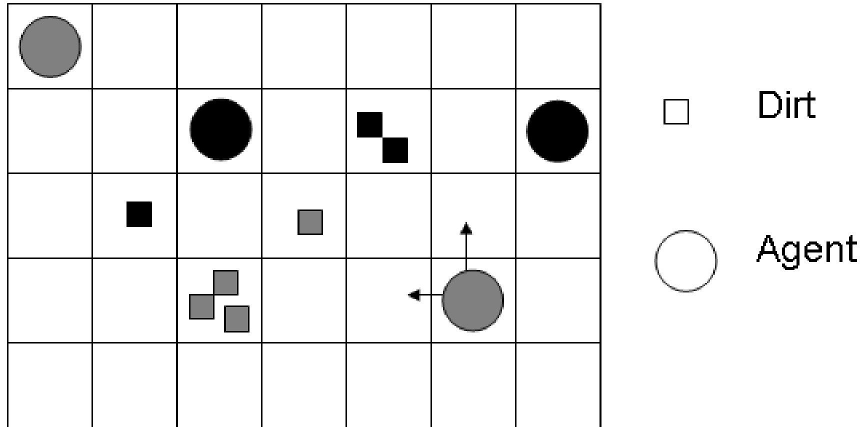


Fig. 6.7. Heterogeneous Vacuum Cleaner problem

experiments are different to the criteria in the first set of experiments. We do not restrict the time of simulation like in the first experiment set, but allow the agents to move and clean until the virtual space is cleaned. This allows us to investigate the "convergent point" in the behavior of each team because the removal of the time restriction is equivalent to simulation with infinite time (under the condition that new dirt does not appear after the virtual space reaching the clean state). The main criterion for evaluation of the performance is the average number of *move* commands that each agent issued during the game as shown in the following formula (note that each cycle an agent can only clean a unit of dirt, so the total number of move commands can be calculated from the cycles for reaching the clean state):

$$nMoves = \frac{(nCycles \times nAgents - nTotalDirt)}{nAgents} \tag{6.2}$$

$nMoves$: average number of move commands
$nCycles$: number of cycles to reach the clean state
$nAgents$: number of agents in the team
$nTotalDirt$: total unit of dirt in the map

It can be easily recognized that the smaller the number of move commands, the better the team performance because with fixed amount of dirt, the team with smaller number of move commands will reach the clean state faster than the team with bigger number of move commands.

In addition to the main criterion, we also investigated the communication cost between agents by measuring the average times each agent sends messages to other agents. For a `Bcast` operation, we assume that the communication cost is aggregated to the invoker, and the number of times sending message is multiplied by the number of agents in the team.

There are 5 different agent teams in this set of experiments: in addition to the team without communication, with broadcast strategy and reduce strategy as described in the first experiment set, we added 2 teams with broadcast and reduce strategy but the agents also broadcast the information about cleaned lines (i.e., after scanning a line and known that the line is cleaned, an agent will broadcast this information to all other agents in the team).

Each map in this experiment set contains 2 types of dirt called type A and type B, respectively. There are 3 agents that can clean type A and 3 agents of type B in each team (unlike the first experiment set, in these experiments we fixed the number of agents). The experiments are carried out with 3 maps: `sparse_20`, `sparse_40` and `random_20`. The map `sparse_20` is a grid of 20×20 cells with sparse dirt distribution while `sparse_40` is a grid of 40×40 cells with the same density of dirt. The map `random_20` is a grid of 20×20 cells with relatively dense dirt distribution. While in the first experiment set, agents are executed on Linux PC cluster, this set of experiments restricts the computation resources: all agents are executed on just one machine (Intel Pentium M, 1.7GHz, 1.0GB of RAM, Windows XP2 Professional).

6.2.2.2 Result for heterogeneous agent teams

Figure 6.8 shows the average number of move commands that an agent issued from the beginning of the simulation until reaching the clean state.

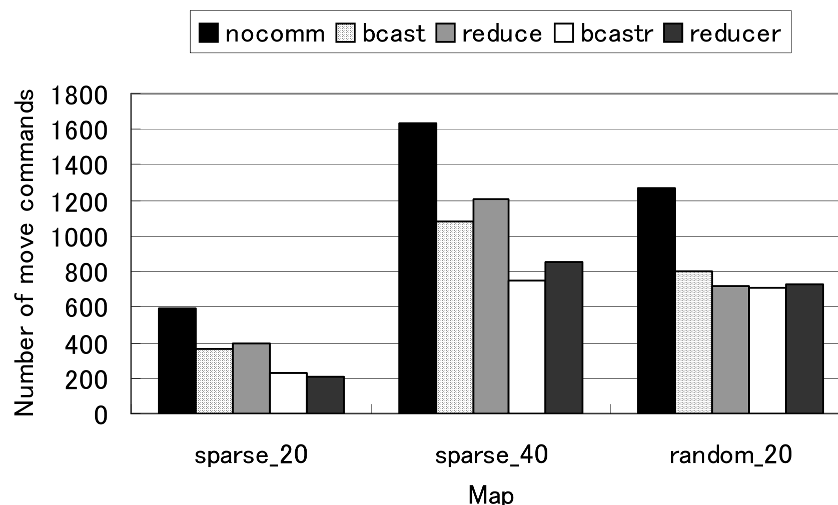


Fig. 6.8. Average number of move commands of heterogeneous agent teams

In sparse maps (`sparse_20` and `sparse_40`), the team without cooperation between agents takes about 1.5 times bigger number of move commands than the team with `Bcast` or `Reduce` operations (without broadcasting scanned lines). Furthermore, when compared to the team with `Bcast` or `Reduce` operations plus broadcasting scanned empty lines, the team without communication takes about 2 times larger number of moves. This confirms that collective operations are very effective for agent cooperation problems. Moreover, by broadcasting scanned empty lines (using `Bcast`, one of the collective operations), agent teams got better performance.

In dense map (`dense_20`), the ratio between number of moves of the team without communication and of the team with cooperation is about 1.7 and the team that broadcasts empty lines information does not perform better than the team without doing this operation. It is because in dense maps, the probability that an agent discovered dirt by itself (i.e., did not rely on information received from others) is higher. Therefore, agents are often busy to clean dirt rather than moving around the virtual space and send information to other agents. That is the reason why the broadcasting scanned line strategy does not work and the ratio between non-communication team with these teams becomes smaller.

Figure 6.9 shows the communication cost of agent teams. The team without cooper-

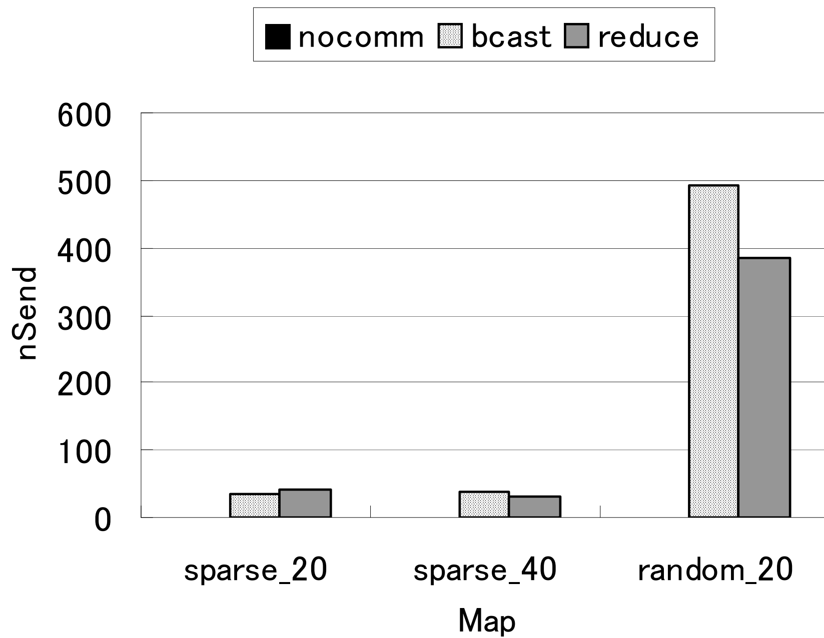


Fig. 6.9. Communication cost that cooperation takes (average number of messages an agent sent for cooperation)

ation does not take any communication cost (the cost is 0 for all maps). With sparse maps, the teams with cooperation do not communicate too much because the probability of discovering dirt position of an agent is low and therefore the probability for sending/broadcasting messages is also small. But note that, with this small amount of communication, the teams' performance improved drastically and therefore the cooperation is very effective. On the other hand, in the dense map, the teams with cooperation protocols send/broadcast a lot of messages because the agents frequently discover dirt positions. However, since other agents are often busy cleaning at other dirt positions, the messages that are communicated often become useless in dense maps.

The experiment results once again illustrate that when cooperation becomes very important (e.g., in sparse maps), collective operations help the agents cooperate with each other very well. Therefore, the proposed model is appropriate for description of cooperative multiagent systems.

Chapter 7

Conclusion and future work

7.1 Conclusion

This thesis proposed an approach for constructing cooperation protocols of multiagent systems from simple cooperation primitives called collective operations, a concept found in MPI [33]. Agent cooperation protocols implemented with these cooperation primitives are intuitive, well-structured and therefore easier to understand and simpler for verification. In particular, we have shown that, many sophisticated cooperation protocols in the FIPA Interaction Protocol set can be directly mapped to several collective operations. This means these protocols can be easily implemented with only few of proposed cooperation primitives and the implementation based on these primitives leads to a more readable code. Collective operation primitives also reduce the effort that the programmers have to make to implement these sophisticated cooperation protocols. Furthermore, constructing agent cooperation protocols from these primitives may yield well-optimized code because optimizations of communication can be done in the implementation of the primitives by the runtime system.

We have also presented a new agent execution and communication model that supports the integration of cooperation into agent oriented programming. Our model not only provides supports for the description of mental state of agents (belief, desire, intention), but it also abstracts the cooperation process that occurs in the entire multiagent system using the proposed cooperation primitives. To prove that the proposed cooperation primitives can be smoothly integrated in agent oriented programming, we designed and implemented a new agent oriented programming language called Yaccai, which supports the execution of collective operations while maintaining the autonomous computational model of agent. Our system is therefore effective for developing individual agent as well as multiagent systems. We carried out many experiments on the Vacuum Cleaner problem with different problem settings and computation resources. The results of these experiments confirm that collective operations help agents to effectively cooperate to reach the goals of the multiagent systems.

7.2 Future work

7.2.1 Revealing more power of collective operations

So far in this thesis, we have discussed many applications of collective operations in the description of agent cooperation protocols. However, we believe that the power of collective operations as agent cooperation primitives is not exhaustively investigated in this thesis. Therefore, in the future, we plan to reveal more applications of these cooperation primitives in order to make the proposed primitives to be standard building blocks for

implementing cooperation protocols. For example, as we have stated in Section 4.1, the many-to-many negotiation in multiagent systems can be modeled using other collective operations such as `AllReduce` or `AllGather` operations. It is important to investigate these abilities of collective operations in the future.

7.2.2 Optimization of communication cost

Our system currently broadcasts messages to all agents at the message passing layers (when `Bcast` operation is invoked) and ignores the messages if the agent is not in the corresponding communicator. In the future, it should be better if we manage the communicator information and only broadcast messages to agents that are in the communicator. To achieve this, we need to use a centralized agent to manage communicator's data or replicate the data at each agent. The preferable method is replication of data, but it may lead to the data consistency problem so the consistency model must be investigated.

7.2.3 Supporting wide-area computing environment

The current implementation of Yaccai with collective operations can work well with PC clusters, but it does not work for wide-area grid computing environments where computational resources are unreliable and network connections between hosts are not always available (e.g., some clusters may be behind firewall or NAT). Still, the model proposed in this thesis is not restricted for only a single machine or PC clusters, it can be used for an arbitrary environment provided that the underlying message passing system supports the communication between agents. Therefore, adapting message passing system of Yaccai for these environments will surely make the proposed model more useful.

7.2.4 Building real-world multiagent applications

In this thesis, we have built a small multiagent application, that is, the Vacuum Cleaner simulation problem. The power of the model can not be convincingly proved without implementing other real-world multiagent systems with complex cooperation protocols. For example, creating a Robocup Soccer [42] or Robocup Rescue [43] team can be an interesting testbed for the evaluation of our system.

Publications

(1) Nguyen Tuan Duc, Ikuo Takeuchi. Collective operations as building blocks for agent cooperation. *International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC08)*, Dec. 10–12, 2008.

(2) Nguyen Tuan Duc, Ikuo Takeuchi. Abstraction of agent cooperation in agent oriented programming language. *11th Pacific Rim International Conference on Multi-agents (PRIMA 2008)*, Dec. 15–16, 2008. (short paper)

(3) グェン トアン ドゥク, 竹内郁雄. 「エージェント指向プログラミングにおける集団操作の応用」(Application of collective operations in agent oriented programming language). 合同エージェントワークショップ&シンポジウム２００８（*JAWS-2008*）, Oct. 29–31, 2008.

(4) Nguyen Tuan Duc, Ikuo Takeuchi. 「Yaccai: A multiagent system development framework」. *IPSJ/SIGSE ソフトウェアエンジニアリングシンポジウム 2008 (SES2008)* ,Sep. 1–3, 2008. (ポスター発表 - poster).

References

[1] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[2] N. R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75(2):195–240, 1995.

[3] David V. Pynadath, Milind Tambe, Nicolas Chauvat, and Lawrence Cavedon. Toward Team-Oriented Programming. In *Proceedings of the 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL '99)*, pages 233–247, 1999.

[4] Michael Schumacher. *Objective Coordination in Multi-Agent System Engineering. Design and Implementation.* Springer Berlin / Heidelberg, 2001.

[5] Minsoo Kim, Minkoo Kim, and Jungtae Lee. Group Situation based Cooperation Model. In *Proceedings of the 2007 International Conference on Convergence Information Technology (ICCIT '07)*, pages 1372 – 1377, 2007.

[6] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI 1973)*, pages 235–245, 1973.

[7] Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Proceedings of The 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (MAAMW 1996)*, pages 42–55, 1996.

[8] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[9] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Formal Semantics for an Abstract Agent Programming Language. In *Proceedings of The 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL '97)*, pages 215–229, 1997.

[10] Mehdi Dastani, Frank S. de Boer, Frank Dignum, and John-Jules Ch. Meyer. Programming agent deliberation: an approach illustrated using the 3APL language. In *Proceedings of The Second International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2003)*, pages 97–104, 2003.

[11] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[12] Mehdi Dastani and Leendert W. N. van der Torre. Programming BOID-Plan Agents: Deliberating about Conflicts among Defeasible Mental Attitudes and Plans. In *Proceedings of The 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 706–713, 2004.

[13] A. Rao and M. Georgeff. Modeling rational agents within a BDI architecture. In *Proceedings of The 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR91)*, pages 473–484, 1991.

[14] François Felix Ingrand, Michael P. Georgeff, and Anand S. Rao. An architecture for Real-Time Reasoning and System Control. *IEEE Expert*, 7(6):34–44, 1992.

[15] Marcus J. Huber. JAM: A BDI-Theoretic Mobile Agent Architecture. In *Proceedings*

of the Third Annual Conference on Autonomous Agents (AGENTS'99), pages 236–243, 1999.

[16] P. Busetta, R. R"onnquist, A. Hodgson, and A. Lucas. JACK - components for intelligent agents in Java. Technical Report TR-1, Agent Oriented Software Pty. Ltd., 1999.

[17] David Morley and Karen L. Myers. The SPARK Agent Framework. In *Proceedings of The 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 714–721, 2004.

[18] Mehdi Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[19] Mehdi Dastani, Dirk Hobo, and John-Jules Ch. Meyer. Practical extensions in agent programming languages. In *Proceedings of The 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007)*, page 138, 2007.

[20] Timothy W. Finin, Richard Fritzson, Donald P. McKay, and Robin McEntire. KQML As An Agent Communication Language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, 1994.

[21] The Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification, Version J. `http://www.fipa.org/specs/fipa00037/`.

[22] Thuc Vu et al. MONAD: A Flexible Architecture for Multi-Agent Control. In *Proceedings of The Second International Joint Conference on Autonomous Agents & Multiagent Systems, (AAMAS 2003)*, pages 449–456, 2003.

[23] Yisong Liu, Lili Dong, and Yamin Sun. Cooperation Model of Multi-agent System Based on the Situation Calculus. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2006)*, pages 424–427, 2006.

[24] Zhongzhi Shi, He Huang, Jiewen Luo, Fen Lin, and Haijun Zhang. Agent-based grid computing. *Applied Mathematical Modelling*, 30(7):629–640, 2006. Parallel and Vector Processing in Science and Engineering.

[25] Munehiro Fukuda, Yuichiro Tanaka, Naoya Suzuki, Lubomir Bic, and Shin ya Kobayashi. A Mobile-Agent-Based PC Grid . In *Proceedings of The 5th Annual International Workshop on Active Middleware Services (AMS 2003)*, pages 142–150, 2003.

[26] Rafael Fernandes Lopes, Francisco José da Silva e Silva, and Bysmarck Barros de Sousa. MAG: A Mobile Agent Based Computational Grid Platform. In *Proceedings of The 4th International Conference on Grid and Cooperative Computing (GCC 2005)*, pages 262–273, 2005.

[27] Rafael Fernandes Lopes and Francisco José da Silva e Silva. Fault Tolerance in a Mobile Agent Based Computational Grid. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops (CCGRIDW'06)*, 2006.

[28] Mohammad Tanvir Huda, Heinz W. Schmidt, and Ian D. Peake. An Agent Oriented Proactive Fault-Tolerant Framework for Grid Computing. In *Proceedings of The First International Conference on e-Science and Grid Technologies (e-Science 2005)*, pages 304–311, 2005.

[29] Mitsubishi Electric ITA. Concordia: An Infrastructure for Collaborating Mobile Agents.

[30] Wei Liu, Zong-Tian Liu, and Yun Li. A modeling framework for agent-oriented analysis and design based on grid architecture. In *Proceedings of The Third International Conference on Machine Learning and Cybernetics (ICMLC 2004)*, pages 210–215, 2004.

[31] Grzegorz Frackowiak, Maria Ganzha, Maciej Gawinecki, Marcin Paprzycki, Michal Szymczak, Myon-Woong Park, and Yo-Sub Han. On Resource Profiling and Matching in an Agent-Based Virtual Organization. In *Proceedings of The 9th International Conference on Artificial Intelligence and Soft Computing (ICAISC 2008)*, pages 1210–1221, 2008.

[32] Michael Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley & Sons, 2002.

[33] The Message Passing Interface. `http://www-unix.mcs.anl.gov/mpi/`.

[34] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):47–56, 2004.

[35] Foundation for Intelligent Physical Agents. FIPA Interaction Protocol Specifications. `http://www.fipa.org/repository/ips.php3`.

[36] Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification, version H. `http://www.fipa.org/specs/fipa00029/index.html`.

[37] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing Agent Interaction Protocols in UML. In *Proceedings of The First International Workshop on Agent-Oriented Software Engineering (AOSE 2000)*, pages 121–140, 2000.

[38] Foundation for Intelligent Physical Agents. FIPA Recruiting Interaction Protocol Specification, version H. `http://www.fipa.org/specs/fipa00034/index.html`.

[39] The LINQ project. `http://msdn.microsoft.com/en-us/netframework/aa904594.aspx`.

[40] Fabio Bellifemine, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa. JADE: A software framework for developing multi-agent applications. Lessons learned. *Information & Software Technology*, 50(1-2):10–21, 2008.

[41] Jörg Fischer and Sergei Gorlatch. Turing Universality of Recursive Patterns for Parallel Programming. *Parallel Processing Letters*, 12(2):229–246, 2002.

[42] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. RoboCup: A Challenge Problem for AI. *AI Magazine*, 18(1):73–85, 1997.

[43] Tomoichi Takahashi, Ikuo Takeuchi, Tetsuhiko Koto, Satoshi Tadokoro, and Itsuki Noda. *RoboCup Rescue* Disaster Simulator Architecture. In *Proceedings of The Robot Soccer World Cup IV (RoboCup 2000)*, pages 379–384, 2000.

Appendix A

Implementation of some FIPA protocols using collective operations

A.1 Implementation of the FIPA Recruiting Interaction Protocol

The pseudo code for implementation of FIPA Recruiting Interaction Protocol is shown in Figure A.1. The code contains plan for initiating Recruiting actions of the Initiator, and message process plans for the Recruiter, the Receiver and Service Agents.

A.2 Implementation of other FIPA Interaction Protocols

The pseudo-code for implementation of other FIPA Interaction Protocols using collective operations can be found at the homepage of Yaccai:
http://www.nue.ci.i.u-tokyo.ac.jp/%7Educ/yaccai/

```
1  // Initiator
2  plan init_recruit( task, cond, Receiver )
3    send to Recruiter the recruiting_request;
4    if got recruiting_agreed from the Recruiter then
5       sa_set = get service agents set from Recruiter;
6       if empty?( sa_set ) then
7          failure_no_match
8       end if
9    end if
10 end plan
11
12 // Recruiter
13 plan processMsg( msg )
14   if msg is recruiting_request then
15      task, cond, Initiator, Receiver = parse from recruiting_request;
16      if proxy_permitted?( Initiator, task, Receiver ) then
17        send recruiting_agreed to Initiator;
18        comm.Bcast( recruiting_request );
19        comm1 = GetCommunicator( "recruiting_comm" );
20        service_agents = comm1.Gather( @{my_id} );
21        comm1.Leave();
22        if empty?( service_agents ) then
23           send empty set to Initiator;
24        else
25           send all service_agents to Initiator;
26           send inform_done_proxy to Receiver;
27        end if
28      else
29        send permission_denied to Initiator
30      end if
31   end if
32 end plan
33
34
35 // Service agents
36 plan processMsg( msg )
37   if msg is recruiting_request then
38     cond, task, Receiver = parse from recruiting_request;
39     if true?( cond ) and can_perform?( task ) then
40       comm1 = GetCommunicator( "recruiting_comm" );
41       set member variable task_result = perform( task );
42     end if
43   end if
44 end plan
45
46 // Designated receiver
47 plan processMsg( msg )
48   if msg is inform_done_proxy then
49     comm1 = GetCommunicator( "recruiting_comm" );
50     result_set = comm1.Gather( @{task_result} );
51   end if
52 end plan
```

Fig. A.1. Implementation of the FIPA Recruiting Interaction Protocol

Appendix B

Grammar of the Yaccai programming language

```
translation_unit ::= include_stmt* (class_decl | agent_class_decl)*
include_stmt ::= "include" string_literal ";"
access_type ::= "public" | "protected" | "private"
class_decl ::= access_type "class" identifier ["extends" "{" identifier_list "}"]
        "{"
             (method_decl | member_var_decl)*
        "}"
agent_class_decl ::= access_type "agentclass" identifier  ["extends" "{" identifier_list "}"]
        "{"
             (method_decl | member_var_decl | plan_decl)*
        "}"
member_var_decl ::= access_type ["static"] identifier [ "=" expression]
                ( "," identifier ["=" expression])* ";"
method_decl ::= access_type ["static"] "function" identifier "(" identifier_list ")"
                stmt_block
stmt_block ::= "{"
          statement*
        "}"
plan_decl ::= access_type "plan" identifier "(" identifier_list ")"
        ["precond" "(" expression ")"]
        ["maintains" "(" expression ")"]
          stmt_block
agent_allocation_expr ::= "create" identifier "(" expression_list ")" ["@" expression]
stop_eval_expr ::= "@" "{" expression "}"
belief_expr ::= "belief" (belief_query_expr | belief_remove_expr | belief_add_expr)
        ( "join"  belief_expr  "on"  expression
          ["as" belief_projection] ["where" expression] )*
belief_add_expr ::= "fact" expression
belief_projection ::= identifier [ "{" identifier_list "}" ]
class_signature ::= string_literal
belief_query_expr ::= "query" class_signature ["as" belief_projection ["where" expression] ]
belief_remove_expr ::= "remove" class_signature [ "as" belief_projection ["where" expression] ]
object_allocation_expr ::= "new" ( identifier "(" expression_list ")" ["{" expression_list "}"] |
                "{" assignment_list "}")
u_expr ::=  primary | "-" u_expr | "+" u_expr | "!" u_expr | "~" u_expr
        | object_allocation_expr
        | agent_allocation_expr
        | belief_expr
        | stop_eval_expr
```

```
primary ::= atom   trailer*
atom ::= literal | identifier | "(" expression ")"
trailer ::=  "(" expression_list ")"
        | "[" expression "]"
        | "."  identifier
expression ::= assignment_expression
assignment_expression ::= (primary assignment_operator)* or_test
assignment_operator ::= "=" | "+=" | "-=" | "*=" |  "/=" | ..... (same as other language)
or_test ::= and_test ( "||" and_test)*
and_test ::= bo_expr ("&&" bo_expr)*
bo_expr ::= bx_expr ( "|" bx_expr)*      //(bit or)
bx_expr ::= ba_expr ("^" ba_expr)*       // (bit xor)
ba_expr ::= comparison ("&" comparison)*   //bit and
comparison ::= s_expr (comp_operator s_expr)*
comp_operator ::= ">" | "<" | "!=" | ">=" | "<=" | "=="
s_expr ::= a_expr ( ( ">>" | "<<" ) a_expr )*    // (shift expression)
a_expr ::= m_expr (("+" | "-") m_expr)*
m_expr ::= u_expr (("*" | "/") u_expr)*
expression_stmt ::= expression ";" | ";"
statement ::=  expression_stmt
        | if_stmt
        | while_stmt
        | for_stmt
        | do_while_stmt
        | foreach_stmt
        | continue_stmt
        | break_stmt
        | return_stmt
        | label_decl_stmt
        | switch_stmt
        | stmt_block
        | match_stmt
if_stmt ::= "if" "(" expression ")" statement ["else" statement]
while_stmt ::= "while" "(" expression ")" statement
for_stmt ::= "for" "(" expression_list ";" expression_list ";" expression_list ")" statement
do_while_stmt ::= "do" statement "while" "(" expression ")" ";"
foreach_stmt ::= "foreach" "(" identifier "in" expression ")" statement
continue_stmt ::= "continue" ";"
break_stmt ::= "break" ";"
return_stmt ::= "return" [expression] ";"
label_decl_stmt ::= identifier ":"
switch_stmt ::= "switch" "(" expression ")" "{" (label_decl_stmt | statement)* "}"
match_stmt ::= "match" "(" expression ")" "with" "{"
                regex_expr_list "->" statement
                ( "|" regex_expr_list "->" statement )*
        "}"
regex_expr_list ::= "*" | expression ["+" | "?" | "*"] ("," expression ["+" | "?" | "*"])*
identifier_list ::= <empty> | identifier ("," identifier)*
expression_list ::= <empty> | expression ("," expression)*
literal ::= int_literal | double_literal | string_literal | char_literal
```