

Abstract

Agent-Oriented Software Engineering (AOSE) covers issues on developing systems with software agents. There are many techniques, mostly agent-oriented and object-oriented, ready to be chosen as building blocks to create agent-based systems. There have been several AOSE methodologies proposed intending to show engineers guidelines on how these elements are constituted in having agents achieve the overall system goals. Although these solutions are promising, most of them are designed in ad-hoc manner without truly obeying software developing life-cycle fully, as well as lacking of examinations on agent-oriented features. To address these issues, we investigated state-of-the-art techniques and AOSE methodologies. By examining them in different respects, we commented on the strength and weakness of them. Toward a formal study, a comparison framework has been set up regarding four aspects, including concepts and properties, notations and modeling techniques, process, and pragmatics. Under these criteria, we conducted the comparison in both overview and detailed level. The comparison helped us with empirical and analytical study, to inspect the issues on how an ideal agent-based system will be formed.

A comparison of Agent-Oriented Software
Engineering frameworks and methodologies

Chia-en Paul Lin

Department of Computer Science

University of North Texas

Oct 2003

Contents

1	Introduction	1
2	Survey of Agent Techniques	8
2.1	Terminology of intelligent agent	8
2.2	Techniques using Object-Oriented Development	11
2.2.1	Object for Agent	11
2.2.2	Object Oriented Programming Patterns	14
2.2.3	Object-Oriented Programming tools	17
2.3	Techniques using Agent-Oriented Development	20
2.3.1	Theory of modeling	20
2.3.2	BDI architecture	22
2.3.3	Agent UML	23
2.3.4	Patterns for agents	26
2.3.5	Agent-based Platform	28
3	Agent-Oriented Software Engineering Methodologies	33
3.1	Tropos	33
3.2	Gaia	39
3.3	MaSE	44
3.4	Extending UML for Modeling and Design Multi-Agent Systems	49
3.5	Engineering Multi-Agent Systems using Object-Oriented techniques	52

4	Summary	57
4.1	Comparisons in overview level	61
4.2	Comparisons in detailed level	72
4.3	Summary	78
5	Conclusion and Future Work	79
5.1	Conclusion	79
5.2	Future Work	81

List of Tables

1.1	Agent Applications Catalogs	2
4.1	Concepts and properties Comparison A	61
4.2	Summary of Methodolical Comparison A	63
4.3	Notations and modeling techniques Comparison B	64
4.4	Summary of Methodolical Comparison B	66
4.5	Process Comparison C	67
4.6	Life-cycle coverage of methodolgies	68
4.7	Summary of Methodolical Comparison C	69
4.8	Pragmatics Comparison D	70
4.9	Summary of Methodolical Comparison D	71
4.10	Agent-based elements involved in life-cycle process (G: Goal, R: Role, A: Agent)	76

List of Figures

2.1	MAS Framework Model	12
2.2	Interaction Diagram of Mediator Pattern	14
2.3	UML Diagram of Mediator Pattern applied to Agent Property Model	15
2.4	Structure of Role Object Pattern	16
2.5	AspectJ Programming Structure of Aspect Class	19
2.6	Social Level Model	20
2.7	AUML package model	24
2.8	AUML package and template model	25
2.9	extensions to support concurrent multithread model	25
2.10	Seven Layered Agent Pattern	27
2.11	JADE Agent Platform	30
2.12	JADE Agent Platform Architecture	30
3.1	Notations used in Tropos Methodology	34
3.2	Relationship of Actor Dependency used in SD Modeling	35
3.3	Means-Ends Analysis and Decompositions used in SR modeling	36
3.4	Gaia Model and its abstract/concrete conceptual elements	40
3.5	Role template in Gaia	40
3.6	Gaia Protocol template and Interaction Model	41
3.7	Agent Model of Gaia	42
3.8	Gaia acquaintance model	43
3.9	MaSE Model	44

3.10	MaSE Agent Role Model	46
3.11	MaSE Agent Class Model	46
3.12	MaSE Deployment Model	48
3.13	Graphic notations for Extending UML Modeling	50
3.14	Agent Domain Model for Extending UML Modeling	50
3.15	Agenthood Model	53
3.16	Agent prototype in Agenthood Model	54
3.17	UML diagram for Agenthood Model	54
3.18	Agent Aspects Diagram of Agenthood Model	55
3.19	Role aspects paradigm for Agenthood UML diagram	56

Chapter 1

Introduction

Within the last decade, Information Technology (IT) has been largely brought into the everyday aspect of people's lives. The power of IT has increasingly influenced software development in every field of application from personal computing, to home appliances, to industrial systems, and more. As well as accompanying the growth of the world-wide Internet, developing a software system involves an overall challenge of coping with more embedded computational complexities of distribution, multi-tasking, and real-time. Moreover, it is becoming increasingly difficult for users to manually control and command software systems. There is a need for sophisticated, automatic, intelligence to be brought into system development. Agent-oriented computing thus becomes an interesting research field in computer science. "Agent", as the name suggests, is a programmed system under certain authority that can handle tasks autonomously with intelligence. Systems using Agents are becoming one of software's developing mainstreams.

Treated as a new technique in software development, what Agents are able to provide beyond existing methodology must be discussed. Retrospects of the types of software systems, complexities are inherited from simple functional systems and reactive systems, to distributed concurrent systems; making software systems even more complex. As Jennings and Wooldridge suggest [2], three classes of systems should utilize Agent-oriented

computing as a new technique: Open Systems, Complex Systems, and Ubiquitous Computing Systems. First, the feature of an Open System is that its components or structure could be changing dynamically. For example, the Internet is a highly open software environment. Its size and complexity is increasing exponentially. Software systems live on it; hence mechanisms to adapt it are needed. Second, Complex Systems require for large integration by sophisticated components with unpredictable results. Usually, modularity and abstraction are solutions to ease this complexity. Agents could play a role to coordinate these different modules toward a solution. Additionally, human-interaction with computer systems will become a bottleneck if users must be involved in all steps of software processing. In order to benefit from autonomous control and reduced running costs, system functions have to be performed automatically by Agents. Recently, these Agent-oriented considerations have been steadily accepted into software design.

There are several application domains of software system functions that are appropriate to use Agent techniques. To observe the breadth and variety of Agent applications, Jennings and Wooldridge [2] proposed a catalogue to classify application domains for Agent-based systems. Table 1.1 summarizes the catalogue, in which we traverse examples in different domains according to the classification.

Table 1.1: Agent Applications Catalogs

Application Domain	Application Fields
Industrial	Process Control, Manufacturing, Air Traffic Control, Network and Telecommunication Management, Transportation System
Commercial	Information Management, Electronic Commerce, Business Process Management
Medical	Patient Monitoring, Health Care
Entertainment	Games, Interactive Theater and Cinema

- Industrial domain:

Industrial domain is the first to adopt Agents. Among the industrial applications, Process Control is apt to using Agent-oriented approach, since they are autonomous, reactive systems. One of the earliest field-tested, multi-agent systems is ARCHON, which is a software platform on which to build Multi-Agent systems. Communication, decision-making, information-maintaining, and expertise are four components of Agents in the system.

In manufacturing, hierarchy workflows are modeled to operate different production units. The primary abstraction of these units is a resource. Agents among these modules are used to negotiate and cooperate with different peers in the system. For example, YAMS (Yet Another Manufacturing System) is a manufacturing control application, which applied Contract Net Protocol. The goal of YAMS is to efficiently manage production at plants. In a factory, work cells respond to different tasks. These work cells could be grouped into flexible manufacturing system(FMS), which will provide specific functionalities. Agents are responsible for controlling changing parameters , such as charges in resources, and efficiently arrange FMS to get better production results.

Another well-known application is the air traffic control system OASIS, which controls aircraft as well as traffic control operations. In this case, Agents could be used to model real-world situations, autonomous processes in allocating aircraft, reckoning information, and manipulating control. It has been implemented on the dMARS system, and tested at Sydney airport in Australia.

In Network and Telecommunication Management [3], Agent-based approaches are used to manage the complexities resulting from network communications.

Agents can provide intelligence in interactions and negotiations to effect network control, transmission and switching, and service management. Another example is the application to Traffic Control, which lend naturally to Agents in the systems of distributed nature.

- Commercial Domain:

As information grows, Agents can play a role in handling information overloads, including information filtering and information gathering. As more and more information becomes available from the Internet, usually only a small portion of that amount is all that we need. For example, many SPAM emails arrive among legitimate, important ones. Checking each of them to discover which one to read is not efficient. Agent-based email filters should separate the junk from essential messages to the receiver. On the other hand, searching valuable data from the huge and up-to-date information is quite a heavy task, and it is not economically efficient for human users to spend time on the search. Agents should be used for searching tasks based on the query criteria getting the precise results back.

Electronic Commerce is another emerging field where trading is accomplished online. Buyers and sellers negotiate with each other to set the final price for their goods. The processes involve high levels of interactions. There is no reason why Agents cannot handle it more specific tasks by taking the goals their users set and accomplishing the sale process. For example, Kasbah is a virtual marketplace. Users create their Agents to buy or sell goods on their behalf.

In Business Process Management, decision-making is usually made from collecting information from different departments. Managers reference this data, and make the final decision based on the rules of the company. However, up-to-date information is one key to making good decisions. Agents can help with observat-

ing, gathering data in time, and responding in a reasonable amount of time. For example, the ADEPT project builds business processes as negotiation and service-providing Agents. Each Agent represents different departments in a company and cooperates with other Agents in order to run the business.

- Medical Domain:

In Patient Monitoring, Agents can assist doctors who are unable to monitor their patients continuously. While the patients are cared for by nurses, it is not sufficient to handle an emergency situation with appropriate expertise requiring doctors. Agents can collect information for patients, share information with the medical teams, and collect expertise from different areas to organize patient health care. For example, the Guardian system is to help patient management in a Surgical Intensive Care Unit (SICU). Three types of Agents are in the system. Perception/action Agents are responsible for the interface between the system and real-world situations. Reasoning Agents organize system decision making processes. The top-level controller of the system is the Control Agent. These Agents share knowledge by using a blackboard data structure. A similar situation is applied to health care applications. The Agent-based system is responsible for integrating patient management processes, which traditionally involves many specialists for inspecting and making decisions.

- Entertainment Domain:

The entertainment domain features animated and virtual characters. In game world, Agents play roles to participate in as actors. These actors are programmed in terms of behavior for each character and the rules to select behavior. Likewise,

interactive theater allows users to play a role along with artificial characters in the scene. Agents can be trained to participate in the interactive cinema.

In summary, aspects of different Agents could be used to analyze Agent-based applications. Agents can be categorized to perform simple tasks, like a gopher. They can also provide autonomous services to assist a user without being asked. Agents play roles in terms of different expertise and functionalities while supporting services to the application. They are responsible for making good decisions to execute those activities. Agents can work alone, but most importantly they cooperate with each other in some applications. Agents are software components in the system that are viewed as many individuals living in a society working together.

More Agent applications are becoming available – from personal assistance devices, software helps and tutorials, to smart home environmental management. Agent applications have become almost ubiquitous.

With the trends and needs for Agents, building industrial-strength applications in a robust, fault-tolerant, and flexible way seems a demanding requirement. From traditional functional processes to mainstream object-oriented paradigm, studies in software engineering already provide abundant methods and tools for sophisticated development. Standards for traditional software have been proposed and tested for efficient analysis and design. However, there is not much agreement on how to build a comprehensive Agent-oriented system. Observed from the intrinsic properties of Agent applications, it is not a trivial task without a necessary infrastructure. From the landscape of researchers in software engineering, solving a problem should encompass steps from problem realization, to requirements analysis, to architecture design and implementation. To prove what the built system is sound, these steps need to be implemented within a life-cycle process including testing, verification, and reengineering. While working on software development, documents, and specialized tools are required to reach efficient results. When

Agents are explored as a new research field, the properties, theories, methodologies, and computing tools of Agents are the primary foundations toward reliable development.

Fortunately, many academic researchers are conducting experiments in exploring the new Agent field piece-by-piece. However, some of this work needs to be refined and enumerated with analyses while others need to be integrated from their theoretical common grounds to all shapes of development. To do this, we should explore some building blocks from Agent bases, fundamental theories, and methods as well as available assistance from notations of analysis, architecture design, and implementation toolkits. It is still a distant hope to treat these exciting tactics for practical use and integrating them as methods to gain a stable system. However, with different shrewd pursued by researchers, the road towards mature Agent software development may be reached.

To gain a better understanding of Agents so as to realize the way to study and develop the field, some investigations and comparisons in Agent technology should be conducted. In Chapter 2, we explore the concept of Agents and examine some widely used techniques, which could be applied in building Agent systems. In Chapter 3, different aspects in Agent-oriented methods are discussed along with empirical study of some well-known methodologies. In Chapter 4, a summary of comparisons between these widely used methodologies will be provided. In Chapter 5, some conclusions and proposed future works are presented.

Chapter 2

Survey of Agent Techniques

It is an elaborate work to speculate, plan, and construct a system with Agents. Before we study guidelines on how to build the system, we will first investigate what tools are available for developers to adopt in the Agent software process from analysis and design to implementation.

2.1 Terminology of intelligent agent

An Agent is a software entity that is not only designated to run routine tasks commanded by its users, but it also has to be committed in reaching its purposed setting in the computing environment. The difference between an Agent and a software entity is that the latter just follows its designed functions, procedures, or macros to run deterministic codes and derives finite results while the former has the ability to practice intelligence by making decisions based on dynamic runtime situations.

There is no specific definition of an Agent. According to N. R. Jennings [1], autonomy is the central notation to Agent. In a more specific outlook on Agent's characterization, an Agent is an encapsulated system, which is situated in some environment. An Agent is capable of flexible, autonomous action in the environment in order to meet its designed objectives. Some essential concepts that an Agent has will be discussed in the following.

An Agent is a software system that is objective-oriented. These objectives often are in terms of goals. Depending on application, an Agent may be associated with one or several goals. In order to meet goals, there will be some guidelines for an Agent as a computing entity to follow. Guidelines are called a "plan" in an Agent system. Goals should always have corresponding plans to be accomplished.

- Goal – Goal is the purpose why an Agent exists.
- Plan – Plan is the guideline for an Agent to accomplish goals.

An Agent can perceive its surroundings. In the computing environment, the surroundings could be resources, run-time conditions, or other Agents. An Agent should be able to sense information from its running environment. Based on the information evaluated in a suitable condition before either program codes execute or after, it can select a plan. An Agent should also realize its counterparts in the reachable environment. Information exchange or service providing could let Agents contribute to each other.

- Environment – computing atmosphere in which an Agent lives.
- Pre-condition – runtime situation before a plan has been executed.
- Post-condition – runtime situation after a plan has been executed.
- Communication – Agent to Agent interaction in the environment.

An Agent can run tasks in real-time systems. In order to be autonomous, an Agent reacts events in its environment. Additionally, it should be able to gather information and take actions based on goals without any invocation from the outside.

- Reactive – An Agent can reply to its environment.
- Proactive – An Agent can act properly in a timely manner.

An Agent is a self-contained software entity. Upon being activated, an Agent should be sufficiently flexible to apply suitable rules based on real-time situations. Determining decisions for Agents depend on reasoning ability. An Agent will have to evaluate the next best step to take with the environmental parameters. Even during the period with no external connection, an Agent should be able to act on its own initiative in order to complete goals. Whenever any external request is received, an Agent should be able to act using its reasoning. It should be able to decide freely with acceptance or refusal. If it is accepted, it still depends on the Agent to decide what, how, and when to act.

- Reasoning – An Agent acts based on calculating logically toward goals

As the system gets larger and more complex with many desired purposes coordinated together, it is natural for several Agents to appear in the system and cooperate on their tasks. The terminology "Multi-Agent System" (MAS) refers such systems. From the computing system's point of view, MAS should be a concurrent, distributed, multi-threaded system. Each agent in MAS has its own state with respect to the runtime situation. Complexity will increase because it is not an easy task to describe and model the system. To have a life-cycle process to develop MAS, methodologies of how to model it must be proposed. Agent-Oriented Software Engineering (AOSE) is that technique which develops processes concerning how to build MAS with practical applicable steps.

2.2 Techniques using Object-Oriented Development

2.2.1 Object for Agent

Software engineers often use Object-oriented programming to implement Agent systems. There are some similarities between an Object-oriented (OO) entity and an Agent-oriented entity. They are distributed instances that can communicate with peer entities. They both have state and member functions to support their behavior.

Practically, while using Object-oriented software engineering (OOSE) methods to develop Agents, well-tested techniques make the Agent software process more adaptable to a majority of systems. Systems thus could be utilized more easily and integrated into Agent embedding. However, some deficiencies exist between Agents and Objects. For example, Agents as autonomous entities should be self-contained. They should act to do something without external influence or initiation. In other words, an Agent could decide when and how to execute its functions. On the contrary, Objects are passive entities. They can only run their behavior as their member functions are triggered. Besides, Objects lack explicit mental state concept; they do not have mechanisms to act proactively toward goals.

Lacking major features of an Agent system while using pure Object-oriented paradigms, software engineers usually work with their experiences and intuition in modeling and implementing Agents. However, Object techniques like encapsulation, inheritance, and class hierarchy add practicality and maturity to programming. One solution to build an Agent system is to build an infrastructure of Agent semantics using Object structures. Some Object-oriented frameworks for Agents have been proposed to overcome the inherent gap in constructing conceptual models and implementing of an Agent. Properties such as mental states and autonomy are candidates to be solved by new frameworks. Here we look at a recent endeavor, named MAS Framework, as an example.

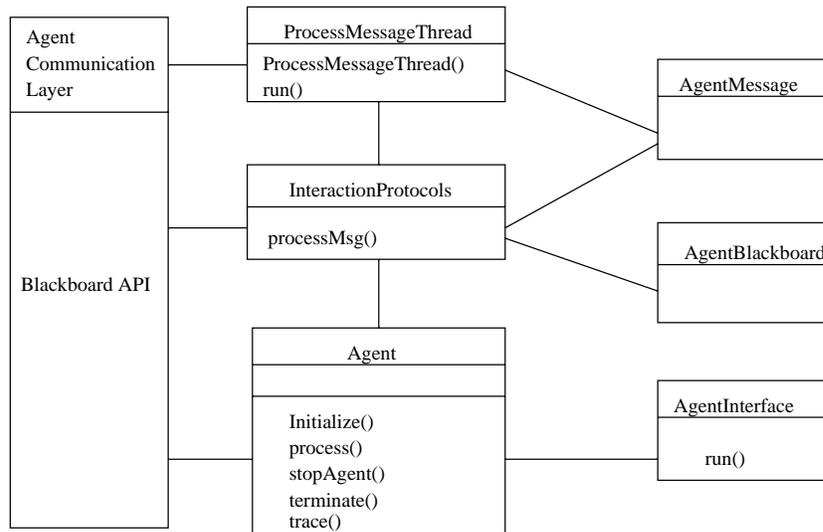


Figure 2.1: MAS Framework Model

MAS Framework is presented by J. Sardinha et al [18]. It is composed of an abstract class called Agent, two final classes called ProcessMessageThread and AgentCommunicationLayer, and four interfaces called AgentMessage, AgentBlackboardInfo, InteractionProtocol, and AgentInterface. A model diagram is depicted in Figure 2.1. Two essential issues are handled here: one is autonomy and the other is communication. In the Agent abstract class, prototype members deal with the Agent autonomy and are modeled. Methods process() and stopAgent() are used to start and stop the Agent. Method process() will use AgentInterface to implement a thread, which has a method named run() to operate something that the Agent could issue and handle without any interaction from other Agents. The Abstract class also provides Initialize(), Terminate() and Trace() methods, which are responsible respectively for initialize, terminate, and trace activities of Agents' autonomous execution. Developers will need to complete the codes in these three methods to apply the framework.

Another feature in the model is to deal with Interaction using asynchronous threads and Blackboard mechanisms. InteractionProtocols interface is to define the way Agents interact with other Agents. Thread ProcessMessageThread will always be created for

every incoming message so that the thread InteractionProtocols can coordinate response flexibly with activities of the Agent. Additionally, AgentCommunicationLayer is a final class that implements the entire communication infrastructure based on a Blackboard ready system, such as the IBM Tspaces.

Developers using this framework are to instantiate the Agent by inheriting from the core Agent class, which is extended by adding special ability to the class. Some member functions for the Agent need to be filled for autonomy in each Agent. Also, Agent interactive protocols are needed to implement the InteractionProtocols interface. Then, the newly instantiated class has embedded autonomy and communication features from the framework.

2.2.2 Object Oriented Programming Patterns

Pattern provides a template for a solution to a recurring problem. In Object-oriented modeling, design patterns have been mostly used to increase flexible, reusable, and effective software entities. Practitioners can apply patterns to fit parts into applications thus gaining clear views of design and comprehensive detailed scenarios within the pattern. In the view of design and implementation, an Agent not only preserves Object properties but is also an extended Object with specific capabilities. These features should be considered while contemplating Object-oriented patterns for Agents.

There are many significant patterns introduced and collected by Gamma et al [32]. Some commonly used patterns in Object-oriented design with similar Agent features can be used to apply Agent scenarios easily. For example, both Objects and Agent classes can use the idea of coordination. Proposed by S. Hayden et al [21], the Mediator pattern design for an Agent enables the encapsulation of Agent interactions. Mediator facilitates complex interactions between each peer by coordinating as an arbitrator between them. As the hub of a communication range, Mediator takes control and coordinates interaction among Agents or groups of them. This mechanism reduces the number of interconnections between Agents and the complexity of processing details with each entity's behavior. A structure of this Mediator Interaction is depicted in Figure 2.2.

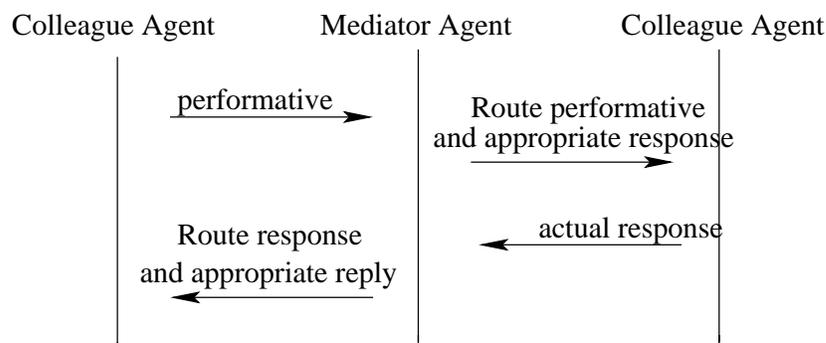


Figure 2.2: Interaction Diagram of Mediator Pattern

The Object-oriented Mediator pattern in the UML diagram, portrayed in Figure 2.3, applies to the example of an Agent property. The Mediator design patterns can be used to handle the composition of various Agent properties. Agency properties are encapsulated as classes, while they refer to each other only through the mediator. This could facilitate the maintenance of Agent properties.

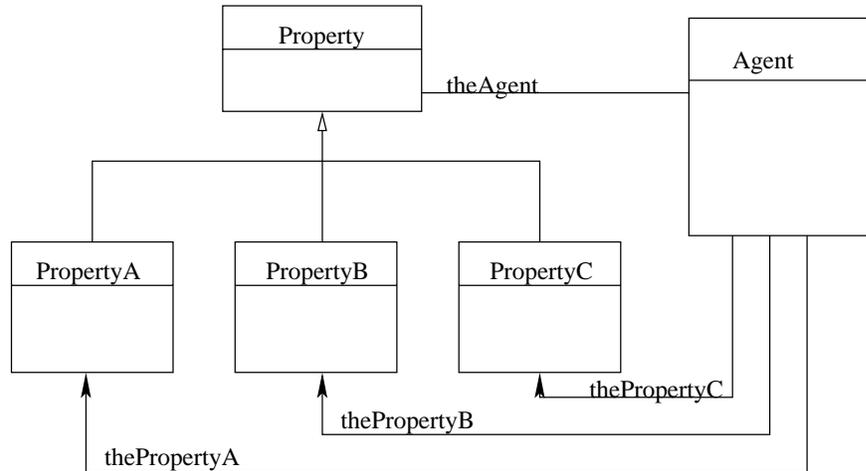


Figure 2.3: UML Diagram of Mediator Pattern applied to Agent Property Model

Evolved from OO paradigms, patterns designed for an Agent can be assembled by special needs in managing the aspects of Agents. In Agent paradigms, role concepts are used to model an Agent's common behavior. An Agent often plays different roles in different situations. To use class technique of object to handle various roles, the Agent class design could either add new role member functions to the Agent class or sub-class from different characteristics of roles. The former method could lead to interface bloating while the latter could face problems of confusion in Object identity. Both methods would lead to complicated considerations of what is the proper arrangement of classes and how to manage them. To simplify the work of frequently used Agent concepts in OOD, a role Object pattern has been proposed by D. Baumer et al [22]. Role Object patterns model roles within specific contexts as separate Objects, which could dynamically attach or remove themselves from the core Agent object. A Role Object pattern structure is

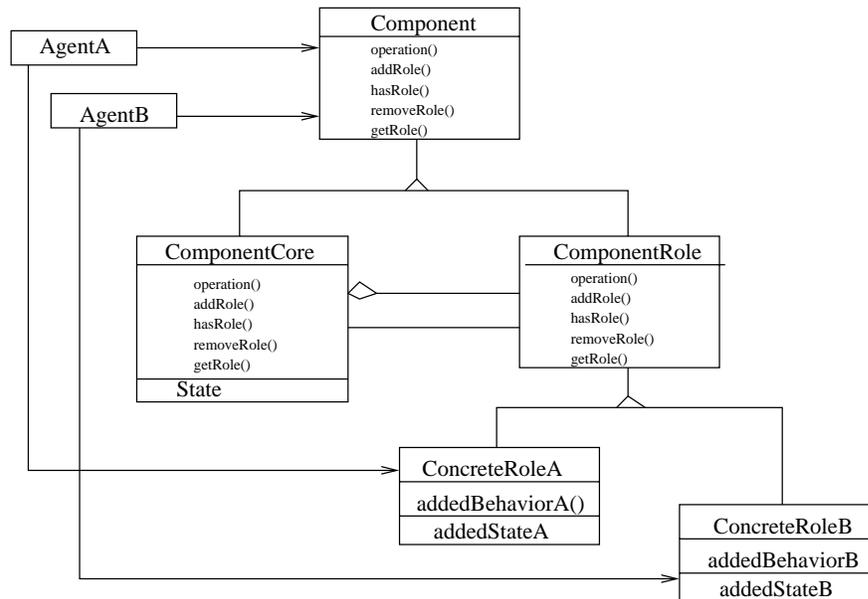


Figure 2.4: Structure of Role Object Pattern

depicted in Figure 2.4. In the diagram, a component is defined as an abstract superclass. The ComponentCore class implements the component interface. A common superclass of different roles is provided by the ComponentRole, which also implements interface to the component. ConcreteRole classes are to be instantiated at runtime. In each role specific subclass, additional properties and activities are defined specifically. For a component within an application, the only needed roles are loaded to be manipulated.

In the design of patterns to be used in an Agent, most of them could be inspired by common OO patterns. In the Object-oriented design environment, Objects and Agent classes usually coexist. A type of behavior could be perceived as an unit object in the system. Patterns, typified by the preceding as an excellent integrated unit, could be deployed under Agent conceptual modeling.

2.2.3 Object-Oriented Programming tools

While implementing using Object diagrams, programmers can adopt widely used Object-oriented languages such as C++ and Java. Java, especially, is the most popular programming tool in dealing with Agents. Because of its platform independence, it is much more convenient to deal with porting, while the features of Agents are most likely to be distributed and mobile. As developers run projects of multi-Agent systems using OOP, the benefits of using OO tools, such as UML modeling and design patterns, can be inherited by using Object-oriented programming.

Object techniques feature encapsulation, inheritance, and polymorphism. These class properties can be treated as concerns in Object-Oriented Programming. However, while constructing multi-Agent systems, the principle concerns are not just class attributes or member functions. There could be Agent states, task plans that are critical to Agent programming. The influence is substantial if these Agent proceedings are taken into consideration while building the system with Object-oriented programming.

Aspect-oriented programming (AOP) [33] has been proposed for improving separation of concerns (SOC) in software development. The concept of AOP is to provide a more understandable, visualized, and manageable programming, which confronts specific concerns in the program. A crosscutting concern is usually called an "Aspect". It is not an ideal situation for programmers to handle Aspects scattered in a large project. It could cost too much for developers to revise, debug, and maintain each of the details. Rather, AOP supervises all these concerns automatically. AOP does not replace OOP but provides a manageable way to separate Aspects from components.

The basic element of AOP is to define join points. A join point is a chosen program point located on the track of a program running flow. A pointcut is a collection of join points. Programmers usually define specific pointcuts based upon project's need. For instance, a programmer is concerned about program outputs by using print commands.

The pointcut can thus be defined as the collection of every join point wherever the program may call to print. This mechanism provides a service for programmers in that they need not go through the whole program to seek code line by line. Aspect programming will handle it automatically.

A powerful function of AOP is that it can affect implementation at designated join points. When a program encounters a join point defined in any pointcut, it is called crosscutting. As it happens, an additional code is provided to run at the join point. The additional code is called "advice". The mechanism that both join points encountered in which advice is provided is called "weaving". When an Aspect is woven into a class, it could either introduce behavior with new methods or add behavior that is already in the class. Advice weaves can be designed to run before or after the method. In this way, passive objects can be transformed into active ones. It is one of the approaches to make Object-Oriented Programming more like Agent-Oriented.

There are many popular OOP languages that now have extensions to Aspects. In this paragraph, we will take a closer look at an extension to Java called AspectJ [33]. AspectJ extends Java to two types of crosscutting implementation. One is dynamic crosscutting, which defines additional implementation to run while the program meets at certain join points. The other is static crosscutting, which defines new operation on existing types as join points meet. This could even change class hierarchy without revising related classes. This phenomenon is called inter-type declaration. It provides a mechanism for aspects in AspectJ to reference members in ordinary object class without inherent limitations from inheritance. Aspects are implemented using object class to host crosscutting concerns. An Aspect class can be composed of different concerns. In the class, some pointcuts and related advices are defined. Figure 2.5 shows the structure outline.

An example, published by E. A. Kendall [23], is to implement Agents in role modeling using AspectJ. A role is a high-level concept to describe responsibility for an Agent. Sometimes, an Agent needs to play a role that is different from other Agents in order

```
Aspect MyAspectClass{  
  
    pointcut MyPointCutA() : <<join points definition>>  
    pointcut MyPointCutB() : <<join points definition>>  
  
    before MyPointCutA() : { << MyAdvice program >> }  
    after MyPointCutB() : { << MyAdvice program >> }  
  
}
```

Figure 2.5: AspectJ Programming Structure of Aspect Class

to collaborate. From time to time, an Agent could change its role to that of something different in order to support its tasks in contrasted relationships. Roles for Agents can be treated as crosscutting concerns. In order to handle different roles, different aspects are provided in the class dynamically with weaving mechanisms.

2.3 Techniques using Agent-Oriented Development

2.3.1 Theory of modeling

A software process consists of steps that developers can adapt as road maps to follow in building desired systems. Within these steps, some models and techniques have to be offered for developers to handle complex problems easily. These tools and approaches have to be stated clearly and proved successful so that practitioners can procure desired specifications in constructing the system.

In the context of building multi-Agent systems, Agent-oriented approaches have to deal not only with traditional issues but additional complexity of states among multiple Agents. As Booch suggested for Object-Oriented approach, decomposition, abstraction, and organization are principles to handle software complexity. N. R. Jennings inspects these mechanisms to prove that Agent approaches can also use these principles for Agent construction [1]. However, essential issues presented by N. R. Jennings concern that an Agent's interaction and behavior are unpredictable. To solve the problem, he suggests following approach, inspired by a social level model.

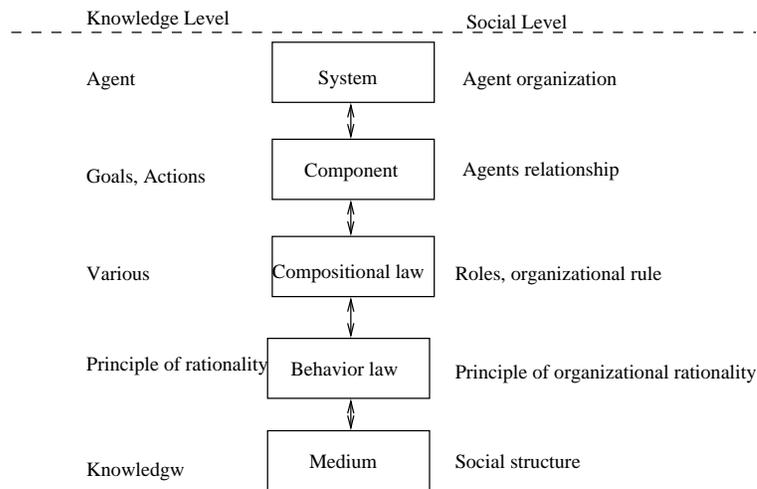


Figure 2.6: Social Level Model

There are two ways to describe the social model. The first is the social-level viewpoint and the second is knowledge-level viewpoint. Figure 2.6 shows the social level model in a summary diagram. The social level describes a system to be modeled as an organization or society. The system is composed of components. The majority constituents of components are Agents. Their communication channels include content and mechanisms, dependencies between agents, and organizational relationships between Agents such as concepts of peers and competitors. In the society, compositional laws are made as guidelines for how components in the system are organized under the regulation of the society. In plain text, every component plays a role or roles in the society under the organizational rules. Rules regulate which component can adopt which roles. A role can be taken by more than one Agent and the implicit concept is to achieve its objectives. Functionality of roles could be used to regulate organizational relationships, communication channels, and interaction patterns. Different matches between components and their adherent rules results in different component activities. Behavioral law regulates how components act in terms of a resident in the organization to meet both their roles and social commitment. The lowest level of the model is the medium. From the social level viewpoint, units of the system are different organizations in the society. Different organizational mechanisms and structures can influence the behavior of components residing in it. The way organizational structures change can also have effects on role relationships, especially adding new roles or removing them according to the adjustment. From the knowledge level viewpoints, Agents are central to a system. An Agent perceives its goals and accomplishes them by actions. These goals and actions have rules of rationale, which are provided as laws in the level mode. All laws are based on the knowledge of their environment. Based on the knowledge they have, Agents are working toward their goals.

Theoretically, the social level conceptual model stands on top of the knowledge level and provides social concepts as a foundation for Agent-oriented systems. It could facili-

tate a high level system constructed without going into the details of greater complexity. Moreover, law in the society helps to regulate an Agents' behavior and makes the results of the system more predictable.

Organizational concept models is the most widely used fundamental philosophy that software developers use to devise Agent-oriented methodologies. N.R. Jennings [1] also proposes a prototype that fits the conceptual model into an analysis and design. In analysis, the main task is to identify overall goals of computing organizations, basic skills, interactions between organizations, and rules of behavior these organizations follow. The identification results will be roles, protocols, and rules. In design, readjustment roles, organizational patterns, and the definition of organizational structure are exploited. From the prototype, the central metaphor generated from organizational concepts can be roles in the society. Thus, role modeling becomes a subsystem in the system architecture. Goals, implicit behind roles, and Agents are important elements to analysis. In social concepts, goals are mostly mentioned as organizational aspects. Goal structure analysis can help refine them in the organizational structure. This is only one of two dimensions. Applying the concept to methodology design, if goals can be properly modeled with roles, it would cover most conceptual scenarios in the organization and provide clear semantics in the software development process.

2.3.2 BDI architecture

A crucial capability for autonomous Agents is reasoning. To model mechanisms of how Agents think can help designers observe micro-activities in Agents. A popular, well-known method to describe how Agents behave in rationale philosophy is the BDI model, proposed by A. Rao and M. Georgeff [4]. The motivation and foundation of the modeling is from recognizing practical limitations and requirements of environments within the application domain. To model the behavior of an Agent, both dynamic factors from system and environment should be considered. An Agent BDI model stands for

”Belief, Desire, and Intention”, which are system and environment parameters, goals, and plans respectively. Based on ”belief”, an Agent has goals delegated and can have corresponding plans executed to achieve the goals. Agents could reason about what is the best plan for a specific belief about the environment. As system or environment parameters change, an Agent should review its goals and respond with different plans, if necessary, to accomplish them.

Although it is useful, someone may argue that the BDI model could have limitations to Agent-oriented architecture design. As more Agents require learning capability, BDI models seem to be unable to adapt their own behavior to consider the next move. It has no explicit expression of Agent behavior. However, it is still a widely relied method to successfully model the various mental states of Agents.

2.3.3 Agent UML

Unified Modeling Language is widely used in a Object-oriented paradigm. Inherited from the object nature, limitations exist for it to model multi-Agent systems. However, UML has been a successful modeling language in Object-Oriented design. It would be a benefit if Agent oriented design could also use UML by extending itself to adapt agency features. Software engineers could easily adapt Agent concepts into their work and benefit from reengineering legacy systems into Agent systems.

To inherit benefit from Object-Oriented methods, alternative methods have been studied. One way of doing it is to build models of different Agent properties as an infrastructure, which has been discussed in section 2.2.1. The others could extend UML by adding Agent features to make it capable of addressing issues in Agent-Oriented paradigm.

For instance, the Agent UML has evolved from the extensions of UML to an Agent capable modeling language. The underlying motivation of extending UML is to accommodate required features of Agents that are lacking in UML. Many proposals have been

suggested by different research groups. Currently, both FIPA (Foundation of Intelligent Physical Agents) and OMG (Object Management Group) are securing proposals to extend UML to accommodate Agent features. There will be an official standard for AUML in the future.

As proposed example, a three-layered approach to present Agents' interactions is introduced by Odell et al. [16]. The rationale behind it is the view from a whole protocol package down to the details concerning interactions among Agents and intra-Agent behaviors. The way of viewing it is from macro to micro. A table in Figure 2.7 shows the summary for the layers and the modeled techniques corresponding to them.

Layer Category	Models applied
Level1(overall protocol)	Packages, Templates
Level2(agent interaction)	Sequence Diagram, Collaboration Diagram, Activity Diagram, Statecharts
Level3(internal agent proc)	Activity Diagram, Statecharts,

Figure 2.7: AUML package model

The first layer emphasizes using patterns that facilitate interaction protocols for Agents. Agent interactive protocols can often be reused in various kinds of Agent communications. The package diagram used here presents a whole protocol pattern with the tag name of the package representing the meaning of the interaction. Figure 2.8 displays the package model for protocol. The template attached at the upper right corner is to establish parameters and interaction sequences of the protocol involved in the package diagram. The mechanism of the template is good for users to apply with other sets of Agent communications that could reuse the pattern.

The second layer focuses on interactions among Agents. As we looked into the package of the first layer, interactions are denoted as a sequence diagram. However, the pure

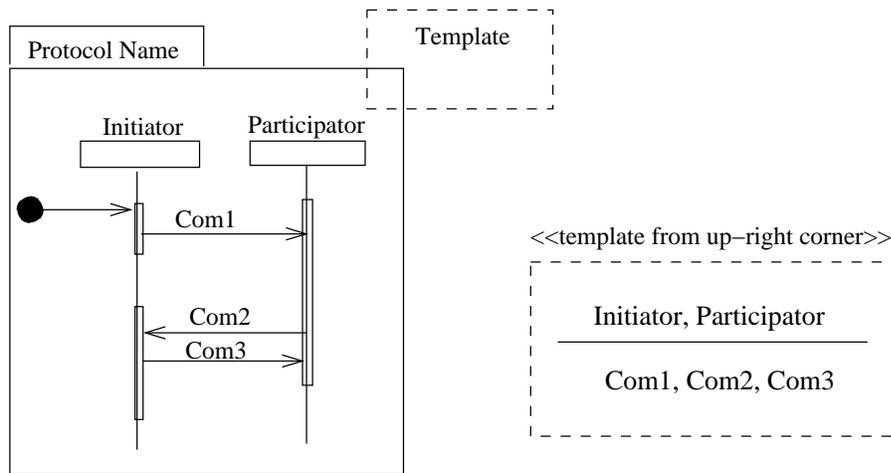


Figure 2.8: AUML package and template model

objects sequence diagram does not fully cover all the scenarios of concurrent thread use. Expressions of three basic multi-threaded extensions are introduced. Figure 2.9 portrays different sequence types. From the left of the diagram, it indicates that all the threads from Th-1 to Th-n, total of n threads, interact concurrently. The one in the middle says that zero or more threads interact sent. The righthand picture shows an exclusive-or decision node, which means that only one of n threads will interact.

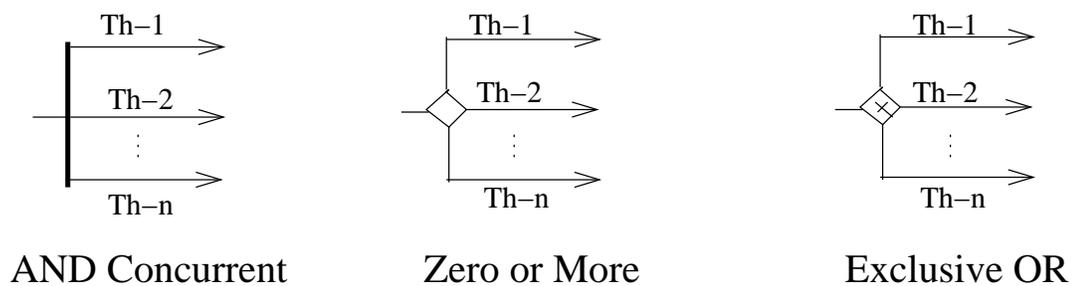


Figure 2.9: extensions to support concurrent multithread model

Meanwhile, Agent interaction sequence diagrams are transformed into partnership diagrams for getting clear views of the system. To describe interactions more specifically, activity diagrams are introduced to express a summary of operations and events that

every role must perform. Another diagram used in the layer is the state chart. As used in UML, the internal states of interactions between Agents can be modeled as a state machine with transition labels between state moves.

The last layer represents internal Agent processing. For each Agent running in the system involving interaction protocol, it must record its internal status to know how and what to do to interact. The activity diagram here is used to model detailed processing of each Agent's response to events of specific interactive purposes. Also, state charts present more details about internal states in the interactions of an Agent.

2.3.4 Patterns for agents

A design pattern is a technique to facilitate software development by identifying complex aspects and making them into a module of software architecture. Generally, a reusable and ready-to-use template for the software entity is a great time-saver in software design. Inspired from patterns using Object-oriented techniques, some researchers have been adapting them to multi-Agent systems. Categories of patterns, as suggested by E. Gamma et al [32], are creational, structural, and behavioral. According to different abstractions modeled in an agency paradigm such as roles, systems, societies, architectures, and interactions, tasks and environments can be patterned categories for multi-Agent systems. Moreover, different application domains can be applied using specialized patterns. For example, in a mobile Agent context, three classes of patterns, including task, traveling, and interaction are introduced by Y. Aridor and D. B. Lange [30]. As an example of traveling, the itinerary pattern is concerned with routing among multiple destinations. It maintains a list of locations and routing schemes to handle the "Where to go next" questions.

While an Agent as a software entity has its specialty in constituents, agency patterns could be building blocks of these properties. E. Kendall et al have suggested a layered

pattern language [24]. The contexts for the model patterns are first defined as Agents are autonomous, social, reactive, and proactive. The context mainly is to model an Agent's behavior. In the behavioral model, there are sensors and actuators as input and output interfaces for an Agent. There is an interpreter to handle collaboration and multi-disciplinary problems. Additionally, a migration module is to handle migration and load balancing through the network. With the preliminary domain fixed, a seven-layered Agent pattern has been proposed. Figure 2.10 shows the layered Agent pattern and each layer's corresponding responsibility. Patterns from higher layers depend on lower layers to provide services. Patterns in lower layers provide responses to its upper layered patterns. The three lowest layers are responsible for handling the mental states of Agents. The higher level patterns consist of more constituencies of Agent behavior.

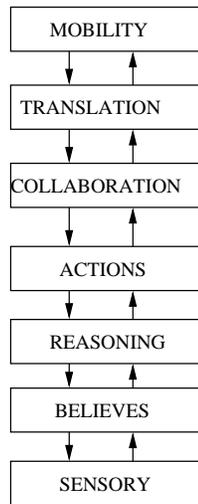


Figure 2.10: Seven Layered Agent Pattern

Observing from examples above, there are still constraints in applying patterns to Agent modeling. Because of the Agents' nature, either the application domain or the Agents themselves have to be identified before patterns can be exploited. The modeling plays a pattern building role. While taking advantage of pattern language, there are common elements of Agents that need to be discussed before applying it.

2.3.5 Agent-based Platform

In addition to constructing an Agent-based system from scratch using prominent tools one-by-one, there are various Agent development frameworks that have been established as Agent platforms. These platforms provide an infrastructure for developers to concentrate on specific Agent attributes or write member function codes as needed. Examples would include Agent class design, Agent communication library, and Agent platforms. Since it is the interface between the system design and implementation, the infrastructure is also called the Agent middle-ware. In the following, we will give an overview on some of the featured platforms, which realize how Agent-oriented engineering can take advantages of these.

Zeus [34]

Zeus was developed at the British Telecom Laboratories. It is built as a collection of Java classes and categorized into three components. The first one is the Agent component library, which is the element for generating an Agent in the system. Agent components provide tools of knowledge manipulation, communication, coordination protocol, and a planning and scheduling system. The second is an Agent building software, which provides sets of graphical user interfaces to guide users in procedures necessary for the system to function. The third is the Agent's society visualization tools, which provide utilities used at runtime to monitor and manage the behavior of Agents in the resulting system.

As developers begin their work, it is recommended to choose using role modeling as the analysis approach. Four stages to follow are defined in the design process. As the Agent class is central to the toolkit, the first step in considering multi-Agent systems is deciding on how many and what kinds of Agent roles will operate in the system. This phase is called "Agent configuration." With conceptual role modeling, the responsibilities of these roles are defined and mechanisms of cooperation between Agents are

designed. Then, Agent building software can be applied to complete these Agents' settings. This phase is called "task definition". Zeus supports role modeling as well as social context. Besides the basic settings for rules of Agents, Agent acquaintances and relationships need to be defined. This phase is called "Agent organization." The last stage is "Agent coordination," in which interactive protocols are defined. A library of predefined protocols is provided within the toolkit.

Finally, Agent production tools can be used to generate Java codes automatically for the design. Developers can supply each Agent with code via implementing their associated special abilities. These modules are linked through well-defined interfaces in Zeus. Agent visualization tools are used to monitor runtime systems and debug the design.

JADE [35]

JADE, which stands for Java Agent Development framework, has been developed to provide Java-based Agent platforms that allows developers build in Agent applications complying with FIPA specifications of multi-Agent systems. JADE is an open source Agent-based project developed at Telecom Italia Laboratories. An Agent platform is the principal software architecture of JADE. It provides an environment in which Agents live and work. To provide services of the platform, three modules are basic to the Agent platform architecture. Agent Management System (AMS), Directory Facilitator (DF), and Agent Communication Channel (ACC). All these are activated during platform startup. Figure 2.11 shows a model of an Agent platform.

Inside the platform, the framework is based on the coexistence of Java Virtual Machines (JVM) and Remote Method Invocation (RMI) for a communication infrastructure. Each virtual machine is a basic container in which Agents run. A container consists of several Agent threads that can run concurrently and a system thread responsible for dispatching messages. An Agent platform is composed of several containers that are

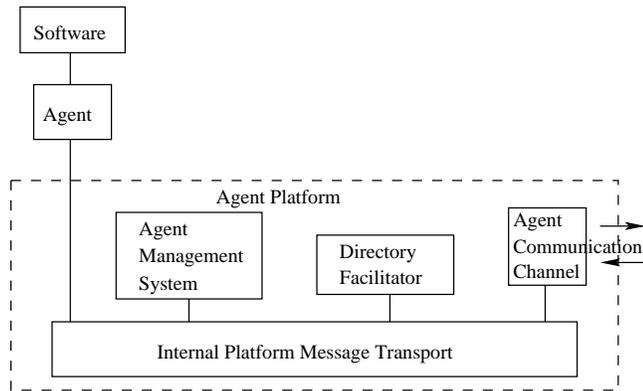


Figure 2.11: JADE Agent Platform

distributed throughout the network on different machines, which provide RMI communications between hosts. Figure 2.12 shows an intra-platform communication between Agents in different containers.

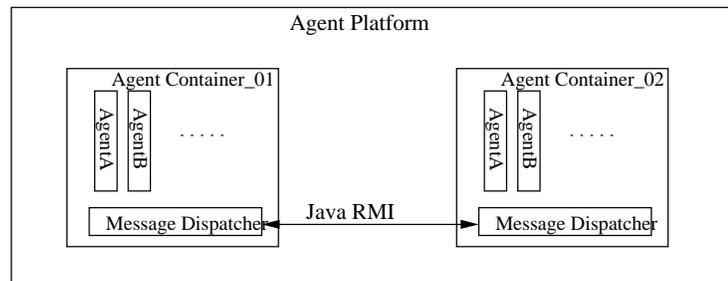


Figure 2.12: JADE Agent Platform Architecture

In JADE, each Agent is implemented as a thread. An Agent abstract class is the core class to implement an Agent. JADE also uses behavioral concepts to model tasks that an Agent is going to perform. Developers should implement special behavior of Agents by joining Agent classes with behavioral classes. Specific codes of action are implemented in the behavioral classes and are finally added to Agent classes. The class hierarchy design has benefit of using inheritance mechanisms to manage Agent capabilities.

JACK [36]

The JACK Intelligent Agents framework is created by the Agent Oriented Software Group. JACK adapts architecturally independent facilities that component plug-in mechanisms use to specify Agent architecture. One example is the plug-in of BDI architecture as its default model for Agent reasoning. Meanwhile, three extensions are made to Java in the project. First is a set of syntactical additions to describe Agent mechanisms. For example, Agent and plan and event are keywords to identify an Agent and its related features. Also, some statements are added to the interface to declare attributes and relationships of Agents. Statements for manipulating Agent states are provided. These additions are comparable to the Java language. Second, because of the extensions, compiler is extended to support these statements. Finally, some kernel classes are provided for runtime support to the generated codes. These include automatic management of concurrent tasks, default behavior of Agent reaction, and communication.

Another extension that JACK supports is called "Team Oriented Programming" or SimpleTeam. An Agent-oriented paradigm supports Agent collaboration. The mechanisms map from a high level abstraction to a low-level specification in Agent activity. SimpleTeam is to define the structure of a team and how it functions. Furthermore, statements to describe team behavior are added to the Java language. This can facilitate applying the framework in designs using social organizational concepts.

Developers using JACK can add or change system architecture by plugging in their own components. After establishing the basic Agent prototype, elementary Agent classes are identified. These basic classes will provide primary capabilities that Agents can execute in tasks running the environment. With the prototype of Agent classes, mental states of an Agent must be specified. For example, the goals an Agent can predetermine for itself or tasks executed while reacting to an environment are those acquired to ac-

compish objectives. Then developers should convert the above identities into codes by using sets of statements from Java.

Chapter 3

Agent-Oriented Software

Engineering Methodologies

Agent-Oriented Software Engineering(AOSE) is an active research field and is still on its way to a mature industry-strength solution [31]. A methodology that plays an essential role in the engineering is required to be robust and easy-to-use. More importantly, it should provide a roadmap to guide engineers in creating an agent-based system. There is an increasing number of AOSE methodologies that try to encompass the software issues and compete in being the main approach. In this chapter, we study major representative agent-oriented methodologies, from Tropos, Gaia, MaSE, and Extending UML, to a method using pure object-oriented techniques, which engineer multi-agent systems. For each introduction, we outline theoretical foundations as well as the essentials of each method. During the discussion, some comments will be annotated.

3.1 Tropos

Tropos [7] has been mainly developed by a team with key members in the University of Toronto, Canada, and affiliated with other members from various Universities in Europe.

It adopts E. Yu's i* framework [5] ¹ as the base theory of requirement analysis. i* offers concepts such as actors, goals, and dependencies intended to model social structures and to note detailed relationships between them. Rooted in theory, Tropos provides a method for engineers to design multi-agent systems that can take advantage of this technique throughout the design process [7].

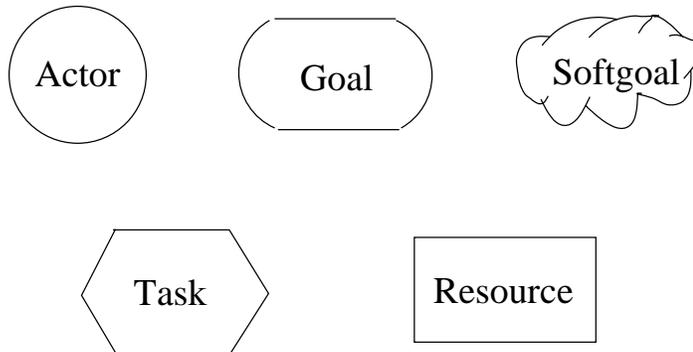


Figure 3.1: Notations used in Tropos Methodology

There are four phases in Tropos during analysis and design. The early requirements analysis phase is the first step for designers in identifying basic stakeholders. Realizing the realm of the project, designers can discern the kind of stakeholders the system is going to have. The initial identified stakeholders are presented as original actors. Each of them has a high-level goal attached to represent its intentions. In Tropos, goals are categorized by functional and non-functional properties. Softgoals, known as non-functional, are defined as those which have no clear-cut definition or criteria as a measure to whether they satisfy or not. Softgoal notation is useful in the easy expression of non-functional semantic meanings, depicted in the analysis diagram. Figure 3.1 shows notations used in Tropos for the modeling.

Identified actors will form a society, ask for services, and take care of each others' needs. Like a human society, dependency plays an important role for actors to get re-

¹More information of i* can be found at <http://www.cs.toronto.edu/km/istar/>

sources they need in order to fulfill their goals. Figure 3.2 depicts dependency between actors. The relationship is to be interpreted as a "depender," for it has to depend on the dependee through the dependum. Developers should begin to explore dependent relationships between them. There are two kinds of modeling tools used in Tropos for dependency analysis: the strategic dependency (SD) model is focused on developing networks of relationships among actors, while the strategic rationale (SR) model describes reasoning relationships of one actor among the others.

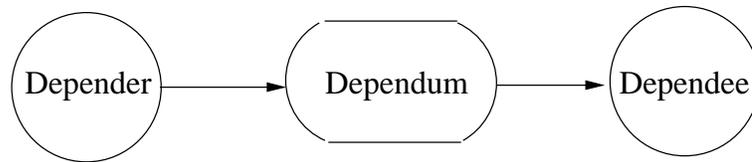


Figure 3.2: Relationship of Actor Dependency used in SD Modeling

In the SD model, a suitable dependee can be found to provide a dependum needed of a depender. The dependum could be goals, softgoals, tasks, or resources. In addition, the dependency type is named after the dependum. For example, goal dependency is the depender finding a dependee to fulfill its goal as a dependum. Likewise, task dependency is the depender needing the dependee to provide activities to serve its needs. Developers outline the relationships among actors by using the SD model in order to realize possible network in the setting.

In the SR model, two types of links are available including means-ends links and decomposition links. A means-ends link is a reasoning tool that decides how to achieve the end of the objective while using available means. From a goal statement, developers can postulate what kinds of means to execute. Then the goal can be decomposed into a refined goal or softgoal. On the other hand, decomposition links take place to decompose goals or tasks into sub-goals or sub-tasks, in which developers assure, at least theoretically, this decomposed entity will finally complete the original goal or task.

By using SR modeling, developers can refine goals and tasks to observe a more detailed analysis. Figure 3.3 shows the SR model diagram.

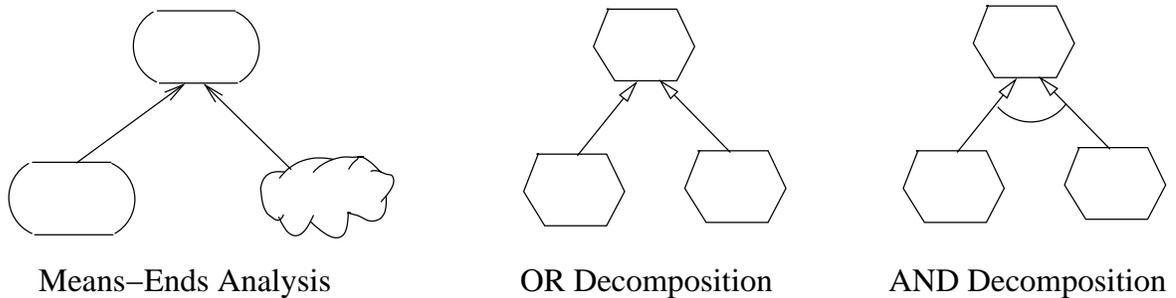


Figure 3.3: Means-Ends Analysis and Decompositions used in SR modeling

The main purpose of the decompositions listed above is to help engineers refine elements in the actors to explore more detailed relationships among them through cooperation. By sharing sub-goals, sub-tasks, and resources, actors can discover what they need and where to ask for service. Engineers need to apply these analytical models iteratively investigating additional refinements and dependencies thoroughly to finish the early requirement phase.

Entering the late requirement analysis phase, a future system actor is introduced. The purpose of this "system" actor is to provide system operational services to actors depending on services from the last analysis phase. Some system functions relate to goals and tasks can be delegated to the system actor, who should manage the system services. The newly introduced system actor needs to be applied to the same analysis as in last phase and dependencies should be established with other existing actors. Thus, the dependency analysis is inspected for adjustment with the new refined scenario.

After all the dependent analyses on actors are performed, it becomes complex for designers to recognize them clearly. The first design phase, architecture design, attempts to help recognize common social dependencies as patterns. In the design phase, it

defines the system's architecture in terms of sub-systems and dependent relationships of resources and functionality. It also describes how system components work together. Social style patterns have been acquired to let developers use them [6]. These social patterns [8] help developers recognize capabilities associated with actors involved in the pattern. In Tropos, we find the capability of an actor by observing the dependent relationships it has.

Each dependency can be interpreted as a capability triggered by the events through dependum. Thus a list of capabilities collected from the actors can be ascertained. Designers can refer to the list identifying different capabilities and classify them. Then a set of agent types is decided according to the arrangement of capabilities.

In a detailed design phase, more explicit scenarios of agents are depicted. The behavior and communications of actors based on their agent types are explored using extensions to UML and AUML. In this phase, Agent class diagrams, capability diagrams, plan diagrams, and agent interaction diagrams are used.

While in the implementation phase, Tropos adapts JACK for its execution because they both rely on BDI architecture. Notations used in Tropos can be seen as mental ones, such as goals and tasks (plans) [10]. Dependencies between actors are realized as beliefs of agents. The notations used throughout the analysis and design phase help transitions preserve the semantic mapping.

The following is a critique of the TROPOS methodology:

- The exploration of stakeholders and the dependencies between them mainly rely on the developers' experience. The realization of goals and their dependencies among stakeholders can be derived based on different points of view, which could lead to various results. The discretion is left wide open.

- During an analysis phase, when a new sub-goal is generated from goal refinement, dependencies between the new goal and every actor in the system have to be recalculated. The iterated algorithm to run this process becomes a non-deterministic concurrent algorithm [7]. It is not an easy task for engineers to conduct this analysis in an efficient way.
- Dependency analysis plays an important role in the analysis phase. Statements of goals and dependencies are prone to be unclear depending on an engineer's interpretation. Meanwhile, there is no standard rule to follow while decomposing goals or tasks into sub-goals or sub-tasks. It all depends on specific interpretations with no general guidelines.
- Although traceability is available through notation diagrams, it is still difficult to pursue all the dependencies backward. A system actor is added in the middle of the analysis. Dependencies among actors are rearranged to accommodate the new actor. There is no formal rationale to support it, which makes it more complicated.
- Tropos rests on the uniform use of small sets of intentional notations [9] throughout the whole development process; However, it is awkward to reflect on a changing environment to adaptively revise beliefs and plans.

3.2 Gaia

Gaia, proposed by M. Wooldridge et al [11], is a theoretical foundation of analysis based on the object-oriented design method called Fusion, from which it borrows terminology and notations. Gaia is rooted on the conceptual organizational modeling [13] and suggests that developers think about building agent-based systems as a process of organizational design. The agent computational organization is viewed similarly to human organization, in which it consists of different roles with several functions interacting with each other.

The main concepts of the elements used in Gaia is divided into two categories: one is abstract, and the other is concrete. Abstract entities are those that appear in the developing process but have no direct realization into the system. Concrete parts are those that appear in the process and have direct counterparts in the runtime system. Based on the logical categorization mentioned above, elements are composed into models designed for two main developing phases. The first is the analysis phase, which accommodates most abstract concepts, consists of role modeling and mode interaction. Second is the design phase, which generates concrete elements, which is composed of an agent model, service model, and an acquaintance model. Figure 3.4 notates abstract and concrete elements to depict the relationship between them.

The objective of an analysis phase is to examine the problem domain and to obtain the system structure. Based on the organizational concepts in Gaia, developers can think of a system that consists of roles for specific tasks, which accomplish system goals. A role is defined by four attributes: responsibilities, permissions, activities, and protocols. Figure 3.5 shows a role template in Gaia.

For every role in Gaia, a name will be assigned to represent the role. In the template, a short description will be provided to illustrate the role based mainly on its available

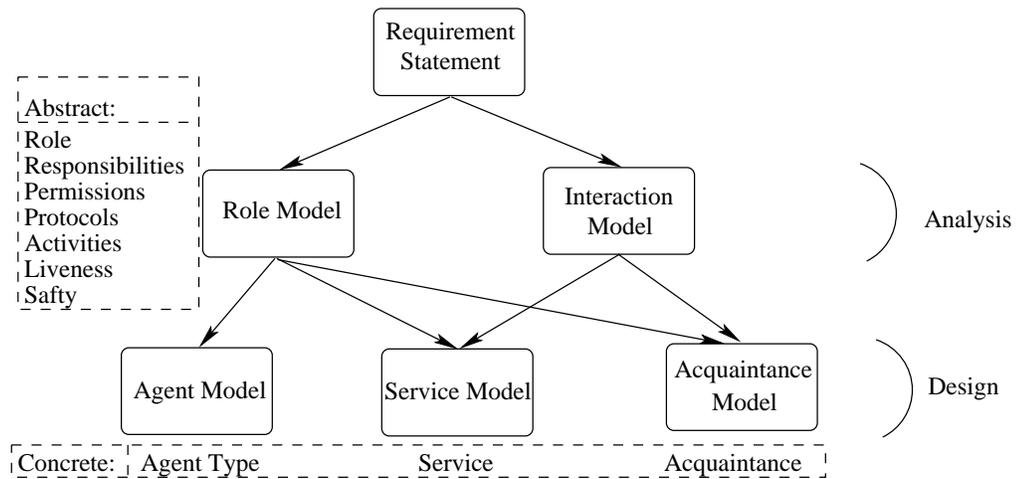


Figure 3.4: Gaia Model and its abstract/concrete conceptual elements

Role Schema:	name of role
Description	short English description of the role
Protocols and Activities	protocols and activities the role plays a part
Permissions	right associated with the role
Responsibilities	
Liveness	liveness responsibilities
Safety	safety responsibilities

Figure 3.5: Role template in Gaia

functions in the system. The building blocks of functions that a role can have are activities and protocols. Activities collect tasks that the role can perform with no external resources. Protocols are well-defined communication routines that regulate their behavior according to characteristics of a role. Figure 3.6 on the left half shows a protocol template, which consists of an initiator. The initiator is that which begins the interaction and response; it is the one to initiate contact. Entities in protocols and activities are used to express role responsibilities.

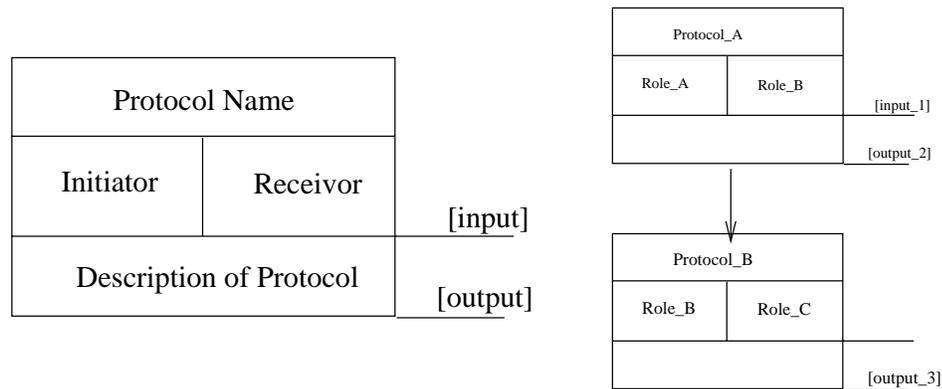


Figure 3.6: Gaia Protocol template and Interaction Model

Usually, functionality can express details using responsibility. Responsibilities are represented as obligations that roles can have and their response to fulfill them. Two types of responsibilities are categorized here: liveness properties and safety properties. Liveness properties represent something good happening, which assures that functionality is available from the role. Liveness properties are composed of entities in protocols and activities in a regularly expressed way. Safety properties state that nothing bad happens. It assures that the criterion of the functionality is monitored well by the role. It guarantees that the results in critical conditions will not be disallowed by the system. Safety property is composed of permission variables using logic expression. Permission setting is a tool to help roles guard their responsibilities. The designation of permissions is dependent on the need of roles to operate. Permissions for reading, changing, and generating variables are provided.

After all desired roles are created, interaction models are constructed. Interaction models are generated based on communication protocols available from roles. To complete a specific task, there can be several agents involved. Thus the interactions between agents can consist of a series of protocols with different pairs of agents participating. Interaction models are expressed as task oriented. Figure 3.6 on the right half shows an interaction model.

With roles and their properties at hand, the first step in the design process is to find out how roles are organized in creating agent types. An agent type is a set of one or more agent roles, and an agent model decides what kind of agent type and how many occurrences will be implemented in the system. Based on the information from the properties of roles and their interactions, developers can compose an agent type from one or more roles. This depends on the consideration of coherence and the efficiency of functionality that roles with agents are going to play. An Agent model is shown in Figure 3.7. On the left-hand side, a RoleA is selected to create AgentType1 with a qualifier +, which represents one or more instances of AgentType1. On the right-hand side, RoleB and RoleC are combined into AgentType2 with the qualifier 1..2, which indicates only one or two instances of AgentType2 will be created.

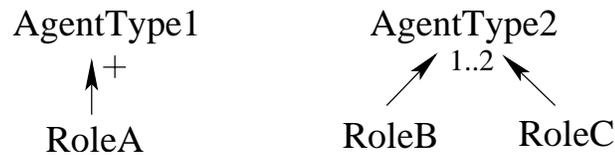


Figure 3.7: Agent Model of Gaia

A service model is used to identify services that each agent type can provide. Every service is defined with a name and its related input and output. In order to make sure services are used adequately, each service is attached with pre and post conditions. These are mechanisms to check validity before and after service execution. They are actually derived from the safety property in related roles.

The acquaintance model is to define communication topology between agent types. Designers can handle it as a prototype of an agent system at runtime. The model serves as an opportunity for designers to check if there could be any potential bottlenecks of communication at run-time. If there is such a problem, reviews of all development phases that lead to redesigned models is one way to solve it. An example diagram of

the model is shown in Figure 3.8.

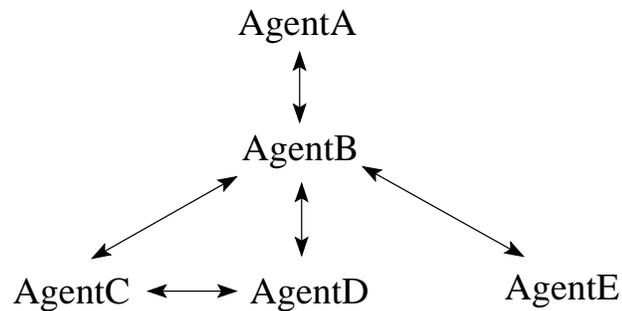


Figure 3.8: Gaia acquaintance model

The following is a critique of Gaia methodology:

- Although Gaia adapts an organizational metaphor in creating role models, which is the beginning step in the methodology, there is no clear guideline on how to derive roles from the conceptual model.
- Role modeling is created by completing all the role templates. In every role template, protocols and activities must be named and created according to its need of communication; however, it is difficult to design agent interactions well. It depends upon the assigned responsibilities among roles. Once the system becomes larger, several roles need to be handled concurrently. The complexity of applying Gaia will increase.
- Gaia has no flexibility in reflecting the changing environment because of focusing on a fixed role model. Roles identified in Gaia are peers in a flat structure. Tasks for these roles are hard-coded by their responsibilities. Their activities are fixed according to the tasks of the role. There is a lack of dynamic reasoning [12].

3.3 MaSE

MaSE, proposed by M. F. Wood and S. A. Deloach, is an abbreviation for Multiagent System Engineering [14]. MaSE methodology is to provide developers guidance from requirement statements to implement an agent system. The developing process consists of two main phases including analysis and design. In each phase, a series of steps is provided to model the system. In each step, related models are created. Models in former (upper) steps can produce output results as an input reference to the models in latter (lower) step. Transitions between steps can be traced forward and backward. Figure 3.9 shows the layered steps of MaSE.

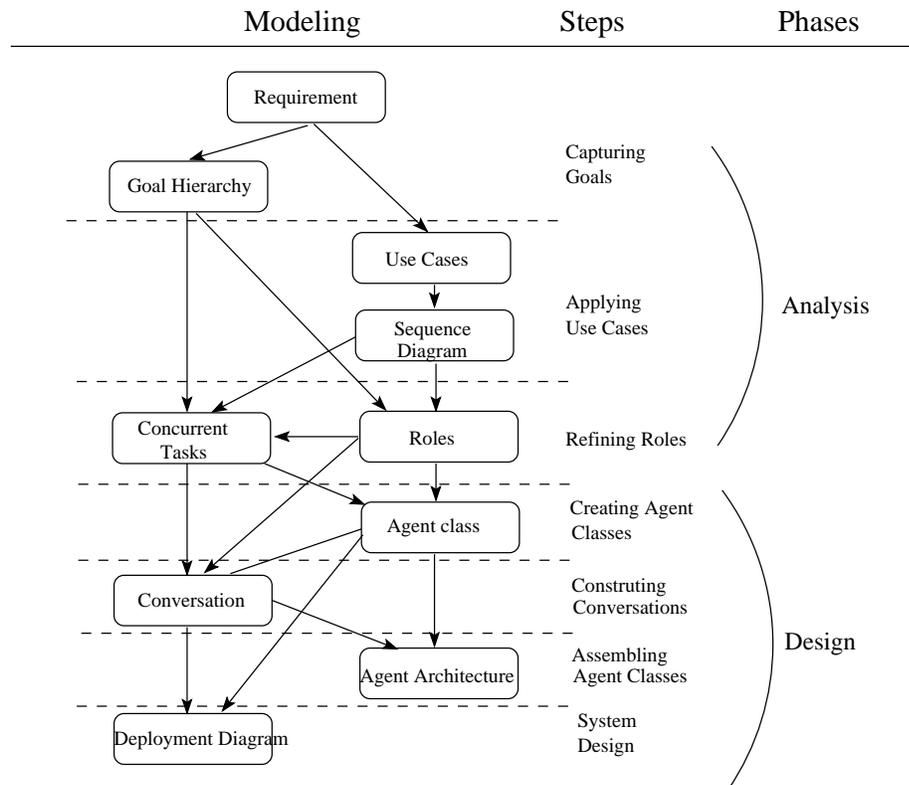


Figure 3.9: MaSE Model

An analysis phase consists of three steps: capturing goals, applying use cases, and refining roles. From requirement statements and analysis, high-level goals are identified

in the beginning step. High-level goals are then analyzed and decomposed into sub-goals. Goals and sub-goals are then accommodated in a tree-like structure. This tree structure is called a "Goal Hierarchy Diagram." The tree root, which represents the overall system goals, is branched into sub-trees, in which sub-goals are located in the child nodes. The purpose of goal hierarchy analysis is to make sub-goals more easily delegated and accommodated in the design phases later.

The second step of analysis should generate use-cases and its corresponding sequence diagram. Use-cases are generated from the same requirement statement as that which is used by goal hierarchy. With use-case scenarios in hand, all possible roles that will appear in the system can be captured. These roles become the basis of agent roles in the system. Sequence diagrams can also be generated to describe the sequence of messages between multiple agent roles.

The last step in the analysis phase is role refinement. The main task here is to map goals into roles. Every goal in the system will need a role to delegate. In general cases, one role can have one goal and its subsequent sub-goals; however, if there are several goals representing similar properties, they can be combined and delegated to a single role. In role modeling, goals and their sub-goal hierarchies associated with a role are listed under the role name. It also displays a set of tasks attached to each role. Each set of tasks is used to define role behavior. Moreover, tasks are tools to achieve the goals of a role. Communications between tasks show how these roles are interacting with each other. For each task, there is also a state diagram to depict activity details. Figure 3.10 shows a MaSE role model example.

There are four steps in the design phase: creating agent classes, constructing conversations, assembling agent classes, and system design. This is a process to create agent classes from analysis and design their interactive behavior. We obtain role models from the analysis phase and roles are the building blocks of agents. agent classes are identified

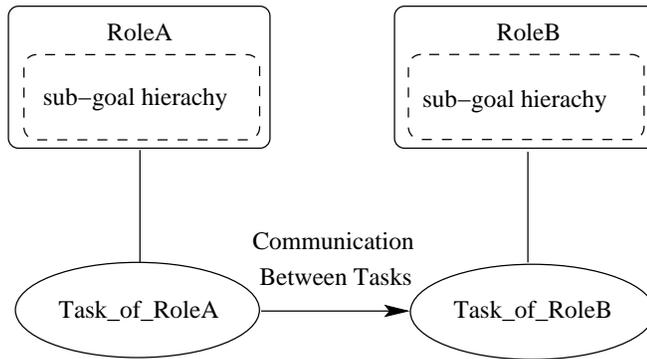


Figure 3.10: MaSE Agent Role Model

from these roles. An agent class can accommodate roles in the class and their communi-
cations. Any path between two roles becomes a conversation between two agent classes.
Thus an agent class diagram, which is inherited from a MaSE role diagram, depicts role
names in agent classes and communications between them. Figure 3.11 shows an exam-
ple of an agent class diagram. An agent class is not only an inherited role with their
activity, but it also has embedded goals that belong to the role. In this way, sub-goals
analyzed in the goal-capturing step pass into class design.

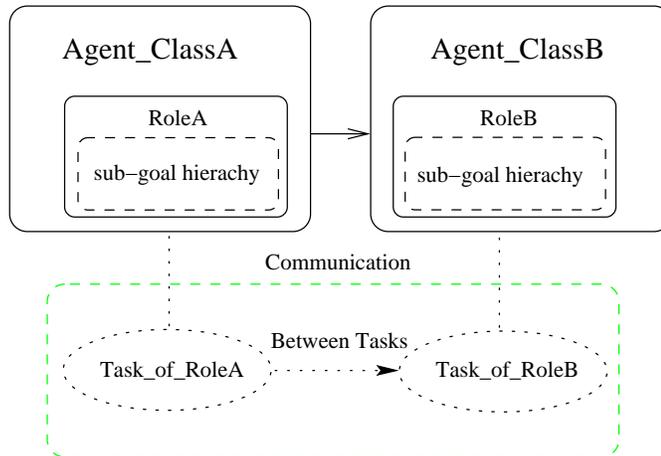


Figure 3.11: MaSE Agent Class Model

After each agent class is recognized, constructing conversation is the next step. In this procedure, designers construct conversation models used by classified agent classes. The result is a MaSE conversation model, which is a coordination of protocols between agents. State diagrams are used to describe detailed agent interaction. For each protocol, there will be a pair of state diagrams between two agent sides to depict the flow of inner states. Sequence diagrams and task diagrams obtained from the analysis phase can now be used to assure that conversation models address all agent interactions properly.

Assembling an agent class step creates agent class internals. Except implementing from scratch, designers can apply templates according to the agent class architecture. Usually, this step and the last step are related activities. Designers can alternate between these two steps depending on the features of the system. If conversation is heavier than expected or many agent classes are being reused, agent class assembly maybe executed first to reduce the complexity while adjusting class structures that affect interactions.

The final step of design is system design. It derives its result from the agent class designs and instantiates it as an actual agent. It also refers to agent conversation models and deploys agents into the computing environment. Designers at this stage must decide on the details about numbers, types, and locations of the agents deployed in the system. Deployment diagrams help designers to visualize the true scenario by drawing instances of class as cubes. Communications between agent classes are expressed as connecting lines. Topological locations are represented where agents are housed. Agents encircled by a dashed line represent location in the same platform. Figure 3.12 shows the MaSE deployment diagram.

The following is a critique of MaSE methodology:

- Goal analysis conducted at the beginning of a MaSE process reinforces goal preservation through analysis and design phases. It facilitates role modeling and agent class modeling in order to focus on clear goal delegation, in which every role is

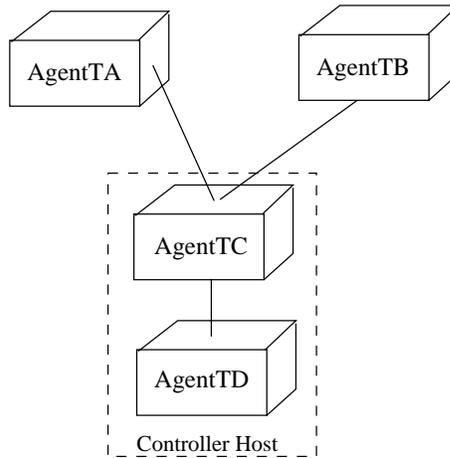


Figure 3.12: MaSE Deployment Model

responsible for its goal to be accomplished. There are tasks that belong to the dedicated goals of roles.

- In a role refinement step, it is crucial to match goals with roles. Every goal has to be associated with a role. With these roles defined, the design of communication between roles and their corresponding tasks are becoming fixed for task flow paths.

3.4 Extending UML for Modeling and Design Multi-Agent Systems

The concern for lack of pragmatic processes that can be adapted by software engineers with efficient support, a framework modeling tool is used. Rooted in BDI, it extends UML and provides integrated modeling of agent-based mechanism support. This proposal is submitted by a research team lead by K. Kavi and D. Kung [17]. As object-oriented methodology is extended to agents, extending UML makes objects agent-capable.

When we consider the important properties in Agents, mental states are central to agent modeling. BDI architecture has been adapted broadly as an agent behavioral reasoning mechanism in constructing multi-agent systems; however, it seems that engineers often use the model in a conceptual level. Implicit modeling of the BDI philosophy always generates unclear semantics in analysis and design. To facilitate a well-defined design, the framework extends UML to model specific agent scenarios. Newly invented notations are mainly used to model agent mental states, such as beliefs, goals, and plans. Figure 3.13 shows these graphical notations. Using these notations, a newly invented model for BDI architecture extending UML is introduced as an Agent Domain Model. Figure 3.14 shows the prototype of an Agent Domain Model (ADM). In the architecture, belief represents the state of a changing environment where agents are situated. Once an event happens to change the current state, beliefs should be updated. This change will affect goals, which are evaluated to reflect the update. Finally, updated plans according to goals are delegated to threads responsible for execution.

Not only does an environment change, but agents can also be influenced by other agents reporting that conditions have changed. Agent communications need to be integrated into the framework in order for an agent to share beliefs or modify execution plans. KQML and FIPA performatives are added into the framework. For ordinary

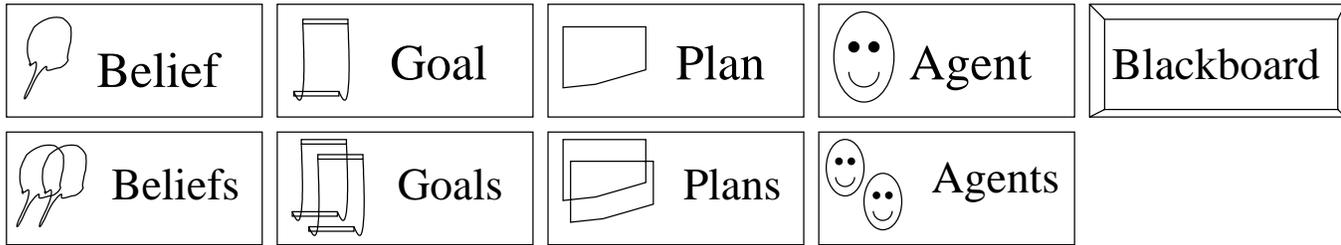


Figure 3.13: Graphic notations for Extending UML Modeling

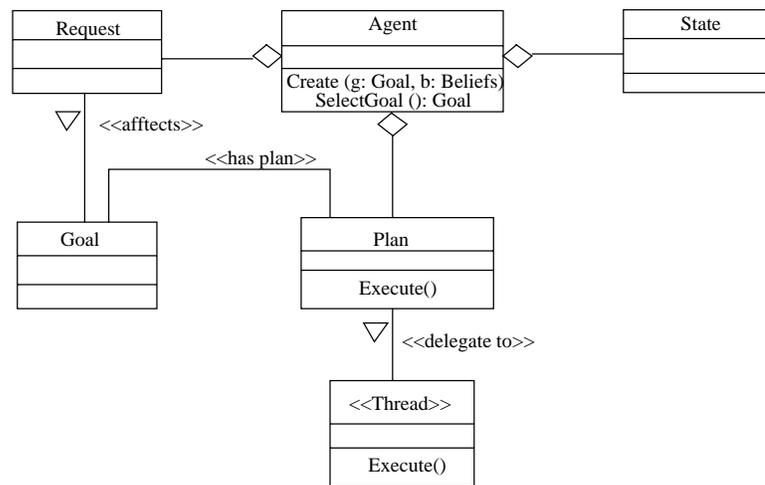


Figure 3.14: Agent Domain Model for Extending UML Modeling

agent interactions, blackboard mechanisms are adapted into the framework as a singleton class, which provides a common place for agents to manipulate shared messages.

Adapting a Unified Process as its backbone, the framework adds new diagrams ² to address both agent mental states and goal preservation through the process. There are six steps to follow in the newly introduced diagrams that facilitate the design. The first step is to identify use cases and goals from requirement. The Use Case Goal Diagram (UCGD) is used to illustrate the relationship between use-case and agent goals. Designers can realize clearly which use-case can affect which goal. Secondly, the previous

²In the following, only newly invented diagrams of the framework are addressed. Diagrams, such as Use-case, system domain, and sequence diagram etc., used in UML modeling will not be reiterated.

diagrams are refined. Goals can be refined into subgoals using a UML modeling diagram like inheritance, aggregation, and association. The third step is to refine the system and agent domain models. Refinements of goals can be used in the agent domain model to clearly state dependent relationships between agents, their corresponding internal states, and external event changes from use-case analyses. The fourth step is to detail agent activities using the Agent Sequence Diagram (ASD). The diagram not only provides agent interactive scenarios using a blackboard, but also agent inner state sequences on how goals and plans are reevaluated while belief change. The last two steps are design class diagrams and other diagrams. An Agent Design Diagram (ADD) is derived from the above mentioned agent domain diagram and agent sequence diagram. Documented with structure and behavior of agents, ADD can be implemented as a package of modules to facilitate reuse in agent design.

The following is a critique of the extending UML framework proposed by Kavi and Kung [17]:

- In the framework, goals and agent classes are keys to modeling the whole system. They are represented using the Agent Domain Model to express BDI architecture in detail.
- Relationships among agent activities and goals are modeled throughout the whole process. Goals are also represented clearly in the newly introduced diagrams, such as the Agent Goal Diagram and the Agent Sequence Diagram. It promotes goal preservation in the complete development process.
- The blackboard communication mechanism provides effective message sharing among agents. It also facilitates clear expressiveness in the agent communication diagram.
- The modeling provides great flexibility in dealing with dynamic running environments for agents. The benefit mainly results from adapting BDI architecture and provides a clear way to depict mental schemes in UML.

3.5 Engineering Multi-Agent Systems using Object-Oriented techniques

Research conducted by A. Garcia et al [20] concerns using object-oriented technique to implement applications. It would be an advantage for a developer using object-oriented programming to use similar approach in developing a multi-Agent system [19]. However, inherent natural limitations of objects are obstacles. Agents should have more complex concerns than merely objective proceedings. Motivated by the idea of separation of concerns (SOC), the framework of modeling agents using OO techniques is to map agent concerns into an object-oriented design. In the methodology, agent related concerns are first discussed. An abstract, high-level agent model is proposed. Based on the model, multi-agent systems are built by coding crosscutting concerns following the framework template using Aspect-Oriented Programming (AOP).

In the framework, agent concerns classify into three types: state, property, and roles. An Agent state consists of beliefs, goals, plans, and capabilities. Beliefs, goals, and plans are the main concerns of an agent mental state. Capabilities are specific abilities that an agent possesses. Agent properties include autonomy, interaction, and adaptation, which are all considered fundamental while leaning, mobility, and collaboration are all alternative features for agents. Autonomy is dealing with the self-governing properties in agents. An agent should be able to execute its work without external intervention. Self-contained threads are modeled in the framework to do the assignment. Interaction mentioned here mainly concerns agent communication with the external environment and among agents using sensor as an input to receive messages and actuator as an output to send messages. Adaptation is dealing with updating the mental states of agents, such as beliefs, goals, and plans. An agent should adjust itself according to external changes. Within the properties, collaboration is overlapping with interaction. The former is considered as an interaction plus coordination. Finally, agent roles are

role types played by agents to collaborate with other agents. An Agenthood model is formed from an agent state and fundamental agent properties. Agenthood is portrayed in Figure 3.15 with shadowed parts. It is the prototype to describe an agent in the framework. Every agent will be created based on this structure.

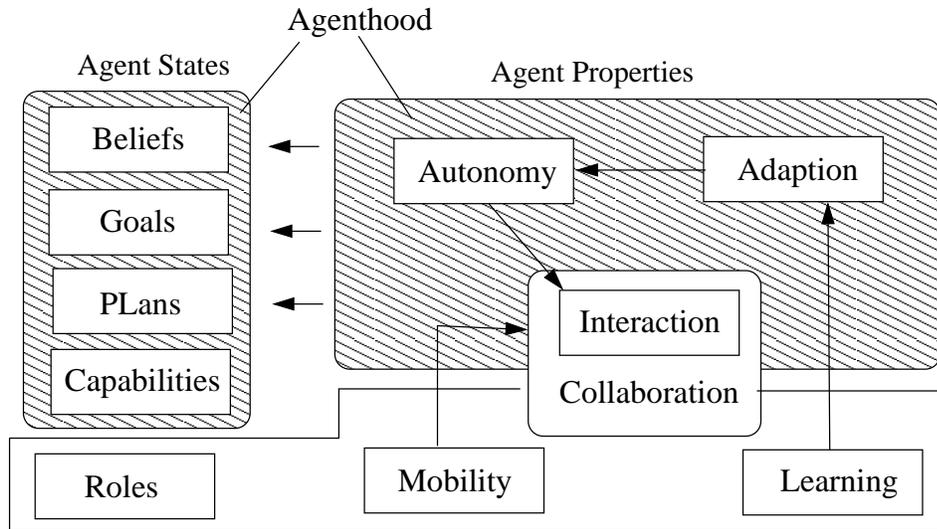


Figure 3.15: Agenthood Model

Using Agenthood definitions, an agent can be expressed in conceptual modeling, which is depicted in Figure 3.16. It describes an agent with states, fundamental properties, and specific roles situated in the computing environment. These elements are represented using circles, rectangles, and hexagons respectively. The agent's name is marked and its capabilities are outlined with points. An agent is free to add more properties and roles as needed in an application. An application of multi-agent systems will have several agents positioned in the environment. Each of these agents will have a name, corresponding capabilities, and specific Agenthood settings.

In the framework, agents are created using classes. An agent core class will have methods to represent its beliefs, goals, and plans. Every agent should instantiate from the agent core class to have competence in maintaining its own mental states and behav-

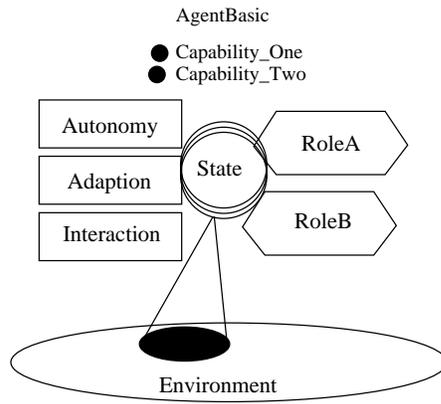


Figure 3.16: Agent prototype in Agenthood Model

ior. The methodology adopts an UML diagram as the modeling language in the design process. Different types of agents have their own capabilities implemented in their class methods. Agents represented by an UML with classes showing states and capabilities are depicted in Figure 3.17.

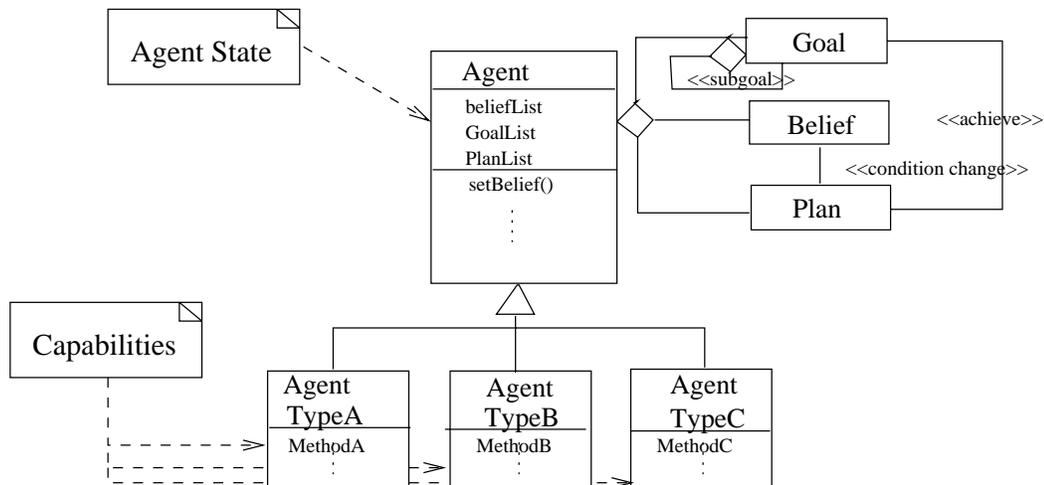


Figure 3.17: UML diagram for Agenthood Model

Aspects are used to implement Agenthood properties. These aspects are called "Agency Aspects." Agency Aspects are responsible for managing properties that an agent possesses. Classes are used to implement these Agency Aspects. As noted in the Agenthood

model, interaction, adaptation, and autonomy are the major Aspect classes that each agent can occupy. For example, every agent needs to communicate with its environment. Interactive Aspects desire to attach to an agent class to add the proficiency of interaction, which is defined in Agenthood as a template of sending and receiving messages from sensor and actuator. The same mechanisms can be applied to autonomy and adaptation. Figure 3.18 shows the template example of using Agency Aspect classes to implement an agent. Aspects are represented as diamonds. The first part of an Aspect class is introduction and the second parts are pointcuts and advices.

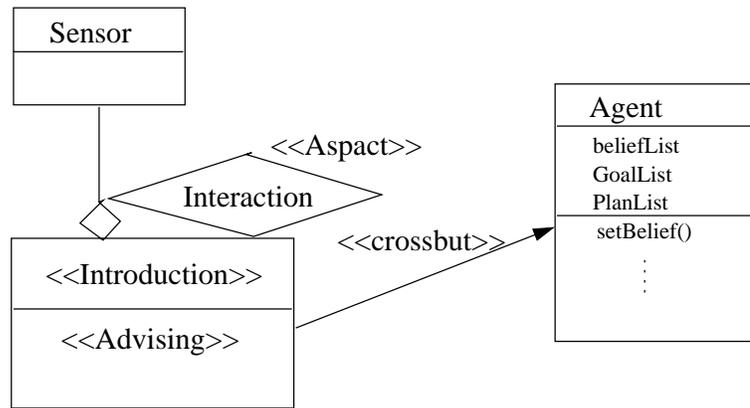


Figure 3.18: Agent Aspects Diagram of Agenthood Model

An agent role is another candidate to implement as an Aspect class. An agent can play various roles according to its different social characteristics. Role aspects provide crosscutting activities according to distinct characters in diversified situations; therefore, agents can play modified roles dynamically to interact with their peers. An example of role aspects attached to the agents is presented in Figure 3.19

The following is a critique of the Aspect oriented framework for Agents:

- Agents adopt an Agenthood model as the fixed design of agent prototype. It restricts the definition of agents in an application to use only the methodology.

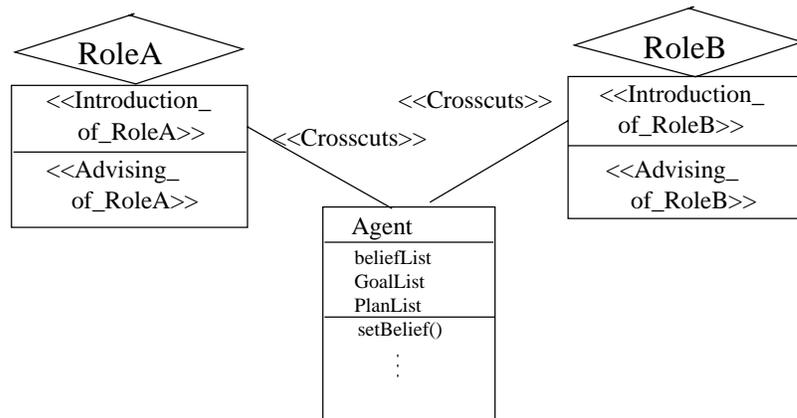


Figure 3.19: Role aspects paradigm for Agenthood UML diagram

Because Aspect-oriented programming is used, all the design has to fit into the programming semantic to be implemented.

- The benefits of using Aspects to implement multi-agent systems is flexibility and autonomy. agent properties as Aspect classes can be attached to an agent class dynamically through weaving ³.
- Methods in the Aspect classes can manage events automatically in response to changing environments. They also provide a good reuse and re-engineering ability for those alternative agent properties. An agent class can add and implement them easily by using Aspects.

³In Aspects as join points are encountered, advices can be weaved into the attached class to run additional modules of code.

Chapter 4

Summary

There are several dozen Agent-oriented methodologies that have been proposed to compete for mainstream Agent-based engineering processes. Software developers, however, do not truly benefit from these abundant choices. Lacking industrial-strength approaches and consensus in standards, there remain obstacles ahead to constrain practitioners from using a full-fledged Agent-oriented methodology [25]. In order to address these exiting problems and understand them fully, an engineer needs to integrate and propose better solutions. An evaluation has to be conducted. The result of the evaluation is not to choose a recommend methodology over others. It is still an open issue of whether there is a leading methodology or some combination of excelling technique in them to be applied as an ideal process. Comparisons about what strengths and weaknesses these methodologies have and how to improve them, can reveal what constituents the ideal methodology will possess.

Few frameworks for comparing Agent-oriented methodology have been suggested. For example, in [26], a multi-dimensional framework containing criteria in each dimensional aspect is one proposal, involving methodology, representation, Agent, organization, cooperation, and technology. Criteria in each aspect are used as indicators for comparison. The results of comparisons, notated in a two-dimensional array with criteria as rows and

names of methodology in columns, are notated as "Y" for Yes, "N" for No, "P" for possible, or simply blank (' '). These indicate the correspondent methodology ¹. In [25], four dimensions are proposed as four divisions of issues being examined. These include (1) concepts and properties, (2) notations and modeling, (3) process, and (4) pragmatics. In [29], in addition to these, support for software engineering and marketability are added into the comparison framework. The advantages of these comparisons cover both software engineering and Agent-based aspects. Within each dimension, criteria are analyzed and evaluated among different methodologies using metric scales to generate scores ². Readers can access the score table and get an overview of comparative points between methodologies. Higher scores are considered better than lower ones. However, these distinct pieces do not tell the whole story of how and why to discern different methodologies.

In [28], a different approach using goal-question-metric (GQM) is proposed to determine what to measure in comparing methodologies. An attribute tree is created according to the objectives of advanced GQM questions to identify comparison criteria. Each tree root represents a specific attribute. Child nodes and all subsequent descendant nodes are all refined properties of the attribute. Each refined indicator is evaluated to get a score. The root of each branch gets a final average from all of its direct descendants. By nature of the tree structure, a root score denotes the degree of all the issues from the attribute branch. The results of an attribute comparison can be obtained by evaluating scores generated by different methodologies using the same attribute tree branch. By asking questions that refine attributes and applying evaluations to them, the framework

¹"Y" (Yes) represents the methodology that takes criteria into account. "N" (No) represents that it does not take criteria into account. "P" (Possible) means the methodology could take criteria into account based on available information. Finally, the blank (' ') means that it is unable to make conclusion based on available information.

²Score scales differ in different frameworks. Some use number scales, such as 1-7. Others use literal descriptions and give applied degrees.

assists to obtain more precise scores on emphasized concerns in the comparison. One other comparison framework stresses measuring modeling techniques. In [27], both traditional software engineering and Agent-based system characteristics are proposed as an evaluation for the methodology. In each field, corresponding criteria are inspected. Verbal statements provide comment on each issue and give readers reasonable concepts. The final evaluation is graded incrementally as good (+), satisfying (*), or not supported (NS). This method provides a high level observation by features that examine these two engineering aspects.

It is natural that Agent-oriented methodologies stem from a combination of software engineering considerations and Agent-oriented compositions. Comparative points include criteria building blocks of both formal software processes and Agent-oriented characteristics. Referenced from viewpoints above, we construct a comparison framework to inspect the checkpoints. Considered in both aspects, four major divisions [25] are adopted in the comparison framework. To get a comprehensive insight, we form the framework with both aspect overview and rationale detail. At the overview level, a summarized checklist is provided from criteria suggested in each of the four divisions. The result will be displayed in a two-dimensional table, with criteria as rows and methodologies as columns. In each cell, "Yes" represents the methodology that takes the criteria into consideration. "No" represents that it does not. For some issues, we will provide short answers instead of simply placing "Yes" in order to make the results clear. On the detail level, logical inferences of concerns will be evaluated by asking questions. These questions are derived both from emphases on displaying logical relationships within methodological issues as well as from experiences we have obtained from case studies. The goal is to compose these questions and gain deep insight on comparisons. Answering these questions will help us to get a better rationale of each methodology, instead of just checking building elements by scores.

Progressing from Chapter 3, we will compare the methodologies of Tropos [7], Gaia[11], MaSe[14], Extending UML³[17], and OO frameworks ⁴[20]. In section 4.1, we will state the four divisions one at a time and describe checkpoints for each of them. Then, we will apply criteria to these methodologies for an overview comparison. In section 4.2, we propose questions concerning these four divisions and document answers for each question submitted. This will perform the methodological comparison in detail.

³We will use ExtUML as an abbreviation throughout the Chapter.

⁴We will use OOF as an abbreviation throughout the Chapter.

4.1 Comparisons in overview level

A. Concepts and properties

Concepts and properties collect all basic building blocks of an Agent. Primitive capabilities or characteristics of Agents are covered in this division. This category deals with questions on whether or not a methodology adheres to basic notions of Agents.

Table 4.1: Concepts and properties Comparison A

Criteria	Description
Autonomy	An Agent can make decisions on its own based on inner states without external supervision.
Mental mechanism	An Agent has mechanisms to realize its intentions by achieving goals.
Adaptation	An Agent is flexible enough to adjust its activities according to dynamically changing environments.
Concurrency	An Agent may need to perform multiple tasks concurrently.
Communication	There are protocols or mechanisms defined for Agent interactions.
Collaboration	An Agent has methods to cooperate with other Agents to achieve goals.
Agent abstraction	Methodology has theory to describe Agents using high level abstractions.
Agent-oriented	The design of methodologies originates from the consideration of Agent-oriented ways ^a .

^aThe consideration is primarily focused on whether or not the methodology addresses Agent-based features in the beginning of the analysis and design.

Tropos[7]: An agent maps its mental states and capabilities obtained from analysis into implementation by using BDI architecture. Autonomy, adaptation, and communication ⁵ are embedded in the Agent class. Concurrency is available from the design

⁵In the comparison, we will consider Tropos adopting JACK as its implementation toolkit. By de-

in which Tropos collects all capabilities into its Agent types. These Agent classes can provide many services simultaneously. Collaboration is already well-defined in dependency relationships.

Gaia[11]: An Agent is a representative to one or more roles. A role is modeled by its responsibility using the properties of either "liveness" and/or "safety". Autonomy is expressed implicitly through the property. Adaptation is addressed by elective functional equations of "liveness". A role can automatically choose its action according to the situation at the time. There are defined interaction models for services among Agents, but no further communication between Agents. Collaboration is achieved by services modeling among roles.

MaSE[14]: The implementation of Agent classes stems from role modeling. A role is embedded in its goals and associated tasks. These tasks are autonomous in achieving goals by communicating with other Agents. However, an Agent cannot respond to its environment by changing its goals. Adaptation is an aspect that cannot be reached. Concurrency and collaboration can be achieved by task modeling with communication among Agents.

ExtUML[17]: An Agent is modeled by its belief, goals, and associated plans. Every plan has a thread to execute its goals. As beliefs change, Agents can select which goal is the current best to pursue. If a goal is renewed, an associated plan will be executed. This clearly shows how Agents in the system can be autonomous and adaptable. Every plan has its own thread to run its tasks in an Agent. Concurrency is achieved when an Agent is able to run several tasks concurrently by multi-threading. Collaboration among Agents is reached by using defined communication mechanisms, which include FIPA performatives and blackboard.

OOF[20]: An Agent is modeled by Agenthood, which contains the properties of auton-

fault, JACK uses a fast TCP-based protocol for communicating with other Agents, back-ended systems, and GUIs.

omy and adaptation. An autonomous property is described as an Agent that would have its own control thread, which can decide whether or not to accept incoming messages. The adaptation property is one that can update beliefs and goals of an Agent if it accepts messages from the environmental sensor. Both autonomy and adaptation are defined explicitly as aspects in the Agenthood model. Concurrency is also accomplished by running plans with distinct threads in each Agent class incident. Collaboration property is defined as an optional aspect of the Agenthood in which an Agent participates in conversation with other Agents.

Table 4.2: Summary of Methodological Comparison A

	Tropos	Gaia	MaSE	ExtUML	OOF
Concepts and Properties (A)					
Autonomy	Yes	Yes	Yes	Yes	Yes
Mental mechanism	Goal, soft-goal tasks	No ^a	Goal, tasks	BDI	BDI
Adaptation	Yes	Yes	No	Yes	Yes
Concurrency	Yes	Yes	Yes	Yes	Yes
Communication	Yes	No details	No details	Yes	Yes
Collaboration	Yes	Yes	Yes	Yes	Yes
Agent Abstraction	Social actors	Roles in organization	Roles	Mental entity	Agenthood
Agent-oriented	Yes	Yes	Yes	Yes	Yes

^aMental mechanisms are not specified in Gaia. However, "liveness" and "safety" properties are tools to express an Agent's behavior.

B. Notations and modeling techniques

Notations and modeling techniques are keys representing elements and activities in a system. During a software development process, consistent expressiveness is a help to address an Agents behavior and it helps efficient communication between engineers. Good modeling can ease the complexities of understanding and implementing systems from concepts to realizations. This discussion deals with the tactics of notations and models that are manipulated in a methodology.

Table 4.3: Notations and modeling techniques Comparison B

Criteria	Description
Expressiveness	Notations are used in this methodology to help design processes.
Complexity management	There are abstract levels from low to high that engage a complex problem with modeling.
Modularity	Using components or modules in the methodology to model in an incremental manner.
Executable	Models used in this methodology are capable of generating or simulating prototypes in some aspect of the specification.
Refinement	A modeling technique that refines factors into simpler entities in order to take advantage of them.
Traceability	Within the methodology, it tracks dependencies between models.

Tropos[7]: Modularity is not available during an analysis. However, social structural patterns have been introduced in the design. This helps to modularize some patterns of recognition. Refinement is achieved by analyzing goals or tasks into sub-goals or sub-tasks.

Gaia[11]: An organizational role is the central idea to form a modularity. Models describe functionalities of roles and their communication protocols. There is no refinement at all for roles. However, the responsibilities of an Agent can be decom-

posed into protocols and activities within itself to cooperate with other Agents. Traceability is clear in the transition from roles to Agent types.

MaSE[14]: Modularity is formed by combining goals and roles to establish a MaSE model. Refinement is unambiguous in the modeling of goals and roles to effectively use them. Traceability is consistent principally because of the precise transition between role modeling and Agent class modeling.

ExtUML[17]: The modularity of an Agent's behavior can be recognized in patterns of an Agent domain and sequence diagram. Goals recognized in each Use case are refined into sub-goals in the modeling. Traceability is available by referring to Agent related models used in the design.

OOF[20]: Aspects of Agent properties are module-based. Aspects cover Agent concerns, properties, and roles. Based on Agenthood prototypes, an Agent can be modeled by attaching these features. These aspects can be understood as refined factors of an Agent's functions. These aspect classes are treated as base elements in the design without additional refinement.

Table 4.4: Summary of Methodological Comparison B

	Tropos	Gaia	MaSE	ExtUML	OOF
Notations and Modeling technique (B)					
Expressiveness ^a	Yes	Yes	Yes	Yes	Yes
Complexity management	Decompose of goals, tasks	Role	Goal, role refinement	Goal re-finement	Property aspects
Modularity	Yes	Yes	Yes	Yes	Yes ^b
Executable	No	No	No	Yes ^c	No
Refinement	Yes	No ^d	Yes	Yes	No
Traceability	Yes ^e	Yes	Yes	Yes	Yes

^aTropos and ExtUML have a designed set of new graphical notations to facilitate the expressiveness during an analysis or design phase. Gaia and MaSE use modeling flow charts to express transitions between different models. Except for Gaia, all the methodologies adapt UML as a companion in the modeling design.

^bBased on Agent prototyping, most modules used in OOF are Agent aspects. Modules are used to represent aspects by implementing classes.

^cThe possible protocols of Agents can be simulated by using Extending UML, an Agent domain model, and an Agent sequence diagram to describe Agents and their behavior. It is also a flexible method for modeling different aspect scenarios of an Agent's activity. This takes advantage of refinement in mental states using interaction sequences.

^dThere is no role or its corresponding responsibility that can be refined using this model.

^eAlthough traceability is available in the process all the time, a system-to-be actor introduced in the late analysis phase would make it inconsistent. It may require additional notes to document it.

C. Process

A process is a series of steps that guide practitioners to construct a software system from the beginning to the end. It serves as a detailed guideline of all activities throughout subsequent phases. This discussion deals with the investigation of development processes for a methodology.

Table 4.5: Process Comparison C

Criteria	Description
Specification	This methodology provides ways of forming a system specification from scratch.
Life-cycle coverage	This methodology covers steps from analysis, design, implementation, and testing throughout system development.
Architecture Design	This methodology provides mechanisms to facilitate design by using patterns or modules.
Implementation Toolkits	This methodology provides suggestions on how to implement Agents in the system.
Deployment	This methodology concerns the practical deployment of Agents.

Here is a summary of life-cycle coverage for these methodologies:

Analysis: There are different analysis concepts applied in the early stages of each methodology, except OOF. OOF adapts an Agenthood model as a ready-to-use analysis.

Design: According to analysis, models are evolved from an analysis phase to form basic Agent types and classes.

Implementation: Once Agent classes are defined, they are considered for implementation. Agents are mostly implemented as active objects or threads to run tasks. Gaia only discusses the previous two stages of analysis and design in its model with

no regard for implementation. OOF takes advantage of using Aspect-Oriented Programming, which provides ready-to-use steps in the implementation.

Reuse and Maintenance: ExtUML provides reuse through its fundamental adopting of BDI architecture. Accompanied with an Agent sequence diagram, an Agent's behavior is well prototyped and can be treated as modules in reuse. OOF provides reuse through its disposition of using Aspect-Oriented Programming.

Testing: No main AOSE methodology deals with test issues in any practical executable form. This is a potential area of Agent-oriented software engineering that has a strong future.

Table 4.6 shows summarization of software developing lifecycle applied in each methodology.

Table 4.6: Life-cycle coverage of methodologies

	Tropos	Gaia	MaSE	ExtUML	OOF
Analysis	v	v	v	v	N/A
Design	v	v	v	v	v
Implementation	v	N/A	v	v	v
Reuse	N/A	N/A	N/A	v	v
Maintenance	N/A	N/A	N/A	v	v
Testing	N/A	N/A	N/A	N/A	N/A

Table 4.7: Summary of Methodological Comparison C

	Tropos	Gaia	MaSE	ExtUML	OOF
Process (C)					
System specification	Stakeholders analysis	Role analysis	Use-cases goal and role analysis	Use-cases and goal analysis	Aspects analysis
Life-cycle coverage ^a	Yes	Yes	Yes	Yes	No
Architecture Design	Yes	No	Yes	Yes ^b	Yes ^c
Implementation	Yes	No	Yes	Yes	Yes
Deployment	No	Yes ^d	Yes	No	No

^aThe lifecycle coverage is considered "Yes" if the methodology covers phases with at least Analysis and Design. For additional details, please refer to table 4.6.

^bIn ExtUML, any type of Agent in an application is implemented by an inheritance from BDI architecture and an Agent sequence diagram. The structure reinforces applications to reuse an existing design by which to form patterns from them.

^cIn OOF, the foundation of architectural design is formed by an Agenthood model. Every Agent in the system will use similar aspects as its basic modules. These aspects facilitate patterns of reuse.

^dGaia only provides information about the quantity of Agent types in a system through Agent model types. For each Agent type location, an Agent acquaintance model can be seen as a topological model to demonstrate how Agents are being deployed.

D. Pragmatics

Pragmatics refer to real use scenarios as developers apply methodology in building Agent-based systems. This provides reviews in real situations from instituting concepts, building models, to implementing details. This division deals with the exploration of practical deployment while using a methodology.

Table 4.8: Pragmatics Comparison D

Criteria	Description
Tools available	There are resources and tools ready to use in methodology.
Required expertise	There is a required background or assumption to apply in the methodology.
Modeling suitability	This methodology is based on a specific architecture.
Domain applicability	This methodology is suitable for a specific application domain.
Scalability	This methodology is able to handle a reasonable number of Agents in an application.

Tropos[7]: Although it uses a UML diagram in its design, there is no other tool available that can assist designers in the analysis and design phase. There is no tool accessible that can help in the transition from design to implementation using JACK. More experience is needed to execute a good system construction.

Gaia[11]: No tool exists for design and analysis. Designers must find a way to complete schemas and forms in the modeling. To improve the concept, designers are required to familiarize themselves with the properties of "liveness" and "safety".

MaSE[14]: An AgentTool is the utility that provides an interface to assist practitioners in running the design process, which automatically generates and checks the relationships of elements in each modeling [15].

ExtUML[17]: Use-Case and UML diagrams are used in this design. Because it adopts a Unified Process as the backbone of development, all the tools available to it also

apply to the methodology.

OOF[20]: The primary tool in this methodology is the Aspect-Oriented Programming itself. In each Agent, codes are filled in the basic aspects of an Agenthood model.

Table 4.9: Summary of Methodological Comparison D

	Tropos	Gaia	MaSE	ExtUML	OOF
Pragmatics (D)					
Tools available	No	No	Yes	Yes	Yes
Required expertise	No ^a	No ^b	No	No ^c	Yes ^d
Modeling suitability	BDI	No	No	BDI	Agenthood
Domain applicability	Yes	Yes	Yes	Yes	Yes
Scalability	Yes ^e	Yes ^f	Yes	Yes	Yes

^aThe analytical algorithms used in Tropos let practitioners decompose relationships among stakeholders freely. However, designers may be required to have experience to create a more efficient way.

^bExploiting role models in Gaia, designers need to familiarize themselves with how to design relative activities in a role that will constitute communication protocols.

^cDesigners should familiarize themselves with the tools of UML and Unified Process, which are popular methodologies in Object-Oriented Design.

^dA knowledge of Aspect-Oriented Programming is required and crucial to both design and implementation.

^eIf there are many stakeholders in the system of analysis, it will make the system complicated and difficult for developers to do an analysis well.

^fThe overall system required to contain less than 100 Agent types [11].

4.2 Comparisons in detailed level

In the following, we propose questions categorized in the four comparison divisions, which provide answers with discussions. These can be referenced in comparing methodologies.

(A) Concepts and Properties:

QA1. From what concept does a root of methodology come and what are the advantages?

Ans: A role concept is being used in most of the methodologies. Roles can have specific responsibilities in a social or organizational setting. Agents are designed to represent abstract concepts while running corresponding tasks that can fulfill their goals. Governed by a hierarchy, an Agent's communication can be regulated by the rank of roles. Role modeling provides designers a way to think concerning an Agent's activities at the run-time, and to effectively describe them. Gaia and MaSE are the main representatives that use role modeling.

BDI architecture is also a good tool to use in modeling an Agent's behavior with mental states. Without mental notation, it is difficult to describe an Agent by active interactions only. ExtUML presents BDI in a clear and detailed manner that visualizes the inner structure of Agents, and designers can take advantage of this to understand better an Agent-based system.

QA2. How is an Agent created in a methodology?

Ans: In Tropos, Gaia, and MaSE, Agent classes are usually transformed from role concepts. As roles are refined into sub-roles, corresponding tasks are refined into sub-tasks and roles are assigned with proper responsibilities. It is then that roles are realized by Agent types. Agent classes are implemented for Agent types, and an Agent capability is created within an Agent class.

In ExtUML, Agents are modeled by BDI architecture with UML. An Agent class contains its own beliefs, goals, and plans. These plans are executed by using threads, and tasks are designed according to goals. In OOF, a similar design is used with BDI architecture as its Agent core class.

QA3. How well constructed is the design that deals with Agent mental mechanisms?

Ans: The earlier a mental state mechanism is analyzed, the easier it is to model an Agent-based system in following processes. ExtUML and OOF are two of the comparison methodologies dealing with mental mechanism from the beginning.

Tropos and MaSE use goals and intentions from the beginning of an analysis. By analyzing relationships between roles, these intentions are implicitly modeled by interactions among Agents. Although goals are accomplished by tasks, it is difficult to handle beliefs and desires in a flexible way, because no precise mental mechanism is mentioned. In Gaia, mental mechanisms are represented by a "liveness" property in a role schema. An Agent type can change its plan by applying different "liveness" equations. Goals are fixed in the design so MaSE lacks flexibility in dealing with a changing environment.

QA4. How well does a design deal with an agent's perception of its environment, and what is its response to it?

Ans: As mentioned in QA3, a design process with explicit mental mechanisms will have a better modeling and performance in answer to its environment.

QA5. How efficient are Agents in achieving their goals?

Ans: Goals are the main reasons that Agents exist. In most of the designs of comparison methodologies, specific tasks are implemented to fulfill goals. There

is no difference in running time. In a dynamic environment, however, Agents must contend with the possibilities of goal conflicts. Mental reasoning and negotiations among Agents play decisive roles. Agent design should be embedded with mental mechanisms as well as efficient Agent-based communication to excel in accomplishing goals. ExtUML is most likely the one to realize goals with better performance.

(B) Notations and Modeling Techniques

QB1. How well are notations and models formed to address Agent-based system scenarios?

Ans: Tropos and ExtUML define their specific notations to assist designers in developing Agent-based systems with analysis and visualization tools. These notations represent goals, tasks, and Agents in their relationship of dependencies. Using these notations, designers can gain a more clear idea on Agent interactions in the system. In Gaia and MaSE, models are used to present Agents with their functionalities, such as in models of roles and interactions. Normally, models are used to present an overview of Agents in the system.

QB2. How consistent and unambiguous are models while running the process?

Ans: Usually, modeling used in these comparative methodologies is consistent and unambiguous. Only minor problems exist in the transition of phases. For example, in Tropos late requirement analysis, analysis diagrams will be affected by a newly added system-to-be actor. It is unlikely to maintain a consistent manner in analysis from the beginning. Another example, from role model to agent model in Gaia, designers have to make decisions on what and how many agent types will be created. There is no clue provided to be referred from agent role schema design. There is also a potential risk while using multiple same agent type. It could lead to contention problems due to no specific guideline of supporting how agents cooperate with each other.

QB3. How well is the modeling technique addressing traceability and reuse?

Ans: MaSE is the one that emphasizes models that can be traced back and forth in each analysis and design layer. Modeling in each layer is derived from its upper layer with explicit rules smoothly. ExtUML mentioned reusing an existing design or implementation by inheriting agent abstract feature. In OOF, it is also natural to reuse all its aspects of Agent concerns.

QB4. How well does the modeling technique represent Agents?

Ans: Methodologies using Agent-based features, such as goals or mental states, to analyze and model the system are better in describing Agents in the modeling. For a counter-example, Gaia did not use any explicit goals or mental states to model the system. As a result, the overall system has the less explicit Agent-oriented feature and flexible management properties in Agents.

(C) Process

QC1. How well does the methodology define system domain?

Ans: In Tropos, Gaia, and MaSE, methodology uses role concepts to explore its stakeholders, which depicts main system specification. In MaSE and ExtUML, Use-cases and UML modeling are used for the system specification. Both methods are ways to explore system domain by extracting roles or agents from requirement statement. In ExtUML, system domain that consists of internal mental states is expressed by using BDI structure. It provides more precise expressions on agent features in the system specification. In OOF, Agenthood is used as the core model for any system. Agent features are expressed by Aspects when no system domain is specified.

QC2. How well does the process cover the whole lifecycle of development?

Ans: Most Agent-based methodologies cover analysis and design phases in the developing process. In the implementation phase, some suggest applying

agent-based implementation toolkits. For example, Tropos adopts JACK to be the implementation toolkit because it easily maps BDI architecture. In most cases, Agents are implemented as object classes by recognizing Agent types analyzed from design phases. Also, reuse and maintenance of designs is seldom available. In the comparison cases, only ExtUML and OOF can barely meet the requirement. In ExtUML, using BDI architecture in design is the key that provides good Agent-based maintenance to the process. Using the architecture, some patterns can be preserved and reused. Likewise, OOF also adopts BDI architecture, and takes advantage of aspects to reinforce its reuses. Maintenance can be achieved by using different aspects according to various requirement changes.

QC3. How well are transitions between phases in a process to preserve goals?

Ans: In each Agent-based methodology, Agents should be managed to bear their goals and achieve them successfully. Keeping goals in each design process for each Agent types is crucial to a successful process. Table 4.10 shows how goals play a role in the processes of methodologies. Observation is based on each column of a methodology, Tropos, ExtUML, and OOF preserve Goal(G) throughout the whole life-cycle.

Table 4.10: Agent-based elements involved in life-cycle process (G: Goal, R: Role, A: Agent)

	Tropos	Gaia	MaSE	ExtUML ^a	OOF
Requirements	G,R	R	G	G,A	A
Analysis	G,R	R	G,R	G,A	G,R,A
Design	G,R,A	R,A	R,A	G,A	G,R,A
Implementation	G,R,A	A	A	G,A	G,R,A

^aRole can be easily designed from deriving general agent types.

(D) Pragmatics

QD1. Is the methodology easy to use?

Ans: From an empirical study of these methodologies, Gaia is the simplest to use in constructing a prototype of an Agent-based system. MaSE provides improved layered steps and phases while building models. Tropos emphasizes its notations used throughout the design process; but rationale analysis introduces complexity. ExtUML uses the benefit of mature processes in developing object-oriented models. Consequently, it provides easily understandable and usable tools in the design. OOF adopts Aspect-Oriented Programming to ease the complexity of implementing Agent-based systems. It lacks analytical steps to accommodate problems in the programming.

QD2. Do agent abstract concepts evolve easily?

Ans: Deriving an analysis from a requirement statement using abstract concepts is not difficult to do; however, in most methodologies there is no explicit rule for developers to follow. Because experiences and emphases differ with developers, the exploration of initial stakeholders could alter. In Tropos, especially, exploring dependencies between stakeholders is not an easy task. Practitioners must assure that the analysis proceeds in the right direction.

QD3. Is Agent-oriented methodology flexible enough in re-engineering?

Ans: The most difficult is Tropos. Designers should re-do all the analyses from initial stakeholders. Gaia, also, has a chance to encounter problems. Once communication bottlenecks occur, the design of Agent roles and interactions needs to be revised from the beginning.

ExtUML and OOF are flexible. These methodologies model each Agent as a unique module. The Agent module is easy to revise and maintain within the system design.

QD4. Are paradigm and architecture suitable in general cases?

Ans: Most of these comparison methodologies are good for general cases.

4.3 Summary

Observed from the comparison above, a good methodology for MAS should have the following:

- A good mental mechanism to support Agents' autonomy, adaptation, and collaboration.
- Communication protocols are crucial to Agents in the system in order to conduct their tasks.
- Goal-oriented methodology should be preserved in Agent at all times, which includes goal management.
- Practical conceptual theories are needed to provide execution of the methodology to ease the complexity of the design.
- Notations for clear expressions and efficient modeling is a key to a successful methodology, which can facilitate easy-to-use applications of the methodology.
- Executable and reliable full life-cycle software engineering process has to be addressed.
- Tools and modeling has to be available pragmatically.
- Module and refinement capabilities are needed to analyze and integrate elements in the system.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Software agent technology has drawn much attention within the study and practice of developing software systems. Agent-based systems are often featured with intelligence, autonomy, and reasoning. These promising attributes of agents quickly become appealing to both legacy systems and modern ones. Agents are building blocks in these software systems, while combinations of attributes are alloyed to form the software entities. The more complexity an agent-based system has, the more sophisticated the methodology must be. Coincidentally, there are no consensus standards on how to create agents or model them in the developing process. It appears that the study of various proposals for creating agent-based systems is the way to gain ideas on what attributes the useful constituents are, so that we will have better way to integrate them into pragmatic use.

In this work, we explored applications of agent-based systems categorized in different fields of software systems. With this in mind, we described what properties agents should have, as well as the ones that help to form an agent society in terms of system. We created a baseline to help us clearly focus on the core of agents throughout the study. We investigated both object-oriented and agent-oriented techniques available

for constructing agent-based systems. In each respect, we addressed theoretical backgrounds on how tools and components can be applied to provide the elements of agent infrastructure. Each one of these techniques in the survey is provided with examples of state-of-the-art research. Standing on the fundamental knowledge evolved from two different respects, we studied various representative agent-oriented software engineering methodologies. We investigated agent-based components as well as software engineering processes in each one of them, and tried to determine the rationale expressed in the processes. To develop a deeper insight regarding the merits and defects, we created a framework with four divisions synthesized upon agent-oriented, as well as software engineering criteria, for a comparison of the methodologies. The framework is composed of two levels. In overview level, evaluation has been obtained by determining whether criteria have been met by the methodology. In detailed level, we proposed weighted questions concerning logical relationships among these criteria, and provide answers as statements for comparison.

Agent-oriented computing can be realized as integrations of virtually different phases, from conceptual genesis to practical accomplishment. In each phase, coordination between its primary elements is fine-grained. Different combinations and applications of layout in creating agents and methodology could lead to significantly different results of the system. The process of organizing these virtual phases is also subject to software engineering procedures. It is crucial in our work to study these building elements and methodologies, as well as comparisons between them, so as to find preferred ingredients and pragmatic methods to comply with. On the other hand, it also provides opportunities to highly refine them in respect to fine-grained elements in each virtual phase. Such refinement, focusing on favored agent-based elements, may facilitate better modeling of agents to software methodological solutions.

5.2 Future Work

There are many tangible techniques and solutions ready to be used in building agent-based systems; however, few of them have been proved to be industrial strength. Most of the agent-oriented software engineering methodologies lack formal methods of verification and testing of the design and implementation. Although we have done the survey of techniques, study on methodologies, and comparison to solutions, a formal method should be introduced to ensure validation and verification of these fine-grained elements, as well as a full life-cycle coverage to engineer them. Some related issues that need to be addressed in our future work, include:

- A framework needs to be set up that is capable of addressing issues on modeling validation and verification for the assurance of goal fulfillment.
- Formalism has to be brought into the modelings throughout the whole engineering process, which is from specification formalism to goal-oriented proofs.
- To automate the software engineering process, a middleware or utilities to map analysis and design into code generating is to be designed.
- Rules of statistical analyses are to be established in the framework to assist agent and system behavior modeling.
- Testing environment and procedures need to be designed so as to accommodate prototypes of the framework.
- Real case study has to be conducted to apply the design in the framework from formalism specification to system testing.

Bibliography

- [1] N. R. Jennings, On agent-based software engineering, *Artificial Intelligence* 117, p277-296, 2000
- [2] N. R. Jennings and M. J. Wooldridge (1998) "Applications of Intelligent Agents" in *Agent Technology: Foundations, Applications, and Markets* 3-28
- [3] M. A. Aborizka," An Architectural Framework for the Specification, Analysis and Design Intelligent Real-Time Monitoring Agent Based Software Systems", PHD Dissertation, Huntsville, Alabama, 2002
- [4] A. S. Rao, M. P. Georgeff, Modeling Rational Agents within a BDI-Architecture, *Proceedings of the second international conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, 1991, pp.473-484
- [5] J. Castro, M. Kolp, J. Mylopoulos, *Towards Requirements-Driven Information Systems Engineering: The Tropos Project*, Information Systems, Elsevier, Amsterdam, The Netherlands, 2002
- [6] M. Kolp, J. Castro, J. Mylopoulos, A social Organization Perspective on Software Architectures. In *Proceedings of the First International Workshop From Software Requirements to Architectures (STRAW 01) at ICSE 2001*, 14 May 2001, Toronto, Canada.

- [7] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, A. Perini, TROPOS: An Agent-Oriented Software Development Methodology, Technical Report #DIT-02-0015, AAMAS Journal
- [8] M. Kolp, P. Giorgini, J. Mylopoulos, A Goal-Based Organizational Perspective on Multi-Agent Architectures., In Proceedings of the Eighth International Workshop on Agent Theories, architectures, and languages (ATAL-2001), Seattle, USA, August 1-3, 2001.
- [9] A. Perini, A. Susi, Discussing strategies for software architecting and designing from an Agent-oriented point of view, ICSE'03 Portland, Oregon
- [10] A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, J. Mylopoulos. A Knowledge Level Software Engineering Methodology for Agent Oriented Programming. Proc. of the Fifth International Conference on Autonomous Agents, Montreal, Canada, 28 May - 1 June 2001.
- [11] M. Wooldridge, N. R. Jennings, D. Kinny, The Gaia Methodology for Agent-Oriented Analysis and Design, Autonomous Agents and Multi-Agent Systems,3 285-312, 2000
- [12] T. Juan, A. Pearce, L. Sterling, ROADMAP: Extending the Gaia Methodology for complex Open Systems, Autonomous Agents and Multi-Agent Systems, Bologna, Italy, 2002
- [13] F. Zambonelli, N. R. Jennings, M. Wooldridge, Organisational Abstractions for the Analysis and Design of Multi-Agent Systems, AOSE, 2000
- [14] M. F. Wood, S. A. DeLoach, An Overview of the Multiagent Systems Engineering Methodology, in Agent-Oriented Software Engineering. P. Ciancarini, M. Wooldridge, (Eds.) Lecture Notes in Computer Science. Vol. 1957, Springer Verlag, Berlin, January 2001.

- [15] S. DeLoach, Analysis and Design using MaSE and agentTool, Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001). Miami University, Oxford, Ohio, March 31 - April 1, 2001.
- [16] J. Odell, H. V. D. Parunak, B. Bauer, Extending UML for Agents, AOIS Workshop at AAAI 2000.
- [17] K. Kavi, D. C. Kung, H. Bhambhani, G. Pandcholi, M. Kanikarla, R. Shah. "Extending UML to modeling and design of multi agent systems", Proc. of 2nd Intl Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS2003), held in conjunction with the International Conference on Software Engineering, Portland, OR, May 3-10, 2003
- [18] J. Sardinha, P. Ribeiro, R. Milidi, C. Lucena, An Object-Oriented Framework for Building Software Agents, Journal of Object Technology, vol. 2, no. 1, January-February 2003, pages 85-97.
- [19] A. Garcia, C. Sant'Anna, C. Chavez, V. Silva, C. Lucena, A. V. Staa, "Agents and Objects: An Empirical Study on the Design and Implementation of Multi-Agent Systems". Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003) at ICSE 2003, Portland, USA, May 2003.
- [20] A. Garcia, V. Silva, C. Chavez, C. Lucena, "Engineering Multi-Agent Systems with Patterns and Aspects". Accepted to appear in Journal of the Brazilian Computer Society, SBC, Special Issue on Software Engineering and Databases, 2002.
- [21] S. Hayden, C. Carrick and Q. Yang, Architectural Design Patterns for Multiagent Coordination, In Proceedings of the International Conference on Agent Systems '99. (Agents'99) Seattle, WA, May 1999.

- [22] D. Bumer, D. Riehle, W. Siberski, M. Wulf, Role Object, In Pattern Languages of Program Design 4. Edited by Neil Harrison, Brian Foote, and Hans Rohnert. Addison-Wesley: 2000. Chapter 2, page 15-32.
- [23] E. A. Kendall, Role Model Designs and Implementations with Aspect Oriented Programming, Proceedings of the 1999 Conference on Object- Oriented Programming Systems, Languages, and Applications (OOPSLA'99), ACM Press, November, 1999
- [24] E. A. Kendall, C. V. Pathak, P. V. M. Krishna, C. B. Suresh, The Layered Agent Pattern Language, Pattern Languages of Programming (PLOP'97), September, 1997.
- [25] A. Sturm, O. Shehory, A Framework for Evaluating Agent-Oriented Methodologies, Fifth International Bi-Conference Workshop on Agent-Oriented Information System (AOIS-2003)
- [26] A. Sabas, M. Badri, S. Delisle, A Multidimensional Framework for the Evaluation of Multiagent System Methodologies, Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI-2002), Orlando (Florida, USA), 14-18 juillet 2002, Volume I, 211-216.
- [27] O. Shehory, A. Sturm, Evaluation of Modeling Techniques for Agent-Based Systems, AGENTS01, February 11-13, 2001, Montreal, Quebec, Canada
- [28] L. Gernuzzi, G. Rossi, On The Evaluation Of Agent Oriented Modeling Methods, In Proceedings of Agent Oriented Methodology Workshop, Seattle, November 2002.
- [29] K. H. Dam and M. Winikoff, Comparing Agent-Oriented Methodologies, Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems(AOIS-2003) 14 July 2003, Melbourne, Australia, at AAMAS'03

- [30] Y. Aridor, D. B. Lange, "Agent Design Patterns: Elements of Agent Application Design", Proc. 2nd Int'l Conf. on Autonomous Agents (Agents '98), ACM Press, 1998, pp. 108-115
- [31] Tveit, A Survey of Agent-Oriented Software Engineering, In: NTNU Computer Science Graduate Student Conference, Norwegian University of Science and Technology, Trondheim, Norway (2001)
- [32] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
- [33] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. An overview of aspectj. In ECOOP, pages 327-353, 2001
- [34] Nwana, H., Ndumu, D., Lee, L., Collis, J.: ZEUS: a Toolkit and Approach for Building Distributed Multi-Agent Systems. ACM International Conference on Autonomous Agents (AA) 1999, Seattle, USA (1999)
- [35] F. Bellifemine, A. Poggi, G. Rimassa, "JADE – A FIPA-compliant agent framework ", CSELT internal technical report. Part of this report has been also published in Proceedings of PAAM'99, London, April 1999, pp.97-108
- [36] N. Howden, R. Rnnquist, A. Hodgson, A. Lucas, JACK Intelligent Agents - Summary of an Agent Infrastructure, 5th International Conference on Autonomous Agents, 2001