A Seamless Approach to the Agent Development

Chang-Hyun Jo
Department of Computer Science
University of North Dakota
Grand Forks, ND 58202-9015, USA
jo@cs.und.edu

ABSTRACT

In this paper, we propose a seamless approach to the agent-based software development. We have developed *Agent-based Modeling Technique* (*AMT*) to support a seamless development from modeling to implementation of agent-based software. The idea of AMT is realized by its modeling language – *Agent-based Modeling Language* (*AML*) and programming language – *Agent-based Programming Language* (*APL*). In this paper, we describe our novel idea of seamless approach to develop agent-based software. We illustrate AMT using a case study for agent software development in banking application. This paper also makes clear difference between agent-based programming and agent-oriented programming. This paper mostly emphasizes to describe agent-based modeling technique. Both the implementation details and practices of AML and APL can be found in the separate papers.

Keywords

Agent-based Modeling Technique, Agent-based Programming Language, Agent-based Software Engineering

1. INTRODUCTION

A success in agent programming is based on providing a good development of a feasible modeling methodology and its successful application in the real world. There are a few frontier research works in this area so far, but none of them suggests a seamless development methodology from modeling to implementation via agent concepts. The aim of this paper is to provide an agent modeling technique that provides a consistent guide to develop software based on agents.

Agent-based programming is emerging as a new programming paradigm in the next decades. However, there have been no programming languages to well support agent programming naturally. To make it worse, even before we try to implement agents, there is no proper modeling methodology for agent software development. The existing software development methods such as structured modeling and object-oriented modeling are not well suitable for agent software development because of the difference of notions in different paradigms.

SAC 2001, Las Vegas, NV

This paper describes how to develop seamlessly from the modeling to the development of software in terms of agents[1]. A noble approach to Agent-based Modeling Method (AMM), the supporting Agent-based Programming Language (APL), and a real-world prototyping have been done in separate works [Jo 00].

Some papers present their idea on agent system engineering [DeLoach 99]. Multiagent Systems Engineering (MaSE) [Deloach 99] attempts to answer the Sycara's six challenges [Sycara 98] of multiagent systems. MaSE methodology is based on Rumbaugh's Object Modeling Techniques (OMT) and Unified Modeling Language (UML). MaSE defines four kinds design levels such as Domain Level Design, Agent Level Design, Component Design, and System Design.

Wooldridge et al. [Wooldridge et al. 99] suggested a methodology for agent-oriented analysis and design. Jennings and Wooldridge argue that analyzing, designing and implementing software as a collection of interacting, autonomous agents represents a promising point of departure for agent-oriented software engineering [Jennings and Wooldridge 00].

Our work defines Agent Software Process (ASP) which is an iterative development model and evolutionary prototyping model. ASP has four development phases such as requirement analysis, modeling, construction, and deployment. Each phase has four steps such as analysis, design, build, and test. Each step is precisely defined by activities. Our model brings with Agent-based Modeling Technique (AMT), Agent-based Modeling Language (AML) and Agent-based Programming Language (APL). Our model provides a more precise and seamless development method by using consistent concepts and firm relationships between modeling and languages than that any other agent model suggested. Our Agent-based Modeling Technique (AMT), is different from the previous work in the following point of views:

- AMT is a seamless approach to develop agent-based software from modeling to implementation.
- AMT comes with a modeling language to model agent-based system and an implementation language. AMT can be specified by Agent Modeling Language (AML), and can be implemented by Agent-based Programming Language (APL).

The next section introduces the basic idea in agent-based computing we adopted in this paper. The section 3 presents our novel idea on our Agent-based Modeling Technique (AMT) using some examples to show how to apply AMT to an real-world agent-based software development. This paper concludes with

[1] This work is partially supported by the North Dakota EPSCoR IIP in summer 2000.

summary of this work and some suggestions for the potential future work.

2. BASIC CONCEPTS

An object is a thing that combines the related data and the associated operations on its data. An agent is concurrent, autonomous, intelligent and self-contained object. Self-containing means that an agent describes its behavior by itself through the goal to achieve and behavior to implement the goal based on the current environment.

An autonomous agent is an object that senses the environment, and acts on it based on its own agenda [Petrie 96]. Agents should be intelligently responding to any events triggered on them. Intelligent agents are not any more passive like regular objects, but they are actively responding to any changes in the environment on which they are. Such reactive action is called in different names like "sensing and acting" [Franklin and Graesser 96]. Mobile agents can move from one machine to another to autonomously performing the goal. Agents are cooperative, perceptive, and pro-active [DeLoach 99]. Agents are cooperative through communication among them. Agents are perceptive if they perceive their environment, react on it, and they can also affect their environment. Pro-active agents exhibit goal-directed behavior. Agents are active and concurrent objects. Pro-active purposeful agents are also called as "goal-oriented" which means those agents do not simply act in response to the environment, but act to pursue well-defined goals [Franklin and Graesser 96]. Leaning properties of agents are also called socially able [Franklin and Graesser 96]. Learning agents are adaptive to the future environment by changing their behavior based on their previous experience [Franklin and Graesser 96] and inference from the existing knowledge. Franklin and Graesser [Franklin and Graesser96] well compare the difference of agents with just programs.

Some agent-based systems imitate reasoning behavior according to the theoretical model of artificial intelligence – Belief Desire Intention (BDI) model [Bratman 87]. Agents have explicit goals to achieve or events to handle (desires). A set of plans (intentions) is used to describe how agents achieve their goals. Each plan describes how to achieve a goal under varying environments (belief). A set of data called belief describes the state of the environment. In our work we suggest, however, there are two kinds of environments: local and global. Dynamic adaptation within environments is important factor of agents' behaviors.

We are referring our system here to the first-order intentional system, which has beliefs and desires, but no beliefs and desires about beliefs and desires [Woodridge and Jennings 95].

In this paper we would like to differentiate the *agent-based* system from the *agent-oriented* system. *Agent-based* programming is programming based on agents. A program consists of a set of agents and their collaboration. *Agent-oriented* approach involves agents that learn themselves by experience and adapt themselves based on the previous learning to the current environment. Therefore, the agent-based system has been implemented based on agents, while the agent-oriented system supports learning and adaptation.

There are some research and practices regarding agent-based approach. JAM [Huber 99] is a BDI-based mobile agent architecture. JAM provides rich plan and procedural representations, and utility-based reasoning over multiple simultaneous goals. JAM also provides an agentGo primitive function utilizing Java's object serialization to support mobility. JACK is an agent-oriented development environment by offering some extensions to implement agent behavior on the Java [JACK 99]. FarGo [Holder et al. 99] is an extension of Java to provide a model to develop distributed applications.

Some programming languages [Thomsen 96] such as Actor [Agha 92], Telescript [Telescript 95], AgentTCL [Gray 95], and Parallel-C++ [Jo 91] provide mobility with an object or process level. Knowledge Query and Manipulation Language (KQML) [Finin et al. 94] is a language and protocol to exchange information and kowledge.

Our language, Agent-based Programming Language (APL), is designed to provide a simple and readable construction of agent-based programs while keeping the concept based on the agent-BDI model. The APL constructs are well matched to the concept of the agent-BDI model. Therefore, programmers can easily mapped the models constructed by agent-based modeling into the agent-based programs more naturally than other languages suggested so far. Agent-based Modeling Language (AML) is to be used to specify models that are constructed at the modeling phases. Both AML and APL are described in a separate paper [Jo 00].

This paper describes our novel Agent-based Modeling Technique (AMT) which provides a seamless approach to develop agent-based software through the phases of analysis, design, and implementation. AML is used to model the agent-based system at the stage of analysis and design, and APL is used to implement the agent-based application based on the models constructed by using AML. AMT defines models, steps, and activities to develop the agent-based system by using AML and APL.

3. AGENT-BASED MODELING TECHNIQUE (AMT)

Agent-based Modeling Technique (AMT) is a technique to provide a framework from which software engineers can design agent-based systems systematically.

The Agent-based Process Model (APM) defines what kind of things should be done, who are involved in, and which tasks each participant should do. The model is an abstraction of the real world problem we are tackling. There are two kinds of models – analysis model and design model. Techniques are methods and tools to build models in the defined process.

3.1 Agent-Based Modeling Technique (AMT)

Decomposition is to decompose a complex problem into relatively small and manageable components. Decomposition level in structured programming is functions and processes. Decomposition level in object-oriented programming is objects. Decomposition level in agent-based computing is:

- Agents to achieve independent goals
- Agents to achieve the same goal independently
- Agents to achieve the common goal cooperatively

However, dynamic interactions among agents are unpredictable at design time and compile time, as Jennings and Wooldridge [00] pointed out. A system based on agents is inherently unpredictable,

because the system may not predict all possible interactions between agents. To make it worse in design and modeling based on agents, but to make it better in AI practices, the knowledge-base for agents are dynamically growing at runtime. For example, agents are learning by experience at runtime, and their local/global environments are growing and changing.

A *self-reflective agent system* is a typical example for that case. A reflective system is modifiable at runtime based on current experience. Local and global environments have been evolving. It modifies its intentions to be adapted into the new environment that has been changed at runtime. Furthermore, to adapt the new environment and needs, its desire may be changed too. If its desire is changed, its intentions should be changed to according to the new goal to achieve. It is highly recursively reflective system.

Therefore agent-based system should be designed in the highly flexible manner. Agents should be independent and autonomous. Agents may be cooperative. However their cooperation should be designed in the highly flexible manner. This is one of the important differences between agent-based design and object-oriented design.

A requirement for an agent-based system can be described in both static requirement and runtime requirement. Static requirement can be extracted from the requirement specification statically defined. Runtime requirement can be specified by assumption of runtime behavior, and the different behavior of the system should be assumed and designed in that way.

3.2 An Example Application

In this section, we practice the AMT with a real-world problem – a banking application while we explain AMT. Assume we have a banking application to be developed by the agent-based approach, which is described by the following statement:

A banking application is simulated. A customer may ask to create accounts, deposit both checking and saving accounts, delete accounts, and report transactions. The bank has other departments taking care of marketing, credits, and payments.

3.3 Analysis

In the requirement analysis, we gather the customer's requirements from the overview statements (user requirement specification) and interview, and identify the goal, users and system behavior through the analysis models.

The analysis phase includes the following steps:

- Analyze the system requirements

  Analyze the system requirements by the requirements specification and interviews, and construct Belief-Desire-Intention (BDI) cards. A BDI card summarizes the requirements of an agent system and its BDI summary. The following figure shows an example of system requirement analysis [Figure 1].

Agent	Deposit
Desire	Deposit_Checking Deposit_Saving
General Belief	Banking DB, Personal Info DB, Credit Info DB, Checking DB, Saving DB

Stimuli	Intention	Belief	Response	Collaborator
Plan-1	Create Checking	Checking DB	Checking Creation	Account
Plan-2	Deposit Checking	Checking DB	Checking Deposit	Report
Plan-3	Create Saving	Saving DB	Saving Ac Creation	Account
Plan-4	Deposit Saving	Saving DB	Saving Deposit	Report
…	…	…	…	…

Figure 1. A BDI Card for Deposit in Banking Application

- Identify agents and concepts

First, we find out a system desire, which is called as a global desire (Dg). To find out the global desire, we identify the system belief, which is called as a global belief (Bg). We also identify a system intention, which is called as a global intention (Ig). Next, we decompose the global desire into several sub-level desires (Di) to achieve with the agent system until we refine the goals of the agent system we want to build. For each desire, we can identify belief (Bi) and also intention (Ii) for it. We call this a series of desires as *desire lists* [Figure 2]. Through this step, *agent decomposition* [Figure 3] is succeeded.

D_g: global desire
D_1, B_1, I_1
D_2, B_2, I_2
:

Figure 2. Desire Lists

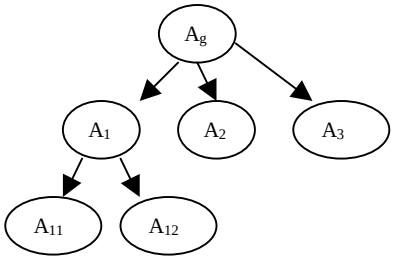

Figure 3. Agent Decomposition

Here we identify the agents we found the system requirement analysis. The agents can be extracted from both agents and collaborators on the BDI cards. The followings are the agents we identified.

Customer, Bank, Deposit, Accounts, Reporting, Marketing, Credits, Service, …

- Identify relationships among agents

  Next, we have to figure out the relationships among the agents identified [Figure 4]. Relationships can be extracted the information shown on the BDI cards we constructed previously. Especially, collaborator information is useful to draw initiator, creator, and collaborator information.

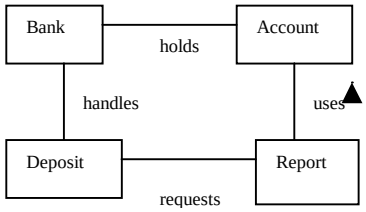

Figure 4. A Relationship Diagram

- Build scenarios

  A scenario describes a process briefly in a few of plain sentences. Each scenario can describe a process of an agent, all the participating agents, or partial behavior of an agent. For each stimuli of an agent, one or more scenarios can be defined. A scenario describes what happened in the agent system by describing a sequence of steps to be performed by each participating agent. In agent-based system, a sequence of processes at the plan level is hidden by goal to achieve. Therefore, in the agent-based system, the scenario describes a client agent, a server agent, and the corresponding current goal to achieve. A description describes briefly the property of this scenario. Here is an example of a scenario for checking deposit [Figure 5].

Deposit a checking account or saving account		
Client	Server	Goal/Plan
Customer	Bank/Deposit	Open_Checking
Customer	Bank/Deposit	Deposit_Checking

Figure 5. A Deposit Scenario

- Identify agent boundary

One of the most important things in analysis is to identify the *agent boundary* [Figure 6]. Agent boundary draws the limits of a set of agents. An agent can be within an agent boundary. However, a set of agents can reside within an agent boundary. Agent boundary can be used as a level of implementation packages in the design and implementation phases.

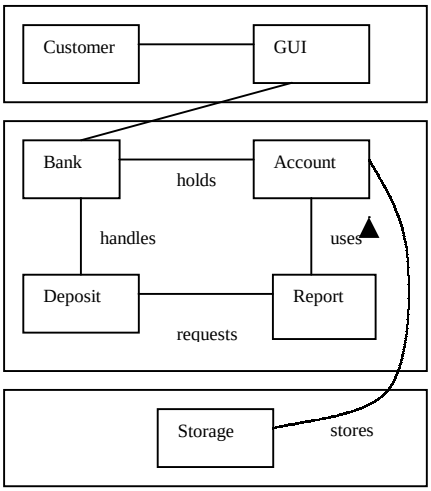

Figure 6. Agent Boundary

3.4 Design

In the design phase, we describe the system more in detail from the models constructed in the analysis phase.

In the design phase, we construct the following models for design.

- Relationship diagrams

  Relationship diagrams show the relationship among agents [Figure 7]. It shows several kinds of relationships such as inheritance, dependency, visibility and logically and physically structured organization. Relationship diagrams built at the analysis phase are refined precisely on consideration of implementation.

  Components for agents in the relationship diagrams include agent's name, functions, belief, desire, and intention. Components for agents in the relationship diagrams can be separately and precisely described in detail in the agent diagrams.

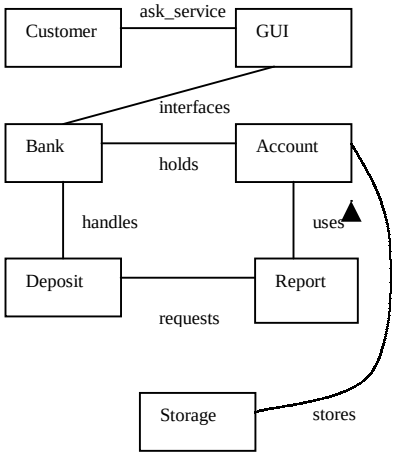

Figure 7. Relationship Diagrams

- Interaction diagrams

  Interaction diagrams show the interactions among several agents. Interaction diagrams are in the different kinds of levels such as local, partial and global. Interaction diagrams are related to the scenario. Interactions shown on a scenario are described on one or several interaction diagrams.

  From the BDI cards some desires with their agents can be extracted. For example [Figure 8], suppose we extract an agent "Deposit" with the desire "DepositChecking". A client agent (e.g., Bank) can interact with the server agent "Deposit" through this current goal (e.g., DepositChecking). The desire "DepositChecking" can also initiate any interaction in the agent.

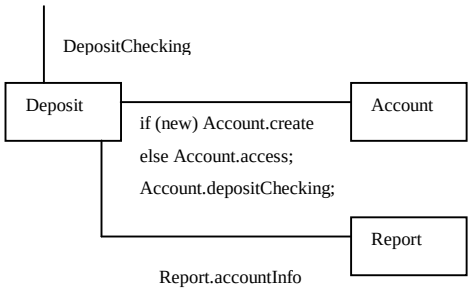

Figure 8. Interaction Diagram: DepositChecking

- Patterns

  While a system is designed via interaction diagrams, some patterns can be used to aid the system design. A pattern describes a problem with a typical solution on the problem, in such a way, we can apply this solution to the problem successfully. Even though there are not many design patterns available for agent computing, but designers can start it with some design patterns for object-oriented computing [Gamma et al. 95] [Aridor 98]. Patterns may help designers to decide to which agents some specific tasks must be assigned.

  For example, in the previous figure, suppose we would like to create or deposit a checking account. To whom we assign this goal to achieve? The agent "Account" has the information of all the accounts. Therefore, the agent "Account" is the most appropriate agent to create and update the account as an information expert [Larman 98], which has the information the agent needs. Therefore, we can assign the goal "create" to the agent "Account". This goal is invoked from the agent "Deposit". This kind of information can also be extracted from the design models we constructed in the previous phases.

- Component diagrams

  Component diagrams [Figure 9] show useful information of packaging with the related agent components when coding. Component diagrams indicates the implementation group as packages in most programming languages. The agent boundary information we extracted at the previous analysis phase can be used to get boundaries of different components.

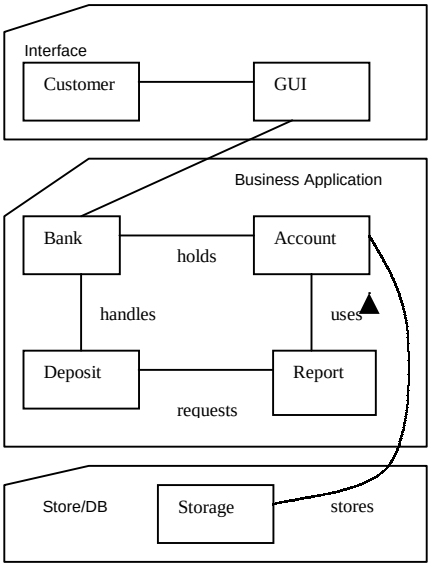

Figure 9. Component Diagrams

3.5 Implementation of Models

The next phase is implementation of the models constructed in the previous phases. One to one mapping from the models to the codes may not be possible. Ideally, there is no component in the code, which is not identified from the models. However, if we find new components in the following phases, the models in the previous phases should be synchronized in each iterative phase.

Models built from the previous phases are used to implement the codes for the agent application we like to build. Actually coding is ideally one-to-one mapping from the design models to the codes.

The programming language used in this example is Agent Programming Language (APL) which has been developed by the author [Jo 00]. The example models built above can be programmed in the following ways. First of all, the necessary agents and their BDIs are defined. For example, the agents and BDIs for the agent "Deposit" can be defined in the following manner [Figure 10]. The main body of this agent is automatically executed when this agent is created.

```
agent Deposit extends BDI {
  belief Deposit_B;
  desire Deposit_D;
  intention Deposit_I;
  main() { … } // main body to execute on this agent
}
```

Figure 10. Agent Definition for Deposit Agent

The information which agents and BDIs should be made is from the design models we constructed in both analysis and design phases. For example, the information we need to code an agent named "Deposit" and its corresponding BDI definitions is from many models such as the Deposit BDI card, the agent diagrams, the relationship diagrams, and scenarios.

The following figure [Figure 11] shows a belief definition for the agent "Deposit". There are some belief declarations such as "CheckingDB" and "SavingDB". Some definitions for accessors and modifiers for some beliefs are also provided. Some goal completion flags can be defined in the set of belief. These goal completion flags are set to represent the completions of goals. Accessors to these goal completion flags are also necessary to define.

```
belief Deposit_B extends BDI {
  CheckingDB;
  SavingDB;
  :    // some other belief declarations
  :   // also goal completion flags defined
  Accessor4CheckingDB();  // accessor for belief
  Modifier4CheckingDB();  // modifier for belief
  :
}
```

Figure 11. Belief Definition for Deposit Agent

The following figure [Figure 12] shows a desire definition for the agent "Desire". There are two goals to achieve, "DepositChecking" and "DepositSaving". There may be some other kinds of desires such as pre-condition, continuous-condition, and post-condition desires. The pre-condition desire is used to check or set the pre-condition before the current goal is set. The continuous-condition is used to check the continuity of the current goal. The post-condition is to check the post-condition after the goal accomplished. These conditions are very useful to know whether the previous services are completed not only when concurrent and collaborative multi-agents are cooperating, but also even when a single threaded agent is running.

```
desire Deposit_D extends BDI {
  DepositChecking() { … }
  DepositSaving() { … }
  :
}
```

Figure 12. Desire Definition for Deposit Agent

The intention definition [Figure 13] for the agent "Desire" is also needed to provide. The intention definition includes several plans to achieve some goals defined in the desire definitions. A plan or a set of plans is used to achieve a goal.

```
intention Deposit_I extends BDI {
  createAccount();
  accessAccount();
  depositChecking();
  depositSaving();
  :
}
```

Figure 13. Intention Definition for Deposit Agent

From the interaction diagram for the "Deposit" agent, some codes can be produced. We can more precisely code the plan "depositChecking()" in the intention definition for the agent "Deposit" like the following [Figure 14].

```
intention Deposit_I extends BDI {
  createAccount();
  accessAccount();
  depositChecking() {    // refined plan
    :
    if (Account is new) checkingAcc = Account.create;
    else checkingAcc = Account.access;
    checkingAcc.depositChecking(amount);
    :
    reportAg.accountInfo(checkingAcc);
    :
  }
  depositSaving();
  :
}
```

Figure 14. Refined Plan in the Intention Definition for Deposit

The packaging information can be extracted from the component diagrams. In our example we can defined three packages, such as "UserInterface", "BusinessApplication", and "StoreDB" [Figure 15].

```
package UserInterface;
    agent Customer;
    :    // BDIs for this agent
    agent GUI;
    :
package BusinessApplication;
    agent Bank;
    agent Account;
    agent Deposit;  // the exemplified agent
    agent Report;
    :
package StoreDB;
    agent StorageDB;
    :
```

Figure 15. Package Definitions for the Banking Agent System

4. CONCLUSION

This paper introduces our novel idea on agent-based modeling techniques with the useful associated schemes such as modeling language and programming languages, which are mostly useful in agent-based software engineering.

The agent-based modeling technique, ATM, we suggested here shows a possibility of a seamless approach from the agent-based modeling to the programming codes that are supposed to run. The agent-based programming language, APL, can be used to implement the models constructed through the modeling phases.

Potential research related to agent-based computing may also include various development processes, software metrics, testing techniques, and the associated tools. Our work will extend to develop such things while we need to refine our work suggested here.

5. ACKNOWLEDGMENTS

The author thanks his research assistants, Allen J. Arnold and Xin Feng, for helping him to implement this idea.

6. REFERENCES

[1] Agha, G., Mason I. A., Smith, S., Talcott, C., Towards a Theory of Actor Computation, The Third International Conference on Concurrency Theory (CONCUR '92), R. Cleaveland (Ed.), LNCS 630, 565-579, Springer-Verlag, 1992

[2] Aridor, Y. and Lange, D. B., Agent Design Patterns: Elements of Agent Application Design, Autonomous Agents '98, Minneapolis, MN, USA, 108-115, 1998.

[3] Bratman, Michael E., Intention, Plans, and Practical Reason, Harvard Univ. Press, 1987. (also by CSLI Publication, 1999)

[4] DeLoach, Scott A. Multiagent Systems Engineering: A Methodology and Languages for Designing Agent Systems, http://en.afit.af.mil/ai/publications/Conference/aois-99/MaSE-AOIS99.htm, 1999.

[5] Finin, T. and Fzitzson, R., KQML as an Agent Communication Language, CIKM'94, Gaitherburg, MD, 456-463, ACM, 1994.

[6] Franklin, Stan and Graesser, Art. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, http://www.msci.memphis.edu/~franklin/AgentProg.html, Also in the Proc. of the 3rd International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[8] Gray, R. S., Agent Tcl: A transportable agent system, http://agent.cs.dartmouth.edu/papers/gray:agenttcl.pdf, Dec. 1995.

[9] Holder, O., Ben-Shaul, I., and Gazit, H., Dynamic Layout of Distributed Applications in FarGo, ICSE'99, Los Angeles, 163-173, ACM, 1999.

[10] Huber, Marcus J. JAM: A BDI-theoretic Mobile Agent Architecture, Proc. of the Autonomous Agents '99, Seattle, USA, 236-243, 1999.

[11] JACK Intelligent Agents user Guide, Agent Oriented Software Pty. Ltd., http://www.agent-software.com.au, 1999.

[12] Jennings, N. R. and Wooldridge, M., Agent-Oriented Software Engineering, J. Bradshaw (ed.), Handbook of Agent Technology, AAAI/MIT Press, 2000.

[13] Jo, Chang-Hyun and George, K.M. Language concepts using dynamic and distributed objects. Proceeding of the ACM 1991 Computer Science Conference (ACM/CSC '91), San Antonio, Texas, (March 5-7, 1991), 211-220, ACM Press (1991).

[14] Jo, Chang-Hyun, Agent-based Modeling Technique (AMT), Agent-based Modeling Language (AML), Agent-based Programming Language (APL), and their Applications, a working paper, 2000.

[15] Larman, Craig., Applying UML and Patterns, Prentice-Hall, 1998.

[16] Petrie, Charles J. Agent-Based Engineering, the Web, and Intelligence, http://cdr.stanford.edu/NextLink/Expert.html, Also appeared in the IEEE Expert, (December 1996).

[17] Sycara, K. P. Multiagent Systems, AI Magazine, 19(2), 79-92, 1998. (Re-referenced from [DeLoach 99]).

[18] The Telescript Language Reference, http://web.yl.is.s.u-tokyo.ac.jp/~masatomo/mobile/Telescript/telescript.html , 1995.

[19] Thomsen, B. Programming Languages, Analysis Tools and Concurrency Theory, http://www.acm.org/pubs, ACM Computing Surveys 28A(4), December 1996.

[20] Wooldridge, M. and Jennings, N. R., Intelligent Agents: Theory and Practice, Knowledge Engineering Review, 10(2), Cambridge Univ. Press, Also at http://www.elec.gmw.ac.uk/dai/pubs/KER95/, June 1995.

[21] Wooldridge, M., Jennings, N. R., and Kinny, D., A Methodology for Agent-Oriented Analysis and Design, Autonomous Agents '99, Seattle, WA, USA, 69-76, 1999.