ABSTRACT

KONG, JIEJUN. Constructing multiagent systems: the MEP project.. (Under the direction of Dr. Munindar P. Singh.)

The purpose of the research was to develop a prototype educational system that emphasizes the autonomy of the participants. Newly proposed as an alternative approach for many distributed applications, multiagent systems emphasize openness, autonomy, and flexibility. Our computational modeling of agents enforces such properties by employing high-level abstractions of interaction among agents, offering users opportunities to flexibly make decisions, and exploiting portable software packages for agent construction.

The multiagent system is named MEP for *Multiagent Educational Program.* It is implemented using a distributed coordination scheduler package previously developed at NCSU, IBM's JKQML package for interagent communication, and IBM's Agent Building Environment.

CONSTRUCTING MULTIAGENT SYSTEMS: THE MEP PROJECT

by

JIEJUN KONG

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER SCIENCE

Raleigh

1998

APPROVED BY:

Dr. Kuo-Chung Tai

Dr. Manuel Aparicio

Dr. Munindar P. Singh
Chair of Advisory Committee

To my family

for supporting me both physically and spiritually all these years

BIOGRAPHY

Jiejun Kong is a graduate student in Computer Science Department, North Carolina State University. In 1991, he got his Bachelor's Degree in Shanghai Jiaotong University (Shanghai, People's Republic of China) with major in Computer Science. In his graduate study, Kong focuses on multiagent systems, with special interest in intertask dependency.

ACKNOWLEDGEMENTS

During the development of this thesis, I received help from many people. First, I wish to express my gratitude to my academic advisor, Dr. Munindar P. Singh who led me into the field of multiagent systems, gave me guidance and inspiration when I was working on the MEP project, made corrections to my academic writing with great patience. Special thanks to Dr. Kuo-Chung Tai and Dr. Manuel Aparicio who kindly agreed to serve on my thesis defense committee.

I would like to thank other students in the NC State University Database Laboratory: Chubin Lin, who implemented the distributed coordination scheduler; Yu Lei, who helped me in learning and implementing the agent communication language; Mahadevan Venkatraman, who helped me in installing ABE and pertinent software on my system; Feng Wan, who gave me lots of good advice as a roommate; Sudhir Rustogi, who gave me good direction in preparing the thesis defense; Anuj Jain, who is an expert of Java and NT station. It is really my pleasure to work with them.

I would also like to thank Diheng Qu, who shared with me his experience in preparing thesis and defense. This saved me a lot of time and trouble.

Finally and most importantly, I want to thank my whole family, without whose love and support I would not have been able to complete my graduate study and this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As distributed computing environments are becoming prevalent, the paradigm of computation is changing. The essence of the new paradigm is to achieve computation by interaction among computing entities physically or at least virtually located in separated units. Unlike the traditional paradigm where computation is done by a centralized processing unit, the new paradigm should respect or at least consider idiosyncrasies of different processing units. Therefore, flexibility, agility, and openness are among the most important properties of software systems designed for the new information environments. Multiagent systems, where different processing units are abstracted as agents, apply naturally in such settings by the virtue of enhancing flexibility and autonomy of component computations.

Computing education is one of the most important topics today. Some of the current difficulties of computing education can be naturally addressed in a distance setting, while reaching out to working people and residents of remote areas—traditionally underrepresented in education. As the teacher and the students are physically distributed rather than being placed in a closed classroom, the assumption that imperative responses will be received immediately does not necessarily hold. We need to offer various users flexibility in such an open setting so that many of them could autonomously adjust their own schedules without disturbing other parties. Now there are many online distance education programs based on the client-server paradigm. It is obvious that interaction is simplex rather than duplex in such a paradigm, although clients may have cherished flexibility. Even though people have proposed some makeup solutions like "server pushing and client pulling" technology, data persistency and generic modelling of such solutions are still open problems. Therefore, we try to use a different paradigm to achieve flexibility, autonomy, and appropriate modelling in open systems. The Multiagent Educational Program (MEP) is designed and implemented to demonstrate our ideas. The essence of our efforts is applying techniques from the cooperative information systems.

1.1 Multiagent Approach

As the name indicates, our system is agent-based. Our abstraction of an agent is a label transition system (LTS) which will be discussed in Chapter 4.2. Sometimes people use finite state machine, a subset of LTS with required terminal states, as an alternative. However, the superset is more appropriate in the situation due to the facts that many available server applications do not have terminal states at all. Because it is valid to treat every modern computer as a state transition system where each instruction execution is a transition, the process algebra research community also applies LTS to their abstraction of a computation.

In every LTS, there may exist some significant events which have temporal restrictions towards other significant events in other systems. We also capture such restrictions among different agents as intertask dependencies which will be formalized in Chapters 2.2 and 4.4. Our approach emphasizes autonomy, openness, and flexibility. Such emphases are mainly achieved by our temporal guard scheduling paradigm. In our paradigm, each agent is equipped with a temporal guard scheduler, which knows all relevant intertask dependencies and tries to enforce the dependencies when the agents try to attempt relevant significant events. To successfully enforce intertask dependencies, when relevant significant events are going to happen, it is obligatory for every agent to seek approval from the scheduler and respect the scheduler's decision. Each agent achieves autonomy by making its own decisions except that it must abide by the scheduler's decisions. We believe such autonomy is the most that current applications may achieve. Intertask dependencies define and only define those non-autonomous computations of an agent, thus least set of constraints is applied and maximal level of autonomy is achieved.

Obviously communication is required among agents. Agents in MEP talk to each other via an agent communication language (ACL) that is general enough to handle all kinds of communication among agents. Most ACLs are based on speech-act theory that offers possibilities of applying natural language conversation for agent interaction. The design of presently available ACLs also facilitates regular data transfer among agents. The ACL constructs the communication infrastructure of MEP.

1.2 Relevant Literature

The most relevant previous techniques for developing and verifying multiagent systems are either not formal, or are designed for traditional distributed systems, or do not fully exploit the modularity inherent in multiagent systems. For example, [Kuwabara, 1996] gives a powerful programming environment, but no formal semantics. By contrast, [Kick, 1995] has a theory, but no associated implementation.

Just like [Kuwabara, 1996], [Barbuceanu & Fox, 1995] also offers a powerful programming environment upon a symbol expression based language named the Coordination Language (COOL). A finite state machine is used as abstraction of each agent. We see that sometimes an agent does not need terminal states at all, especially with respect to presently prevalent server applications. Therefore, we think that an LTS should be a better option. The agents of [Barbuceanu & Fox, 1995] also use an agent communication language. [Ayala & Yano, 1997] and [Ayala & Yano, 1998] proposed a system, Computer Supported Collaborative Learning (CSCL) environments, which illustrates the significance of an ACL. In CSCL agents, user interface actions are classified according to their ACL's communicative act type. Thus a user must know at least some details of ACL. Fortunately, learning an ACL is pretty intuitive since it is related to a restricted and simplified form of natural language. The CSCL system is different from our work in the sense that their research is directed more toward distributed artificial intelligence techniques than toward giving an executable model of workflow.

The active database research community also relates to our research in the sense that they use Event-Condition-Action (ECA) rules as a basic structure. In MEP, we achieve the same effects of ECA rules by separating control flow and data flow. The control flow enforced by a temporal guard scheduler plays the role of event-condition pair. Then the data flow plays the role of action upon receiving permission from the control flow. More details will be offered later in the thesis.

1.3 Organization

This thesis presents our theory and implementation of MEP in the context of cooperative information systems research. Chapters 2 will illustrate theoretical background of our approach. Chapter 4 will discuss the new contributions of MEP in designing multiagent systems. Chapter 5 will discuss the detail design and implementation of the MEP system.

Chapter 2

Background

This chapter presents the important conceptual background to appreciate our approach. Relevant topics are modelling of agents, formal specification of temporal guard system, and theoretical background of agent communication.

2.1 Interactive Machines and I/O Automata

Traditional approaches, such as [Kuwabara, 1996] and [Barbuceanu & Fox, 1995], use an agent interaction model based on communicating finite state machines. The model involves:

1. a collection of finite state machines (FSM), each of them is bound with an agent; and

2. a collection of channels for inter-FSM communication. Each transition of the FSMs is associated with a message send/receive operation of buffered channels.

Messages are tagged so that the FSMs can decide which transition to fire based on the tag. Such a tag could either be arbitrarily defined as in [Kuwabara, 1996] or follow an agent communication language standard as in [Barbuceanu & Fox, 1995], which used KQML performatives.

In our work, FSM modelling is replaced by labeled transition system (LTS) modelling. Although an LTS is not more powerful than a FSM in terms of computation, it is a better choice to depict some presently prevalent systems such as Internet servers, because it does not require a set of terminating states.

Peter Wegner has argued that such machines, which he named as interactive machines, are more powerful than Turing Machines [Wegner, 1998] [Wegner, 1997]. However, since the definitions involving computation and communication are still not settled, those claims are still tentative. From theory of computation, we know at least that such machines are not less powerful than Turing Machines. In a nutshell, the system we model and build is not based on a trivial notion of computation.

2.2 Logic

As the foundation of all formal methods, logic is extensively used in formalization of agent technology. In this chapter, two logic systems relevant to the MEP program, propositional and temporal logics, will be addressed.

A logic system mainly consists of three parts:

1. *well-formed formula*s which are the valid statements permitted by the logic system. A well-formed formula is constructed from atomic units, i.e. *term*s. A formal *language* is the set of all well-formed formulas.

2. *proof theory* which includes the axioms and rules of inference. Axioms give the start point of deductions. Rules of inference state entailment relationships among well-formed formulas.

3. *model theory* which gives the formal meaning of the well-formed formulas.

The language and proof-theory are called the *syntax* since they specify the syntactical part and syntactical evolutions. The model-theory is called the *semantics* since it specifies the meaning of the syntax. Examples will be offered in the following chapters dealing with specific logic systems.

2.2.1 Propositional Logic

Intuitively we can express the *fact*s about the world by a sentence in English or any other natural languages. For example, "If it rains, then the road is wet". In propositional logic such expressions are formalized as *proposition*s and formulas in propositional logic are built up from the propositions. For example, the formula "**rain→wet-road**" includes two propositions **rain** and **wet-road** that correspond to the two facts, respectively.

Formally the language of propositional logic consists of the following symbols.

1. infinitely many *variable*s, which represent atomic facts;
2. the *connective*s: *negation* $\neg$, *conjunction* $\wedge$, *disjunction* $\vee$, *implication* $\rightarrow$, *equivalence* $\leftrightarrow$; these correspond to **not**, **and**, **or**, **if-then**, and **equivalent-to** relations in natural language.
3. parentheses, which are used only as delimiters.

Some sequences of above symbols, such as "$\neg\wedge$" or "**rain wet-road** $\rightarrow$", are meaningless. We must apply some restrictions so that only well-formed formulas can be produced.

DEFINITION **1** The collection of *formula*s is the smallest set satisfying:

1. all variables are formulas;
2. if ϕ is a formula, then $\neg\phi$ is also a formula;
3. if ϕ and ψ are formulas, then $(\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$ are also formulas.
4. if ϕ_0 and ϕ_{n-1} are formulas, then $(\phi_0 \wedge \phi_1 \wedge ... \wedge \phi_{n-1})$ and $(\phi_0 \vee \phi_1 \vee ... \vee \phi_{n-1})$ are all formulas.

ϕ	ψ	$\neg\phi$	$\phi \wedge \psi$	$\phi \vee \psi$	$\phi \rightarrow \psi$	$\phi \leftrightarrow \psi$
t	t	f	t	t	t	t
t	f	f	f	t	f	f
f	t	t	f	t	t	f
f	f	t	f	f	t	t

Table 2.1: Truth tables for connectives

When $n = 0$, the above two sorts of formula degenerate into the empty conjunction which is denoted by $\top$ (*verum*) or the empty disjunction which is denoted by $\bot$ (*falsum*).

By the above restrictions, only well-formed formulas such as "**rain→wet-road**" can be formed.

The model theory of propositional logic requires that every formula must have a truth value which is either *true* or *false*. This is enforced by

- Every connective has a *truth table* that explains how a compound formula's truth value is determined by the truth values of its components. The truth table is shown by Table 2.1 where *true* and *false* are denoted by `t` and `f`, respectively.

- Every variable is assigned a truth value by a *truth assignment* function (also called *interpretation*) γ mapping variables into truth values.

Hence we can determine truth value for any formula of propositional logic under certain truth assignment. For example, suppose it is raining, we assign **rain** as *true*, but the road is not wet, so we assign **wet-road** as *false*, therefore we know that the formula "**rain**$\rightarrow$**wet-road**" is *false* according to the truth table.

Model theory also defines some advanced relations between truth assignment and well-formed formula. For more details about logic, please refer to some books like [Enderton, 1972]. Here I just illustrate some fundamental concepts of logic expressions and semantics that provide the basis of a formalization.

2.2.2 Event Algebra and Intertask Dependencies

The logic used for expressing formulas contains temporal operators, enabling one to describe how a system behaves as time passes rather than simply a characteristic of the system at a particular point in time. Using such a temporal logic one can state properties such as:

- "eventually event a will occur"
- "it is always the case that after an a event the event b eventually occurs"

The coordination specification language of MEP is formalized as an event-based linear temporal logic, which is propositional logic augmented with the *before* $(\cdot)$ temporal operator. More details are available in [Singh, 1997a]. A brief description is offered for completeness as below.

We use $\mathcal{E}$ to denote our specification language of event expressions. The syntax of

$\mathcal{E}$ follows. Ξ includes all event literals (with constant or variable parameters); $\Gamma \subseteq \Xi$ contains only constant literals. A *dependency* $\mathcal{D}$ is an expression in $\mathcal{E}$. A *workflow* $\mathcal{W}$ is a finite set of dependencies.

SYNTAX **1** $\Xi \subseteq \mathcal{E}$

SYNTAX **2** $E_1, E_2 \in \mathcal{E} \Rightarrow E_1 \vee E_2, E_1 \wedge E_2, E_1 \cdot E_2 \in \mathcal{E}$

We assume that $\cdot$ (before) has precedence over $\vee$ and $\wedge$, and $\wedge$ over $\vee$.

The semantics of $\mathcal{E}$ can be elegantly given in an algebraic style as below. Like for many process algebras, our formal semantics is based on traces, i.e. sequences of events. Our universe is $U_{\mathcal{E}}$, which contains all consistent traces involving event instances from Γ. Consistent traces are those in which either an event instance or its complement occurs, but never both, and in which event instances are not repeated.

$$[\![\,]\!] : \mathcal{E} \mapsto \wp(U_{\mathcal{E}})$$

gives the denotation of each member of $\mathcal{E}$. The specifications in $\mathcal{E}$ select the acceptable traces—specifying $\mathcal{E}$ means that the service may accept any trace in $[\![\mathcal{E}]\!]$.

Let constant parameters be written as c_i etc., variables as v_i etc., and either variety as p_i etc. $e[c_1 \ldots c_m]$ means that e occurs appropriately instantiated. $[\![e[c_1 \ldots c_m]]\!]$ consists of all those traces that e occurs.

SEMANTICS **1** $[\![e[c_1 \ldots c_m]]\!] = \{\tau \in U_{\mathcal{E}} : e[c_1 \ldots c_m] \text{ occurs on } \tau\}$

$E(v)$ refers to an expression free in variable v. $E(v ::= c)$ refers to the result of by substituting every occurrence of v by c in $E(v)$. Variable parameters are effectively universally quantified by:

SEMANTICS **2** $[\![E(v)]\!] = \bigcap_{c\in\Gamma}[\![E(v ::= c)]\!]$

$E_1 \vee E_2$ means that either E_1 or E_2 is satisfied. $E_1 \wedge E_2$ means that both E_1 and E_2 are satisfied (in any interleaving). $E_1 \cdot E_2$ means that E_1 is satisfied before E_2 (thus both are satisfied).

SEMANTICS **3** $[\![E_1 \vee E_2]\!] = [\![E_1]\!] \bigcup [\![E_2]\!]$

SEMANTICS **4** $[\![E_1 \wedge E_2]\!] = [\![E_1]\!] \bigcap [\![E_2]\!]$

SEMANTICS **5** $[\![E_1 \cdot E_2]\!] = \{e_1 e_2 \in U_{\mathcal{E}} : e_1 \in [\![E_1]\!] \text{ and } e_2 \in [\![E_2]\!]\}$

EXAMPLE **1** $\overline{e} \vee f$ denotes a set of traces on which either $\overline{e}$ or f occurs. That is, if e occurs on a certain trace, then f must also occur on it. Or in other words, e requires f. We can satisfy this dependency by preventing e or ensuring that f also occurs if e does. ▮

EXAMPLE **2** $e < f$ denotes a set of traces on which e occurs before f if both of them occur. That is, either e or f does not occur, or if both of them occur on a certain trace, then f must follow e. Or in other words, e precedes f. We can satisfy this dependency by preventing either e or f, or ensuring that f occurs after e. ▮

The treatment of complementary events in this approach is novel in some respects and merits additional explanation. $\overline{e}$ refers to the complement of e. Complemented literals are included in Ξ and, therefore, need no separate syntax or semantics rule. Since $[\![\,]\!]$ yields sets of traces, complementation satisfies mutual exclusion, but not mutual exhaustion. Intuitively, $\overline{e}[c_1 \ldots c_m]$ is established only when it is definite that $e[c_1 \ldots c_m]$ will never occur. This is crucial in the correct functioning of our service. The weaker, more common, notion of negation is of course also required for scheduling. However, we do not include it in the specification language, only in the internal temporal language for taking scheduling decisions.

2.3 Agent Communication

Just like people must talk with each other via a common natural language, agents should talk with each other via a common agent communication language (ACL). At the present time, major agent communication languages are derived from speech act theory, which is originated in the philosophy of language and has also been applied in natural language processing.

2.3.1 Speech-Act Theory

The notion of speech acts goes back to the philosophical work of [Austin, 1962], who observed that with language people not only make statements, but also perform actions. An language expression for issuing declaration causes expected results from performing certain actions. For example, when a pastor declares a couple man and wife, he is not reporting on their marital status, but changing it. Such a request or declaration is called a *performative*. Not all language expressions are performatives. For example, "solve" is not a performative, otherwise math students would be a much happier by saying "I hereby solve this problem". Generally verbs that cannot achieve expected results are not performatives.

For autonomous agents, they try to communicate with each other so that certain actions can be performed to achieve expected results. Thus it is not a surprise that agents should be able to communicate with each other via performatives. So speech-act theory has been picked up widely as the basis for agent communication.

Parunak offers an interesting example of how to detect interaction patterns among agents in [Parunak, 1996]. His assumptions include a message classification based upon speech act theory. Figure 2.1 shows the basic communicative acts that an agent can take. The acts consist of two categories: speech-acts and non-speech-acts. It is obvious to see that speech-acts are performatives that attempt to do something while non-speech-acts are just actions.

In Parunak's sample conversation, only the communicative acts showed in Figure 2.1 are exploited to accomplish the analysis. As far as speech act theory is

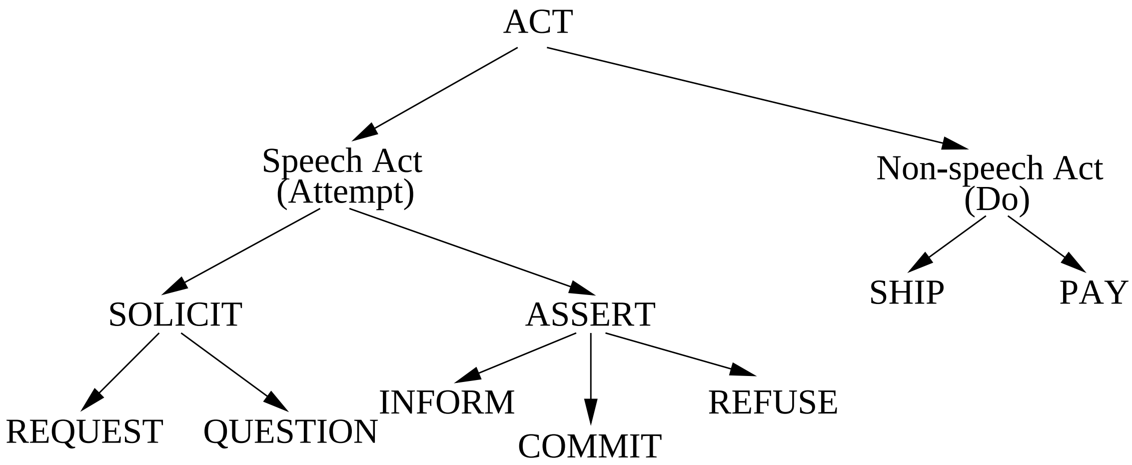

Figure 2.1: Basic types of communicative acts

concerned, the number of communicative act types can vary depending on the complexity of agent communication. In MEP, only several such types are exploited to do agent registration and data flow communication which appropriately meet current complexity requirement of MEP project. More types could be added later so that agents may show their beliefs and intentions as [Ayala & Yano, 1997] and [Ayala & Yano, 1998] have achieved.

2.3.2 KQML and FIPA ACL

The Knowledge-Sharing Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) is an initiative to develop the infrastructure to support the sharing of knowledge among systems.

Based on the speech-act theory, the *External Interfaces* working group came up with a common high-level language which was named as Knowledge Query and Manipulation Language (KQML).

An ontology is a common vocabulary and agreed-upon meanings to describe a subject domain. Agents can successfully communicate only if they can say something others can understand. An ontology gives the common semantics for individual syntactical units. Of course, if such a vocabulary understanding is implied in communication, we only need an implicit ontology instead of an explicit one. For example, a protocol assumed by a client-server application can be thought of such an implicit ontology. In remote procedure call (RPC) protocol, predefined data types such as "int", "string", etc. are instances of common vocabulary between a RPC client and a RPC server.

KQML expressions can be thought of as consisting of three layers as shown in Figure 2.2.

1. The innermost *content layer*: This is actually what traditional communication has delivered. It contains an expression in some language and an ontology which encodes the knowledge to be conveyed. This layer is opaque to KQML in the sense that KQML makes no commitments about this layer. What KQML can do is to deliver without processing the information inside this layer.

2. The middle *message layer*: Here is the essence of KQML. It encodes a message that one application would like to transmit to another application. The message is typed by communication acts classified by speech-act theory. Parameters,

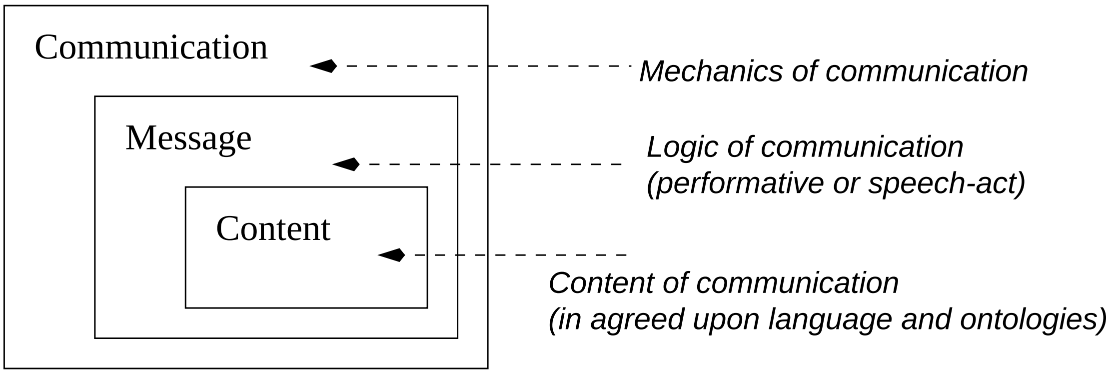

Figure 2.2: Three layers of KQML expressions

along with the contents, are encoded into a message packet, then delivered to the outer layer.

3. The outer *communication layer*: The communication layer is responsible to send the message packet from the message layer. It could be any available means of network communication, such as socket or RPC.

The Foundation for Intelligent Physical Agents (FIPA) is an international non-profit association developing specifications of generic agent technologies to provide a high level of interoperability across applications. FIPA has proposed a standard of agent communication language in [FIPA, 1997a]. An example of FIPA ACL expression is shown in Figure 2.3. Compared to KQML, the FIPA ACL specification comes with a formal semantics so that potential misunderstandings caused by poor defined semantics are less likely to happen. Some of terms are also changed in FIPA ACL specification. For example, "performative" is renamed as "communicative act". An-

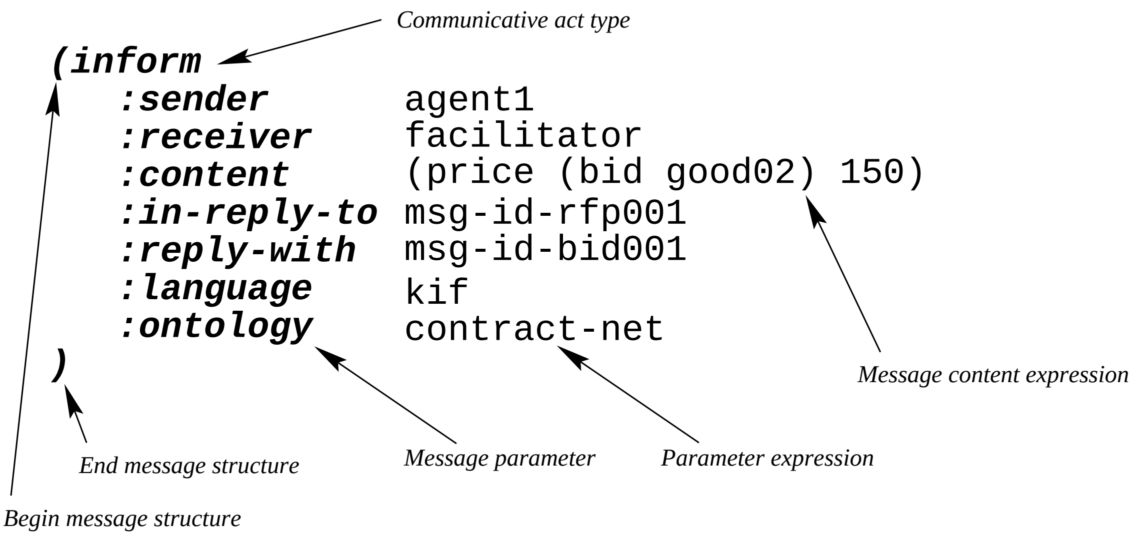

Figure 2.3: Example of FIPA ACL message

other significant change is the redefinition of the set of "communicative act": FIPA removes some acts from the set as non-necessity and adds some as necessity. However, the changes are not important for MEP since we do not use the modified portion.

Chapter 3

Applied Utilities

This chapter briefly introduces two software packages used by MEP—TFM, the temporal guard scheduler and ABE, the rule-based agent building environment.

3.1 TFM: Distributed Temporal Guard Scheduler

TFM is a Java package implementing a distributed temporal guard scheduler using the operational temporal logic described in Appendix A. The architecture and implementation of TFM were described in [Lin, 1998].

In [Attie *et al.*, 1993] intertask dependencies are proposed as a framework to define extended transactional model in multi-system applications. The dependencies are formalized in Computation Tree Logic (CTL) in [Emerson, 1990]. The CTL representation can be quite intricate. [Singh & Tomlinson, 1994] present an approach

	Centralized version	**Distributed version**
scalability	has a limitation of extensibility due to a single server that stores and processes all relevant workflows	has no such single server; every agent only cares about relevant intertask dependents
fault tolerance	if the server crashes, the system collapses	agent crash only effects relevant intertask dependents
communication	has bottleneck at the server	no such bottleneck

Table 3.1: Comparison between a centralized and a distributed scheduler

that specifies the intertask dependencies so that they are more amenable for execution in a distributed manner. [Singh, 1997a] concludes the effort by systematically offering Interaction Oriented Programming (IOP) as the general background.

The model developed in [Singh, 1997a] requires a temporal guard scheduler plugged into local legacy system as a wrapper. The local system sends requests to and receives reply from the scheduler. The major advantage of the scheduler over other approaches is that the scheduler is distributed. The comparison of a centralized and a distributed scheduler is shown in Table 3.1.

What MEP needs from the package arethe following:

- *Temporal guard instance*: When an agent needs to create a temporal guard instance, it must call relevant method functions of TFM.

- *Significant event instance*: When an agent needs to create a significant event instance, it must call relevant method functions of TFM.

- *Guard notification and guard assimilation*: The detail of guard assimilation is discussed in Chapter 4.1. A significant event of MEP must receive intertask guard notification and assimilate the message via TFM package.

- *Monitoring threads*: Monitoring threads are Java threads that detect a guard notification at real time.

- *Intertask dependency instances*: MEP implementation needs two of the intertask dependencies $e \rightarrow f$(require) and $e < f$(precede) implemented by TFM.

- *Java implementation*: Since MEP's major programming platform is also Java, it is simpler to call method functions of the same programming language.

MEP implementation also enhances TFM implementation and the details will be discussed in Chapter 4.6.

3.2 ABE: Agent Building Environment

Agent Building Environment (ABE) is a toolkit developed by IBM for software developers that makes it easy to build an application based on intelligent agents or to add agents to an existing application. ABE is available at [ABE, 1998].

An ABE agent consists of the following components, the terms will be explained below:

- a built-in inference engine called RAISE;

- conduct sets of rules including:
  - sets of rules telling RAISE how to reason;
  - sets of long term facts.
- adapters for relating applications to the agent, starting engine activity, binding fact sets and providing actions that are needed by the agent to carry out its work.

The relation among the components of an agent written in ABE is shown in Figure 3.1.

ABE's design is influenced by expert systems in artificial intelligence research. Rule-based programming plays a major role in such systems. In this programming paradigm, rules are used to represent heuristics, which specify a set of actions to be performed for a given situation. A rule is composed of an *antecedent expression* and a *consequent expression* that are similar to *condition* and *action* in an active database's ECA rule. Procedural programmers may think them as the "if" part and "then" part of an alternative statement. The antecedent expression of a rule is a series of patterns which specify the facts (or data) which cause the rule to be applicable. The process of matching facts to patterns is called pattern matching. The expert system tool provides a mechanism, called the *inference engine*, which automatically matches facts against patterns and then determines which rules are applicable.

Whenever changes are made to facts, the inference engine always performs pattern

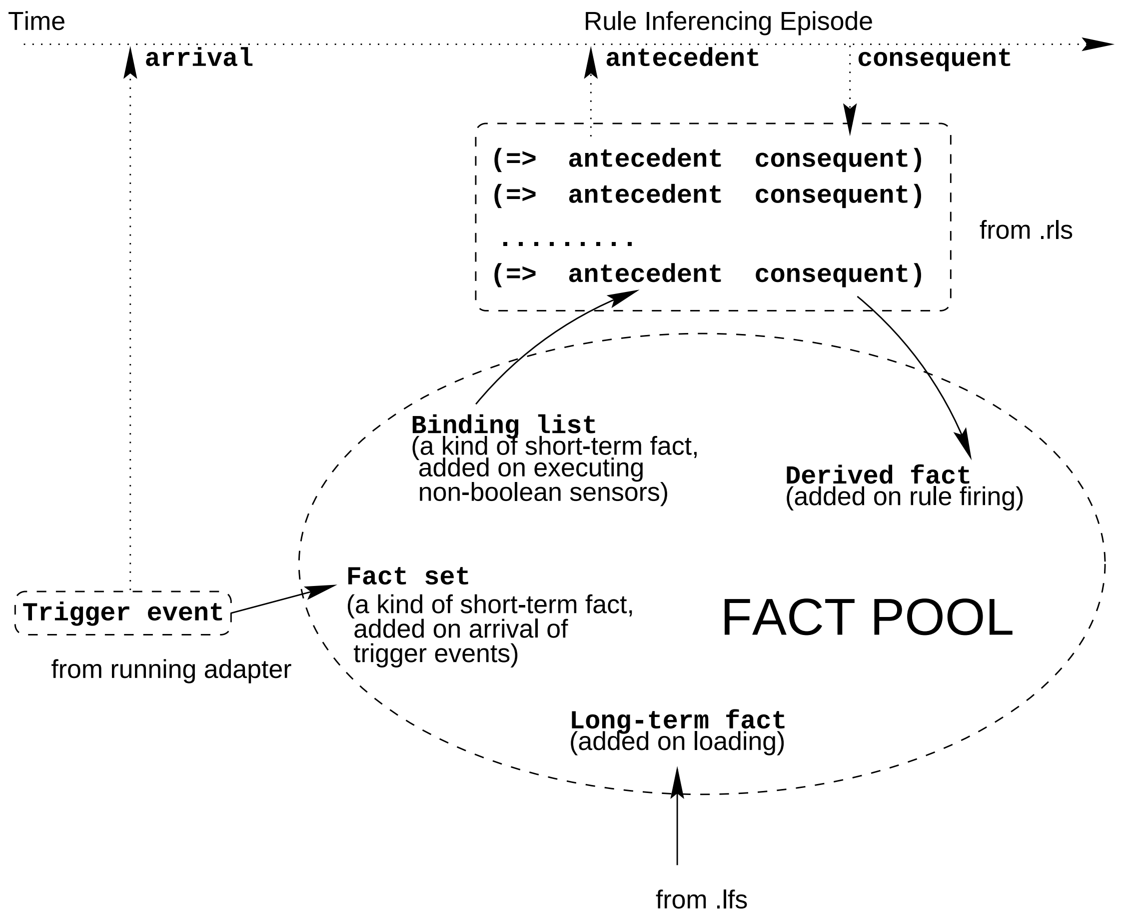

Figure 3.1: RAISE inference episode

matching to make sure that applicable antecedent expressions could be applied. The consequent expression of a rule is the set of actions to be executed when the rule is applicable. The actions of applicable rules are executed when the inference engine is instructed to begin execution. The inference engine selects a rule and then the actions of the selected rule are executed (which may affect the list of applicable rules by adding or removing facts). The inference engine then selects another rule and executes its actions. This process continues until no applicable rules remain.

Inside antecedent and consequent expressions, two kinds of methods need to be mentioned.

1. *When-needed method*: If a value is needed but is not known by the inference engine, a when-needed method for that value activates to obtain it. For example, "If the student agent wants to know the deadline of an assignment, prompt the teacher agent to enter it; if the teacher agent does not respond, use one week as a default."

2. *When-changed method*: If a key value changes, this method activates and performs necessary actions to that change. For example, "If the grader agent has finished grading homework, then he put the grades in the homework database."

In ABE, patterns in antecedent expression are associated with each other by logical $\vee$ and logical $\wedge$ operators, denoted by the strings "AND" and "OR", respectively. When-needed methods are called *sensor*s. When-changed methods are called *effec-*

*tor*s. A sensor that returns boolean result is called boolean sensor, otherwise it is called a non-boolean sensor. Non-boolean sensers accomplish variable binding.

ABE only supports forward chaining which derives all possible facts upon known facts and rules. Backward chaining may be supported in future releases. Facts are classified as long term facts, short term facts, and derived facts. Long facts are always true, thus can be stored in persistent storage. Short term facts are in turn classified as *binding list*s which are produced by a sensor and *fact set*s which are from an event as in an ECA rule. Derived facts are derived temporarily from applied rules in forward chaining.

One of the advantages of ABE is that the IBM research group has supplied multiple powerful programming interfaces for prevalent programming languages including C++ and Java. Users can use such a programming language to implement any sensor and effector in ABE by writing and compiling an adapter file. MEP uses the Java interface to implement some insignificant events in a skeleton.

Chapter 4

Contributions of MEP

MEP implements a distance educational system using the TFM package as a FIPA wrapper agent. MEP can accomplish basic functions for offering a given course over network although the available functions are still primitive due to the restrictions of current complexity requirements. Therefore, operational semantics of agent systems equipped with temporal guarding is realized in the project. Rule-based systems and an agent communication language are also considered and used in MEP's design and implementation. As the project goes on, MEP could be enhanced into a completely rule-based system by converting temporal guards into ECA rules, and an intelligent system by adopting a larger set of performatives in agent communication.

4.1 Operational Semantics

Even though a declarative semantics is clear, concise, and correct, in certain cases there is no way to implement it. For an idea to execute on a computer, we must offer another level of semantics to realize the idea. The required semantics is called *operational semantics* and uses algorithms to implement the expected declarative semantics.

For a practical example, consider one of the students in an operating system course. He had just learned how to use UNIX and X window system and was staying in one of his directory, say "~/dir1", in a window; then sometime later he renamed the directory to "dir2" in another window. Therefore when he tried to tell where he was by typing "pwd" in the first window, he got "~/dir2". Then he was amazed and told everybody that UNIX was so smart!

Nevertheless, when he was told the operational semantics of his operations, he changed his opinion. What happened is that UNIX kernel uses i-nodes rather than file names as the internal representation of files and directories. Therefore once a process changes the file name in an i-node, all other process will see the change. Although the general semantics of UNIX file system is pretty intuitive and straightforward, when something really matters, the operational semantics of the system determines the meaning of user's actions.

For a theoretical example that is more related to our temporal logic case, we may think about the famous logic programming language Prolog. The declarative semantics of Prolog is first-order logic while the operational semantics of Prolog is SLD resolution plus some implementation-defined extensions. To acquire the same power of first-order logic, breadth first search (BFS) is required in searching through resolution tree. However, for reasons of efficiency, Prolog has to use depth first search (DFS) instead which fails to solve some problems that are solvable by BFS. Fortunately, most of the problems in real world still fall in the other class which is solvable by DFS.

While introducing the event-based temporal logic as a formal specification of relationships among workflow dependencies, [Singh, 1996] also offered an operational semantics for it. By implementing MEP, the Multiagent Educational Program, we present a practical program that realizes the operational semantics. The initial chapters of this chapter will discuss the operational semantics in details, but as briefly as possible. The problems of the operational semantics will be discussed in Chapter 6 later.

Generally speaking, the operational semantics use a temporal guard scheduler to prevent invalid traces from happening. The temporal guard scheduler is based on the operational temporal logic mentioned below in Appendix A.

Moreover, the temporal guard system could be also thought as an implementation of the agent communication protocol specified in [FIPA, 1997a]. That is one of the reasons why we classify MEP as an agent-based system besides application of agent communication.

4.2 Labeled Transition System for Agent Skeleton

In MEP, every agent acts in terms of an operational model that focuses on the notions of *state* and *transition*.

1. A *state* of a system is a snapshot taken during its execution.
2. A *transition* represents an *atomic* execution step; it moves a system from one state to another.
   (a) A transition will be labeled with an *action*.
   (b) The action will either be an internal *insignificant event* that is invisible to the environment or a visible *significant event* that represents potential interaction between the system and the environment.

An operational model of a system contains a listing of all possible states (including a start state) the system can be in and the transitions indicating relations among the states.

The above kind of an operational model is termed a *labeled transition system* which is a quadruple $\langle Q, A, \rightarrow, q_S \rangle$ where:

1. Q is the set of *states*;

2. τ is a generic *insignificant event* that is invisible to the outside; $(A - \{\tau\})$ is the set of *significant event*s that are visible to the outside. Obviously A includes all events that are either insignificant or significant.

3. $\rightarrow \subseteq Q \times A \times Q$ is the transition relation (we write $q \xrightarrow{a} q'$ instead of $\langle q, a, q' \rangle \in \rightarrow$); and

4. $q_S \in Q$ is the start state.

Here τ represents a generic insignificant event; since insignificant events cannot affect or be influenced by the environment, we will not distinguish between different kinds of them.

An operational model highlighting the significant events is termed a *skeleton* in [Singh, 1997a]. An intertask dependency assigns a relation between two or more significant events from different skeletons. Since an intratask dependency assigns a relation between two or more significant events from same skeleton, our method could be applied to intratask dependency either even though this area has been well developed by concurrency programming community.

In [Lynch *et al.*, 1994], I/O automaton is offered as a model to depict discrete event systems consisting of concurrently operating components.

The concept *action signatures* captures actions of each automaton. An action signature S is a triple consisting of three pairwise-disjoint sets of actions. The three components of S are written $in(S)$, $out(S)$, and $int(S)$, referring to *input actions*, *output actions*, and *internal actions* respectively. Therefore *external actions* of S is defined as $ext(S) = in(S) \cup out(S)$, *locally controlled actions* of S is defined as $local(S) = int(S) \cup out(S)$. Finally, the union of all the three components is referred to as the *actions* of S, $acts(S) = in(S) \cup out(S) \cup int(S)$.

An *I/O automaton* A is a tuple that consists of four components:

- an action signature $sig(A)$,
- a set $states(A)$ of *states*,
- a nonempty set $start(A) \subseteq states(A)$ of *start states*, and
- a *transition relation* $steps(A) \subseteq states(A) \times class(sig(A)) \times states(A)$, with the property that for every state s' and input action π there is a transition (s', π, s) in $steps(A)$.

Nevertheless, it is insignificant to distinguish input actions and output actions in MEP. Therefore, LTS is a better choice over I/O automata under such circumstances.

To illustrate the above concepts, we may consider the dining philosophers problem. It is a classical problem in concurrent programming. A typical expression of the problem has five philosophers who could be in one of the three states: *thinking*, *hungry*, and *eating*. As they sit around a table, unfortunately there are only one chopstick placed between every two philosophers while every philosopher needs a pair of chopsticks to eat. Therefore, when two neighbors get hungry at the same time or a hungry philosopher has a neighbor in eating, race conditions arise. Many solutions have been proposed to solve the problem. Furthermore, a solution could be improved by considering starvation prevention, maximal parallelism, and fully distributed scheduling. For instance, introducing a new state *starving* may prevent starvation from happening.

Although this problem is concise in expression, it could be applied to a wide range of practical problems. Normally the working pattern of a server application is a combination of alternate working and idling bursts, which correspond to staying in *eating* and *thinking* state, respectively. When race conditions arise between two server applications, mutual exclusion will make one of them stay in bounded waiting status, which corresponds to *hungry* state. A state transition among these states is fired upon dynamic execution of the autonomous philosophers.

In LTS modelling of the problem, for every philosopher $\langle Q_{dp}, A_{dp}, \rightarrow_{dp}, q_S \rangle$,

- Q_{dp} is the set $\{thinking, hungry, eating\}$.
- q_S is *thinking.*
- τ is *appetite_up* and *satiated*, $A_{dp} - \{\tau\}$ is *pickup_chopsticks.*
- $\rightarrow_{dp}$ is $thinking \stackrel{appetite\_up}{\longrightarrow}_{dp} hungry, hungry \stackrel{pickup\_chopsticks}{\longrightarrow}_{dp} eating, eating \stackrel{satiated}{\longrightarrow}_{dp} thinking.$

The behavior of every philosopher satisfy the definition of LTS but not FSM since none of the predefined states is terminal as the states construct an infinite loop.

4.3 FIPA's Concept of Wrapper

Wrappers are small software components that provide an interface to a larger and typically preexisting software system. In [FIPA, 1997b], the concept of *wrapper* is introduced to provide a single generic way for agents to interact with software systems. The temporal guard scheduler we mentioned in Chapter 3.1 could be thought as part of such a generic accessory for every agent. However, [FIPA, 1997b] only offers some examples of its wrapper as an interface between agent communication language and embedded software systems like CGI, Java, and ORB. Our temporal guard scheduler meets the definition of the wrapper, but in the same time is different from those examples in the sense that we actually extend embedded software systems by conferring them temporal logic perception capability rather than just modifying the way they communicate with the outside world.

4.4 Temporal Guards

Given an intertask dependency specified as a temporal logic expression, we have developed a formal method to correctly enforce it. The idea is associating a temporal guard with every significant event to prevent invalid computations from happening. For each computation, the guard captures how far that computation ought to have progressed when the guarded event occurs, and what obligations would remain to realize that computation.

The formalization consists of two parts: the temporal logic used by guard to express temporal activities and derivation of guards for specified intertask dependencies. Appendix A gives a brief description of the formalization. Details are available in [Singh, 1997a].

In TFM, temporal guards are derived manually as a temporal expression string. Such a string is passed to the constructor of a temporal guard Java class to create an instance of the temporal guard. After significant events are instanstiated, certain methods are invoked to associate the instances of relevant temporal guards with the events. Every significant event also has a monitoring thread to detect temporal guard messages from other relevant events and a guard residuation procedure for assimilating received guard messages. The details of design and implementation of TFM are described in [Lin, 1998].

4.4.1 Scheduling with Guards

Execution with temporal guards is straightforward. It involves the following main steps:

1. When a significant event e is attempted, its guard is evaluated. Since guards are updated whenever an event mentioned in them occurs, evaluation usually means checking if the guard evaluates to $\top$. If e's guard is

   - $\top$: e's execution is permitted;

- $\bot$: e's execution is rejected. If the software system continues to execute e, a relevant intertask dependency may be violated.

- otherwise, e is made to wait until e's guard is evaluated to $\top$ or $\bot$.

2. Whenever an event e occurs, a notification $\Box e$ is sent to every event f who has intertask dependency with e. Then f's guard may get chances to re-evaluate if f is waiting.

4.4.2 Invoking the Temporal Guard Scheduler

MEP's significant event scheduler is based upon the TFM software package designed and implemented in [Lin, 1998]. The TFM package implements temporal guard message notification, temporal guard assimilation, and significant event instantiation.

To create an intertask dependency, a programmer has to define all relevant significant events in the dependency and do the following for every significant event:

1. instantiate temporal guards using their manually derived temporal expression strings.

2. instantiate both the event itself and its complement.

3. install appropriate temporal guards for the event itself and its complement.

4. insert relevant events into an event's notification list, so that an event can notify others when necessary.

An intertask dependency is settled when all relevant significant events are instantiated and every significant event is inserted into appropriate notification lists.

4.5 MEP's Architecture

The general architecture of MEP is shown in Figure 4.1. It is easy to tell that MEP's paradigm is completely different from the client-server model. In the paradigm, every agent follows the same execution model to accomplish some tasks like offering lessons, doing homework, and so on. A temporal scheduler is plugged into each agent as a wrapper so that temporal guard scheduling can be enforced.

When a significant event in the wrapped application is about to happen, the wrapped application should initiate an attempt request to the wrapper scheduler. In Figure 4.1, control flow and data flow are distinguished by different style of lines. Data flow consisting of texts of lessons and homework is delivered only after the scheduler evaluates pertinent temporal guards and gives a positive reply.

This practical new feature is the major difference between MEP and its predecessor TFM. TFM has no separate notion of data flow. All communication among TFM agents only passes temporal guards which are no more than control flows. For a practical workflow system like MEP, once through the control flow the system has decided to do something, data flow is necessary to be passed as input or output to accomplish the computation.

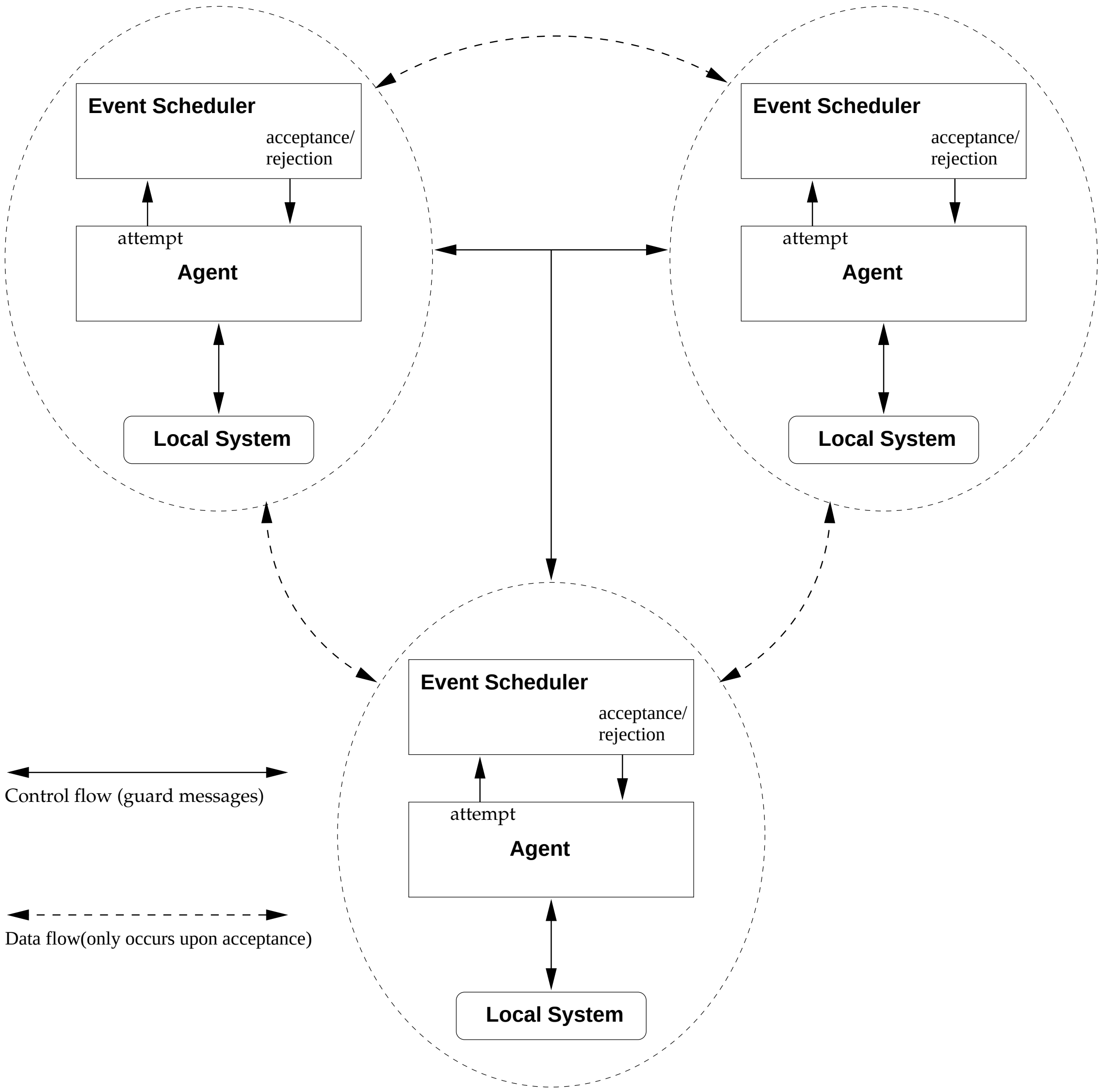

Figure 4.1: MEP architecture

The communication part of TFM is modified to accommodate the newly added data flow part. The algorithm is described below.

Whenever a message M is sent from an agent A:

1. A prepends the control flow mark '0' to the message M, if M is a temporal guard.

2. A prepends the data flow mark '1' to the message M, otherwise.

Whenever a message M is received by an agent A:

1. if the control flow mark '0' is detected at the beginning of the message M, then M is a temporal guard, A passes all of the contents of M except the control flow mark to its guard derivation procedures.

2. if the data flow mark '1' is detected at the beginning of the message M, then M is a data message, A appends all of the contents of M except the data flow mark to its data message queue.

The relation between the control flow and the data flow inside an agent is depicted in Figure 4.2. Control flow consists of execution of agents' skeletons and temporal guard residuations. Data flow occurs upon the result of guard residuations. In Figure 4.2, a single-framed circle means a control flow operation and a double-framed circle means a data flow operation. Words inside the circle describe the supposed operation. The thin arrows indicate execution sequence and the thick arrows indicate

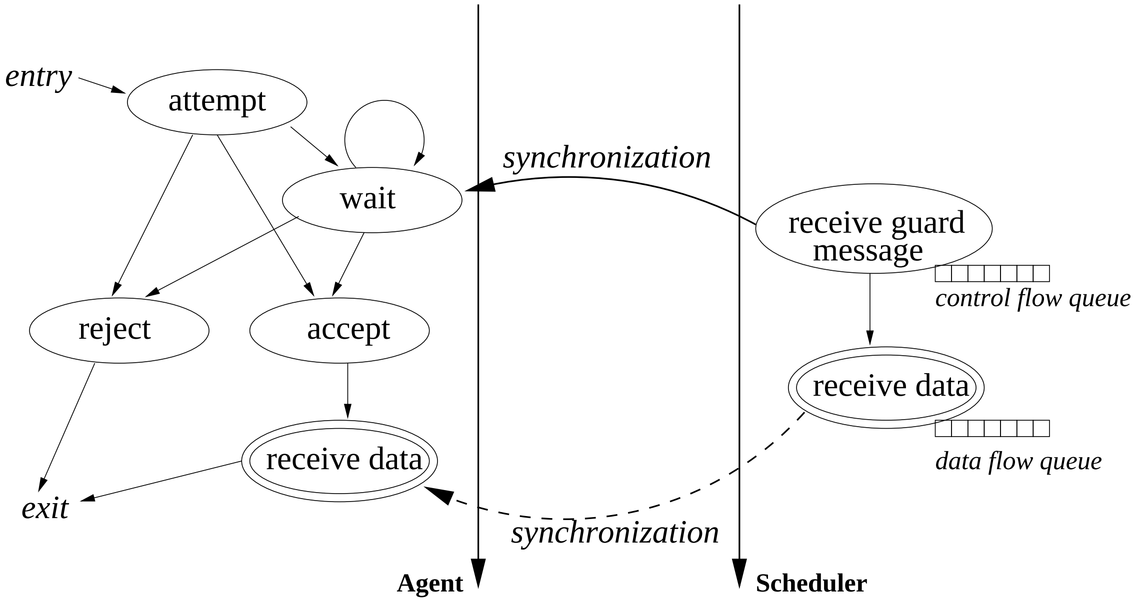

Figure 4.2: Control flow and data flow inside an agent

the direction of time. The wrapped application calls the *attempt* method as the entry and will finally exit to the calling point. The monitoring threads receive both control flow and data flow from other agents and try to synchronize with the wrapped application to whom the information flows should be sent. The synchronization requires correct message deliveries from the monitoring threads to certain execution points of the wrapped application. The initial deliveries are temporal guard messages which will eventually lead guard residuations to $\top$ or $\bot$. If $\top$ is the result, then the last delivery is the necessary data. Otherwise, data flow is avoided. In a nutshell, data flow happens upon control flow's decisions. If the monitoring thread has received multiple control flow or data flow messages before the wrapped application consumes previous ones. The newly received messages are queued in a control flow queue and a data flow queue, respectively. The implementation details of the mechanism will be discussed later in Chapter 4.6.

The active database research community also relates to our research in the sense that they use Event-Condition-Action (ECA) rules as a basic structure. MEP's architecture can be regarded as a variation of ECA rules. The control flow enforced by a temporal guard scheduler plays the role of event-condition pair. Here the received temporal guard messages act as events and the temporal guard assimilating acts as conditon checking. Then the data flow plays the role of action upon receiving permission from the control flow.

4.6 MEP's Implementation

MEP's implementation is an integration of various programming utilities. To our knowledge, these programming utilities have not been integrated before. Among them, JKQML and ABE are supplied by IBM corporation, TFM is implemented by our research group, and the Java JDK1.1 package is a product of Sun Microsystems Inc.

JKQML is implemented in Java. It enables agents to communicate with each other via KQML. ABE has a programming interface that enables adapters to be written in Java. TFM is implemented in Java precisely to facilitate applications like MEP. The Java programming language also has the most mature control structures for concurrent programming which are exploited in MEP.

MEP's implementation stresses on realizing the temporal guard scheduler that enforces intertask dependencies. The following chapters discuss each part one by one.

4.6.1 Intertask Dependencies

Intertask dependencies between the teacher agent and the student agents are extracted as:

- $\mathit{request\_text}_{student} \rightarrow \mathit{offer\_text}_{teacher}$: An temporal expression expresses the intertask dependency between the teacher agent and the student agents. When a student agent wants to learn certain lesson, he needs to attempt the *request_text* significant event. Then when the teacher agent attempts the *offer_text* significant event, the intertask dependency is enforced by the temporal guard scheduler.

- $\mathit{request\_homework}_{student} \rightarrow \mathit{offer\_homework}_{teacher}$: This is an intertask dependency between the teacher agent and the student agents. When a student agent needs to do certain homework, he attempts the *request_homework* significant event. Then when the teacher agent attempts the *offer_homework* significant event, the intertask dependency is enforced by the temporal guard scheduler. Since a teacher in the real world does almost the same thing in offering lessons and homeworks to his students, the intertask dependency is the same thing, i.e. $e \rightarrow f$.

- $send\_question_{student} < answer\_question_{teacher}$: When a student agent has a question, the intertask dependency says that either the teacher agent will not answer that question at all. This is acceptable in the case that the student agent asks something irrelevant to the course or something too difficult to be answered, or the teacher agent's answering must be preceded by the asking, otherwise it sounds illogical.

The operational semantics of the intertask dependencies could be derived from execution of guard assimilation upon the operational temporal logic discussed in Appendix A. In Figure 4.3 a graphical version of the "require" $e \rightarrow f$ dependency is offered as a more intuitive example. More details could be found in [Lin, 1998].

An event ID is associated with each significant event so that multiple attempts of the same significant event can be distinguished from each other. This feature is especially useful when there is a loop in an agent's skeleton. For example, when the teacher wants to offer multiple lessons, the significant event he attempts is the same one. Therefore, every attempt is assigned a unique ID to eliminate potential confusions.

4.6.2 Insignificant Events

Besides those significant events that appear in at least one of the intertask dependencies. A skeleton also includes some *insignificant events*. They do not appear in any intertask dependencies and correspond to internal computation. MEP uses

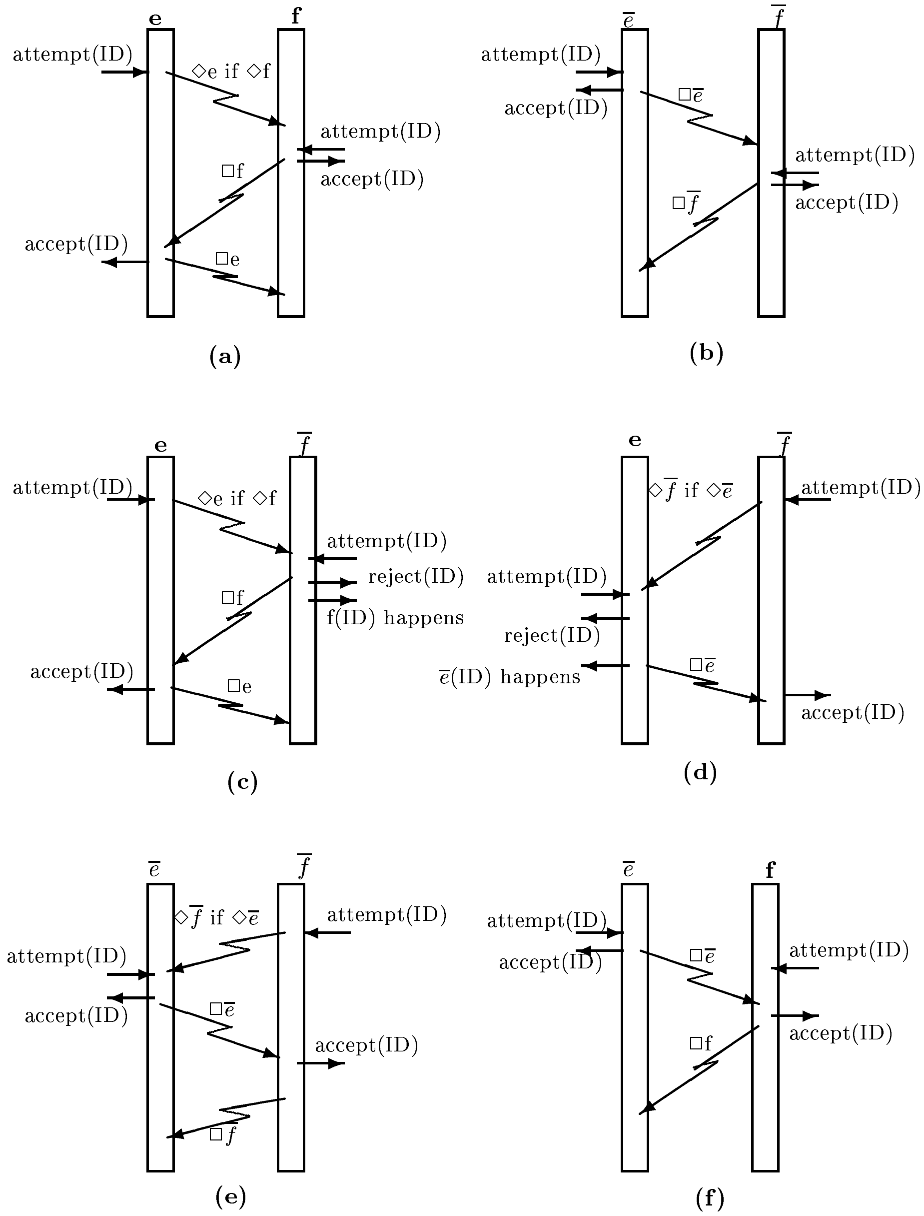

Figure 4.3: The operational semantics of $e \to f$

ABE and an internal representation to realize such insignificant events.

MEP has an internal representation in terms of Java classes to represent states and transitions of an LTS. When an LTS transition is fired, the internal representation is changed at first, then an ABE built-in events is triggered.

However, presently ABE is not strong at representing an LTS or a FSM. The built-in events can simulate an insignificant event properly and could simulate a significant event once the temporal guard scheduler is implemented as one of ABE's adapters. The problem turns to be that we need more powerful mechanisms to operate ABE's facts. For example, if a transition is fired in an LTS, then state changes, say from A to B. In the fact pool B should be added and A should be removed. This could be realized by supplying negation in RAISE or giving users chances to define their own fact operations. Otherwise, the semantics is incorrect because forward chaining will not automatically exclude out-of-date facts from the fact pool.

Right now only the "start" and "exit" insignificant events are implemented by ABE's built-in events. Other insignificant events are realized via internal representation instead. Nevertheless, as we mentioned in Chapter 4.5, MEP's information flow scheme is actually a variation of ECA rule scheme. Considering ABE is able to realize a system based on ECA rules, MEP could be implemented completely in terms of ABE after the following two conditions are achieved.

1. Negations or user-defined fact operations are supplied so that LTS transitions can be realized via ABE's events.

2. Current temporal guard scheduler is converted into an ABE adapter so that significant events can appear in ABE sensors and effectors.

4.6.3 Role Binding

One of the difficulties in implementing MEP is role-binding. *Role* is an abstraction of a group of agents who act identically in interactions. Agents playing the same role have (a) the specified intertask dependencies, (b) the specified significant events, (c) any LTS has the specified events, not necessarily the same LTS. For example, in MEP there could be multiple student agents playing the student role. Each student agent exactly follows the same interaction protocols—i.e. has exactly the same set of specified intertask dependencies and significant events. Obviously multiple teacherss will only cause confusion in a classroom, thus MEP has only one teacher agent. However, the unique teacher agent also plays a role—the teacher role. The same thing also applies to the grader who plays teaching assistant role.

We assume that the communication from or to a certain role applies to all members of that role. For example, if the teacher has ever offered a lesson, the text of that lesson should arrive to *every* agent playing student role, that is, if a student agent does not receive the text, it is invalid. In [JKQML, 1997], a facilitator is proposed so that group communication could be handled correctly. In MEP, group communication is simplified as the teacher also plays the role of the JKQML's facilitator. Whenever a student agent is instantiated, he must send a registration request to the teacher so that teacher knows where to broadcast later: the teacher agent sends lesson and homework to a student agent if and only if the student agent registers successfully on its initialization.

This corresponds to the situation where the teacher has the authority to determine whether a student can be admitted to his course. However, the registration procedures are implemented separately by a subpackage in MEP. Therefore, we can actually implement a separate facilitator later without much modification.

To implement the broadcasting of lesson texts and homework from a teacher agent to all student agents, the communication part of the TFM scheduler is modified from unicast to multicast.

4.6.4 Communication Infrastructure—JKQML

MEP uses KQML as its communication infrastructure. IBM has offers us a KQML implementation in Java—JKQML which stands for Java KQML. JKQML implements all three layers of KQML. However, we have to use our own implementation of the communication layer and the content layer. Therefore, we merely use the message layer implementation of JKQML in MEP.

The performatives `register`, `unregister`, `insert` are used in student registration; `delete-one` is used in cancel registration when a student agent exits normally; `tell` is used in passing data flow. Applying more performatives may confer MEP a sort of "intelligence" as [Ayala & Yano, 1997] and [Ayala & Yano, 1998] show in their applications.

4.6.5 Synchronization

The wrapper temporal guard scheduler spawns monitoring threads to detect incoming control flows and data flows. To achieve correct synchronization between the monitoring threads and the wrapped application is the most difficult part in the whole implementation. As a server daemon thread, a monitoring thread must correctly deliver all received temporal guards to proper significant event instances and all received data to appropriate data storage.

Let's consider a server daemon. There are three major types of implementation:

1. *Infinite loop*: The server application does an infinite loop where it executes code detecting potential connections. This is a kind of busy-waiting since the server application can not do anything but trying to accept connections. However, this method captures the easiest programming. Therefore, servers like *ftpd*, *httpd* use the method.

2. *Multithreads*: The server spawns a thread to detect potential connections. This method solves the problems caused by busy-waiting. The server is released to do anything else. However, multithreads support from either kernel or user-level is needed and programming could be very complicated.

3. *Message queuing*: The kernel maintains a message queue for holding incoming messages for every information channel. The server checks the status of its message queue constantly. If the expected message has arrived, the server extracts it out from the queue and does the neccesary processings. This method is fairly concise in programming and very efficient. However, it requires supports from low-level software.

Certainly programmers can apply a combination of the three methods in their implementations. We have used a combination of multithreads and message queuing to implement synchronization. Adopting the Java language is proved to be a proper decision because it is remarkably strong at concurrent programming. The algorithm of the synchronization is shown below.

Suppose a wrapped application A has instantiated a significant event E which in turn spawns a monitoring thread T. A and T share a queue Q and a counter variable V initialized to 0.

Whenever a data item is needed by a wrapped application A:

1. V = V-1.

2. If V is less than 0, then it means required data item is not available yet. So A has to be blocked.

3. A is waked up by T. Now T has put required data item in Q, A gets and removes the first element of Q.

Whenever a data item is received by a monitoring thread T:

1. Add the item into Q.

2. V = V+1.

3. If V is less than 0, then it means a wrapped application is waiting for the data item, T sends signal to all threads waiting on the queue.

This algorithm accomplishes asynchronous message passing between the monitoring thread as a sender and the wrapped application as a receiver.

The operational semantics mentioned in Chapter 4.4.1 requires implementation of *callback*: the agent attempts to fire a significant event, and if the guard is not $\top$ or $\bot$, the agent keeps a callback point. When new guard messages come, the agent restarts from the callback point and reevaluate its guard until the result is $\top$ or $\bot$. Unfortunately, TFM does not implement this feature. MEP fixes the problem by invoking asynchronous message passing. In other words, MEP queues control flow messages just like data flow messages, but in a different queue.

4.6.6 Concurrent Programming

In concurrent programming, the concept of *conditional message passing* (CMP) is more general than unconditional message passing like we used above. Here the message remains queued until some condition, specified by the receiver, becomes true.

The information communication channel using conditional message passing is called *conditional message channel*. MEP's design and implementation meet the definition of conditional message passing if we replace regular condition checking by temporal guard assimilation. However, things are very different when we compare the concept of condition and temporal guard. In CMP, conditions associated with every participants are not associated with each other. In other words, conditions are supplied extemporaneously, so different participants may have completely different scheme. In MEP, temporal guards are defined in terms of a common temporal logic and are enforced by a common guard scheduler, so idiosyncrasies of every participants are restricted inside applications while interaction patterns are normalized as described in the above chapters. Moreover, the temporal guards used by MEP come with a formal semantics and a distributed scheduling. Both of them, in general, are strong points.

By such a comparison, we believe MEP achieves an enhanced version of CMP by supplying a complete set of design and implementation of relevant condition evaluation along with data delivery.

Moreover, the correspondence between concurrent structures and operational semantics of temporal logic deserves to be exploited further. For example, the operation pair *signal* and *wait* could be thought of as directly corresponding to the temporal logic operators $\Box$ and $\Diamond$, respectively. When a $\Box$ operator is involved, the operational semantics requires a *signal* operation in implementation. When a $\Diamond$ operator is involved, the operational semantics requires a *wait* operation in implementation. Otherwise, the semantics are incorrect since the operators are ignored. We believe all such correspondences could be enumerated one by one in further research. A candidate concurrent structure could be *monitor* given by [Deitel, 1990], [Silberschatz & Gavin, 1994], [Stallings, 1995], [Tanenbaum & Woodhull, 1997], and [Tanenbaum, 1992]. The result may supply formal background for concurrent programming and new concurrent programming structures derived from temporal logic.

Chapter 5

MEP in Action

We illustrate the behavior of MEP through a popular course. An undergraduate programming course, CSC114, an online course from North Carolina State University's [Project25, 1997], was chosen as the content for our implementation. The development of the courses is supported by North Carolina State University to offer a collection of courses on the Internet [Project25, 1997]. The purpose is to make a clear demonstration of the degree to which the institution is ready to enter the distance education market on the Internet. The first assessment of the project shows that it is a success since the courses of the project achieve the same level of satisfaction upon preassigned criteria. Among the criteria, [Project25, 1997] rates better than the classroom based version on learner-content interaction, which significantly helps instruction. Presently fifteen courses are already available online at NCSU.

As a web-based course, CSC114 totally relies on a web browser like Netscape or Internet Explorer. The client-server model is the computational paradigm of such applications. Users through their browsers act as clients while a forever-running *httpd* daemon exists somewhere. This is completely different from our computational paradigm which is based on interacting agents.

In MEP's implementation, we only borrow data from the CSC114 course. The contents of every lessons, homework, and official homework solutions are downloaded and stored on our file systems. When an agent wants to retrieve some lessons, homework, or solutions, MEP accesses the database and gets the relevant information. For the sake of simplicity, in MEP's implementation we only use the file system as such a database and the *java.io* package as the retrieval tool.

CSC114's goal is to teach online students how to program in the programming language C++. The course is well-designed as a tutorial for both learning syntax and semantics of C++ and doing exercises and homework as a novice programmer. The course is divided into several chapters as in a book. Each chapter, which in turn is divided into several lessons, deals with a specific topic step by step. For the sake of learning by try-and-improve, at least one exercise is supplied in each lesson for the student to check whether he or she has learned the essence of the current topics. Also at least one homework is assigned in every chapter for the purpose of grading.

There are seven chapters in the course. Each of them expounds on a specific topic. For the sake of clarity, the topics of every chapter are listed in Table 5.1.

Chapter	Topics
1	Necessity of compiler, C, C++, and OOP.
2	Basic concepts of memory, data, object, type, name, operator, expression, and input/output.
3	Function, argument/parameter, return value, call by reference/value, scope, and life time.
4	Alternative statements, recursion.
5	Loop statements, input/output usage.
6	Data structures and relevant algorithms
7	Aggregates including classes

Table 5.1: Topics of each chapter in CSC114

Lessons are an electronic abstraction of the textbook. Besides offering the lessons, the teacher agent also needs to answer question from students and needs to assign homework so that students' grades can be determined. MEP's user interface supplies panels for all these features. Necessary raw data are downloaded from CSC114

Three kinds of agent are implemented to represent corresponding roles in the real world. Every agent will accomplish jobs in one of two ways, either actively or passively. An agent is active if it initiates the data flow. Otherwise it is passive, since data flow would come from other agents and the agent merely responds.

1. Teacher agent: This agent actively offers teaching materials including lesson texts and homework. Passively it answers questions from the student agents.

2. Grading agent: This agent passively grades homework and puts the gradings into academic records, then gives out official solutions to the student agents.

3. Student agent: This agent actively asks questions if it does not understand something in lesson texts or homework. Passively it should work out the homework and submit its personal solution to the grader agent.

In our implementation, when an agent is active it sends out data to other relevant agents upon receiving permission from the scheduler. When an agent is passive it waits for receiving data for processing.

The following chapters describe the user's view of every participant agent.

5.1 The Teacher Agent

MEP abstracts a teacher agent's task as:

1. The agent actively sends lesson texts to all student agents when the teacher decides to do so;

2. The agent actively sends homework to all student agents when the teacher decides to do so;

3. The agent passively receives questions from certain student agent, and then sends the teacher's answer with respect to the question back to all student agents. This situation follows what happens in traditional classrooms: a student asks a question and all the students hear both the question and the answer.

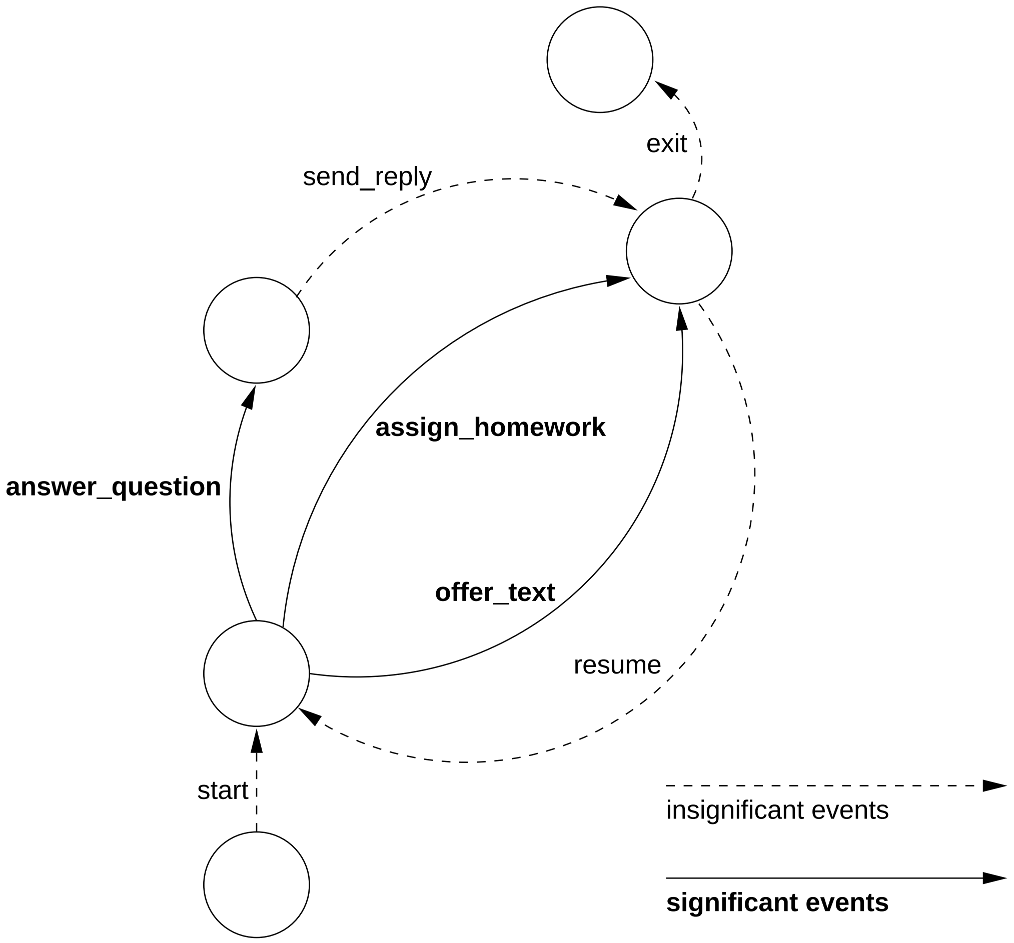

Figure 5.1: The skeleton of the teacher agent

We mentioned in Chapter 4.2 that every agent acts in terms of a skeleton. A skeleton is actually an LTS, a superset of finite state machine, which describes agent's behavior in terms of states and transitions. Figure 5.1 shows the skeleton of the teacher agent.

In Figure 5.1, significant events which are visible to the outside world are marked in bold font. The details of intertask dependencies are described in Chapter 4.6.1. Insignificant events are marked in normal font. They only affect processing inside the agent itself. In the formalization of LTS, we use a generic τ to denote all of them because they actually belong to the same type of events.

The user interface defines what an agent looks like. Figure 5.2 shows the user's view of a teacher agent.

The interface includes groups of JDK1.1 implemented panels and buttons as described below:

1. A lesson area

   The teacher can select which lesson to offer. The selection operation will not affect other agents at the moment. Once the teacher clicks the *Send Lesson* button, an intertask dependency is involved. The click operation corresponds to the "offer_text" transition in the skeleton which is required by the "request_text" transition in the student agent's skeleton.

2. A question area

   Questions from the student agents are queued. Every time the teacher clicks the *Get Question* button, the first queued question is displayed in the area. The click operation corresponds to the "answer_question" transition in the skeleton which is preceded by the "ask_question" transition in the student agent's skeleton. Once the teacher agent sends answer of current question back to all student agents, the next question will be shown here upon clicking the button again until the queue is empty.

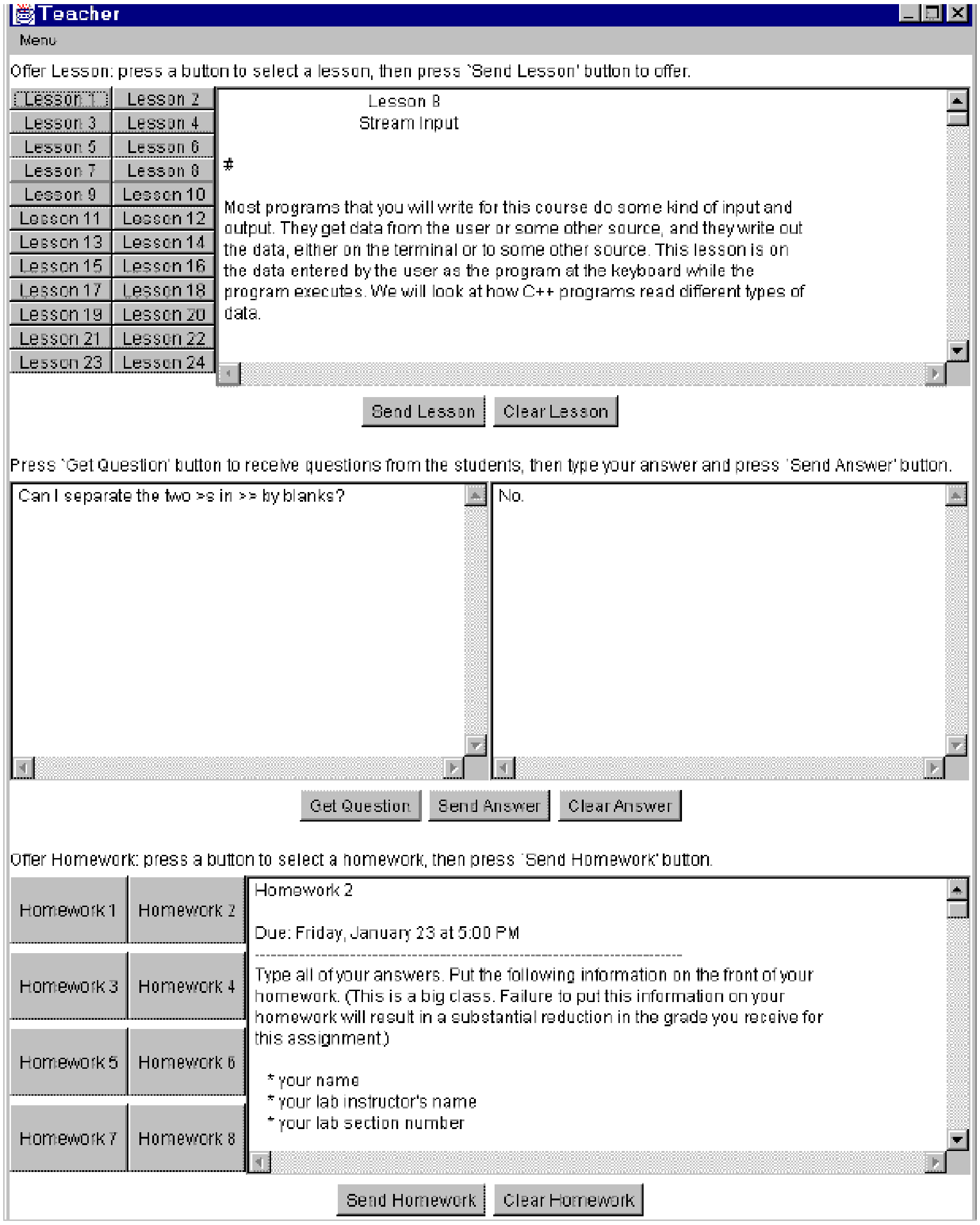

Figure 5.2: The user level view of the teacher agent

3. An answer area

   The teacher should input an answer to the current question here, then click the *Send Answer* button to send the answer back to all student agents. This action should be preceded by "ask_question" transition in a student agent's skeleton.

4. A homework area

   The teacher agent selects which homework to assign. Like for the lesson area, selection itself does not affect intertask dependencies. Only when the teacher clicks *Send Homework* button are the dependencies involved. The click operation corresponds to the "assign_homework" transition in the skeleton, which is required by the "request_homework" transition in the student agent's skeleton.

Lesson texts and homework are acquired from a local database.

5.2 The Student Agent

A student agent's responsibility includes:

1. The agent passively receives lesson texts from the teacher agent.

2. The agent passively receives homework from the teacher agent. Upon receiving a homework, the student should achieve its personal solution by virtue of his or

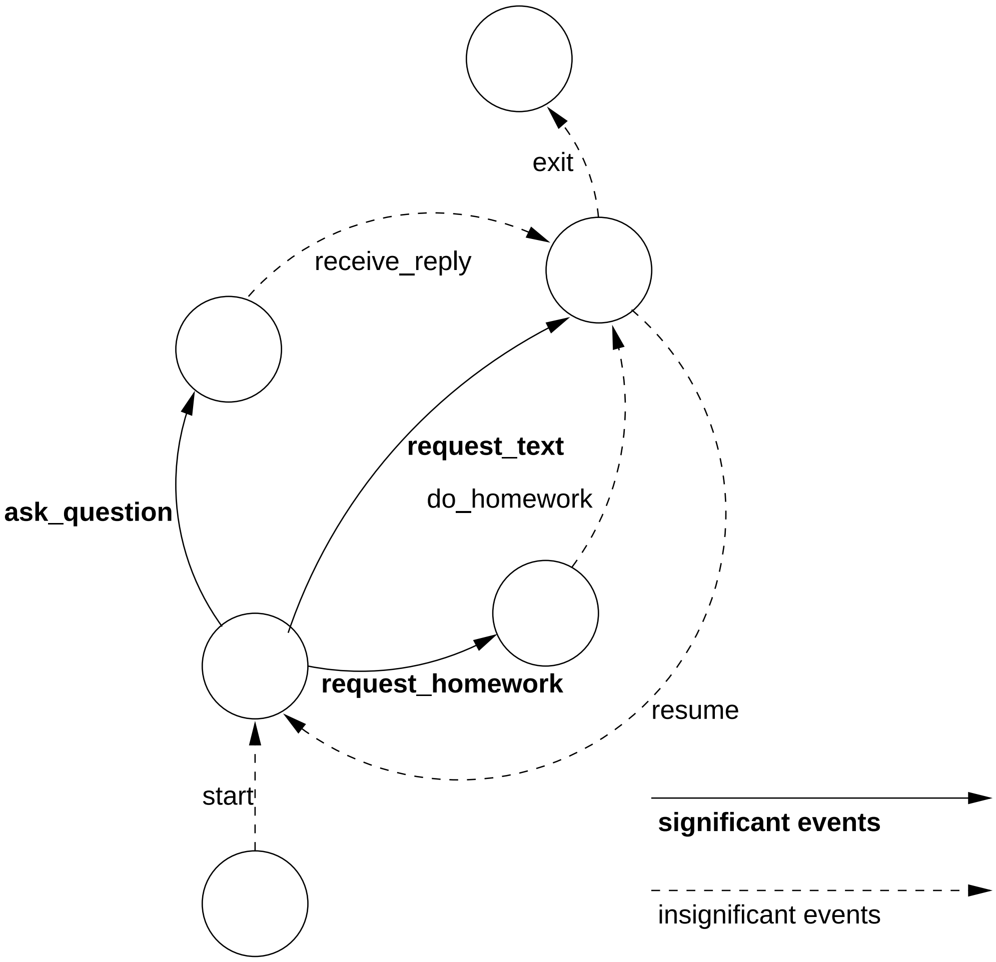

Figure 5.3: The skeleton of a student agent

her own capability and trigger a send personal solution action. The homework will reach the grader agent. Then the student agent waits until the grader agent sends the grade and the official solution back.

3. The agent actively sends its questions when it decides to do so. Then it waits until the teacher answers the questions.

Figure 5.3 shows the skeleton of student agents.

Figure 5.4 shows the user's view of a student agent.

The user interface includes groups of JDK1.1 implemented panels and buttons as described below:

1. A lesson text area

   The student can select which lesson to learn. The selection operation will not affect other agents. Once the student clicks the *Request Lesson* button, the intertask dependencies are involved. This operation corresponds to the "request_text" transition in the student agent skeleton which requires the "offer_text" transition in the teacher agent's skeleton.

2. A question area

   The student types his questions here. Once the student clicks the *Send Question* button, the questions are to be sent to the teacher agent. This operation corresponds to the "ask_question" transition in the skeleton which precedes "answer_question" transition in the teacher agent's skeleton.

3. An answer area

   The teacher's answer will be shown here.

4. A homework area

   The student can select which homework to do. The selection operation will not affect other agent. Once the student clicks the *Request Homework* button, the intertask dependencies are involved. This operation corresponds to the

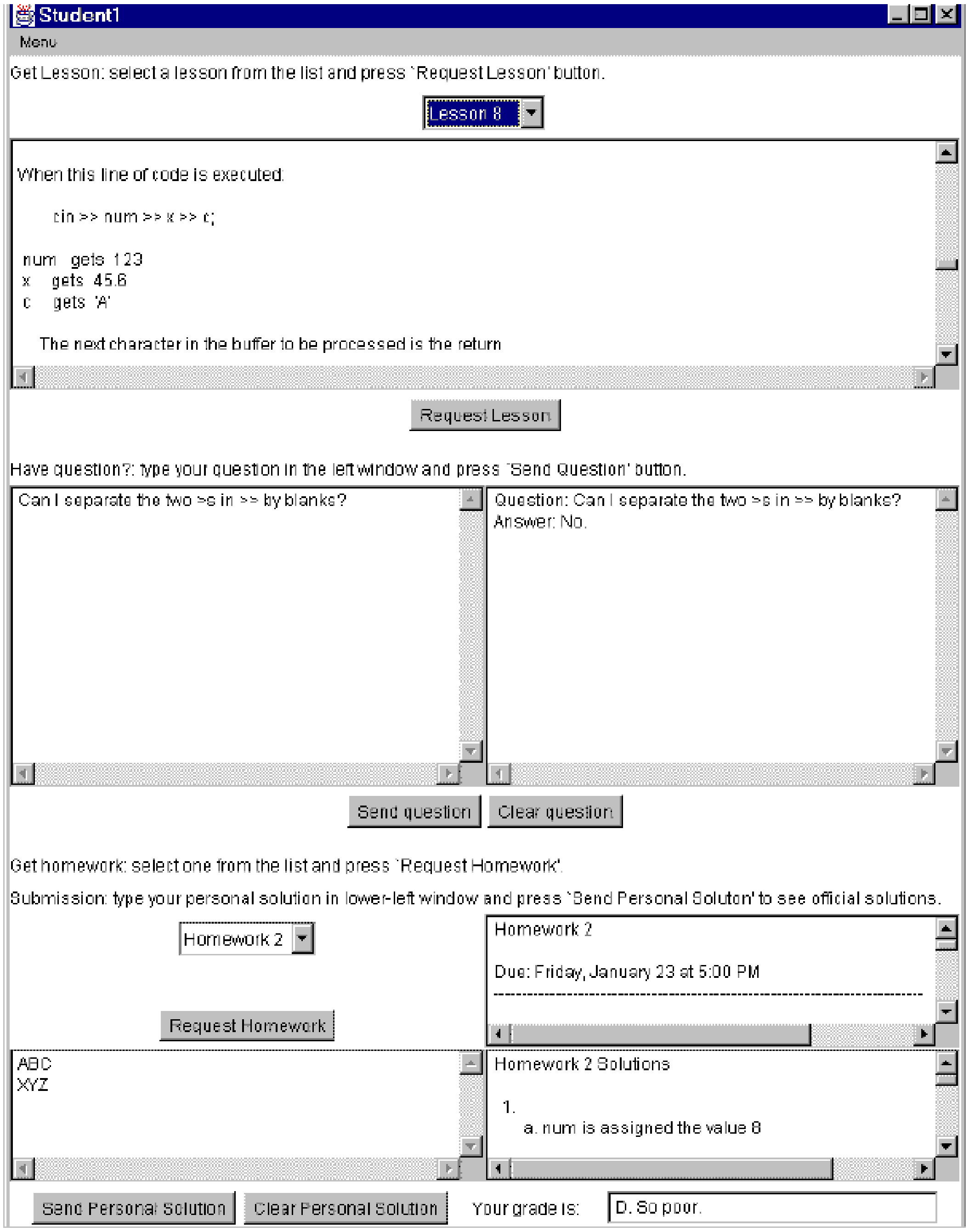

Figure 5.4: The user level view of a student agent

"request_homework" transition in the student agent skeleton which requires the "assign_homework" transition in the teacher agent's skeleton.

5. A personal solution area

   Once the student agent receives a homework, the student should work out his personal solution here by virtue of his own capability.

6. An official solution area

   After the student works out the homework in personal solution area. He should press "Send Personal Solution" button to send it to the grader agent. Then it waits until the grader agent sends the grade and the official solution back.

5.3 The Grader Agent

The grader agent corresponds to a teaching assistant in the real world. A grader agent's responsibility is:

- The agent passively receives personal solutions from student agents, then the grader grades the personal solutions according to official solutions, then sends back the grade with the official solutions.

Official solutions are acquired from a local database.

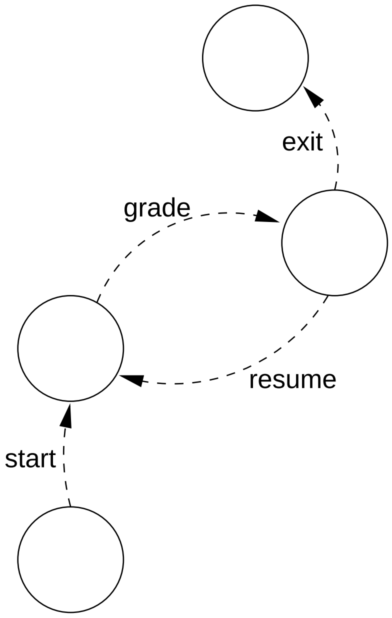

Figure 5.5: The skeleton of the grader agent

Figure 5.5 shows the skeleton of a grader agent.

The grader agent's view is implemented with JDK1.1 and includes a submission area, and an official solution area. Figure 5.6 shows the user's view of a student agent.

1. A submission area

   Submissions of personal solutions of students are queued and displayed in the area one by one. Once the agent grades the current submission and sends the grade back to a student, the next student's submission will be shown here until no more submissions arrive.

2. An official solution area

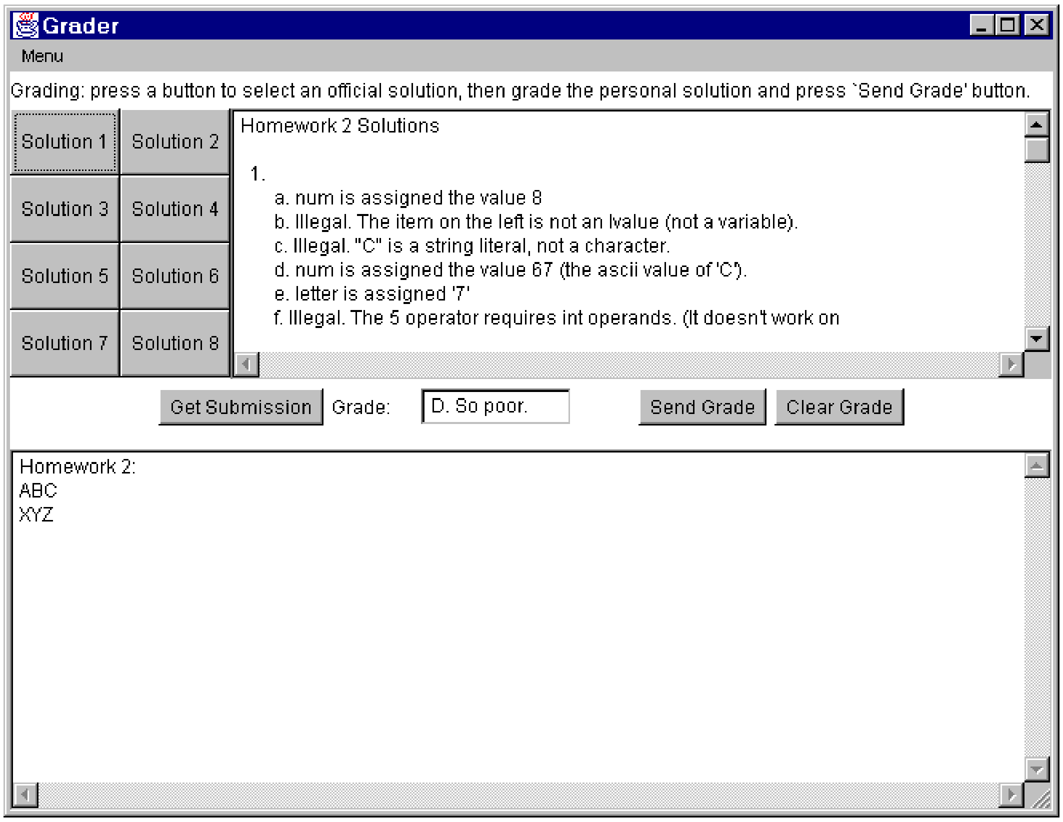

Figure 5.6: The user level view of the grader agent

The contents of this area is triggered by current submission. The agent grades the current submission based on standard solutions shown in this area.

Chapter 6

Conclusion and Future Work

This thesis presents an educational application based on multiagent technology. It discusses the ideas, design, implementation of MEP as well as the valuable experience gained. In MEP, participant agents are peers rather than masters and slaves. They make their own decisions provided intertask dependencies among them are satisfied. We believe that such a paradigm of computation is more appropriate and generic in describing distributed systems. The project shows that implementation is not a problem for realizing this paradigm if programmers can give up the concepts from traditional master-slave systems and adopt those of multiagent systems.

6.1 Evaluation of MEP

It is important to notice that MEP stresses on openness, autonomy, and flexibility of software systems. Compared to the traditional computation model—Turing Machines, interactive machines have single or multiple input streams and synchronous or asynchronous communication channels connecting with outside world. Thus they enable modelling open systems.

As to autonomy, MEP respects users' decisions in both the architecture and the implementation. Agents are autonomous in executing their LTS skeletons, only may be constrained regarding significant events that are non-autonomous. Since intertask dependencies define all and only the interactive actions, the minimal set of constraints is applied. Therefore, maximal level of autonomy is achieved.

As to flexibility, which means that interactions among agents will not fail due to agents' autonomous decisions, the separation of control and data flow enforce the expected interactions by giving agents opportunities to make decisions and scheduling relevant data flows appropriately according to the decisions. We may compare MEP's scheme with a common communication scheme like RPC. In RPC, information flows are mixed together by packing data flows into control flows. Although openness is not violated, it is hard to achieve autonomy and flexibility in such a scheme because unexpected data flows can not be easily avoided.

6.2 Potential Enhancements

Presently intertask dependencies are expressed as temporal logic expressions. Temporal guards of the dependencies are derived manually from the expressions. To automatically derive temporal guards for every intertask dependency, a temporal logic expression compiler is needed to translate a dependency expression into guard expressions. What the compiler does is exactly the same thing as residuation which we illustrate in Appendix A. Such a compiler could be offered as a utility associated with MEP to facilitate those programmers who do not know how to evaluate temporal expressions.

The same idea could also be applied to LTS representation. The most user-friendly way to express an LTS is to draw a graph. General purpose graph tools could be used here. However, to our knowledge none of them can produce Java code and appropriately describe an LTS. Therefore, a specially designed graph tool is needed. First the graph tool should be able to draw a LTS, then Java code is produced to instantiate the LTS and associated with other components of MEP.

Currently MEP does not distinguish various classes of significant events. In [Singh, 1997b], significant events are classified as *flexible*, *inevitable*, *immediate*, and *triggerable* events. To apply such a classification to MEP, an enhanced version of temporal guard scheduler is needed to enforce various semantics derived from various classes.

As mentioned before, JKQML implements all three layers of KQML and we only use the message layer. If JKQML can classify its Java classes clearer to separate the three layers, then we may only need to import those applicable classes instead of all of them.

As to ABE, MEP could be completely implemented in ABE when the following two conditions are achieved: (a) Negations or user-defined fact operations are supplied so that LTS transitions can be realized via ABE's events; (b) TFM is converted into an ABE adapter so that significant events can appear in ABE sensors and effectors.

As to concurrent programming, the correspondence between concurrent structures and operational semantics of temporal logic deserves to be exploited further. For example, the operation pair *signal* and *wait* could be thought as directly corresponding to the temporal logic operators $\Box$ and $\Diamond$, respectively. When a $\Box$ operator is involved, the operational semantics requires a *signal* operation in implementation. When a $\Diamond$ operator is involved, the operational semantics requires a *wait* operation in implementation. Otherwise, the semantics are incorrect since the operators are ignored. The potential starting point could be exploiting correspondence between monitor and operational semantics of temporal logic as mentioned in Chapter 4.6.6.

Appendix A

Temporal Logic for Guards

This appendix is largely borrowed from [Singh, 1997a] and is included only for completeness.

$\mathcal{T}$ is the language in which the guards are expressed. Intuitively, for expressions $E, F \in \tau$, $\Box E$ means E will always hold; $\Diamond E$ means E will eventually hold (thus $\Box E$ entails $\Diamond E$); and $\neg E$ means E does not yet hold; $E \cdot F$ means F has occurred preceded by E. For simplicity, we assume the following associativity precedence (in decreasing order): $\neg \cdot \Box\Diamond$, then $\wedge$, then $\vee$.

Syntax **3** $\Gamma \subseteq \mathcal{T}$

Syntax **4** $E_1, E_2 \in \mathcal{T} \Rightarrow E_1 \vee E_2, E_1 \wedge E_2, E_1 \cdot E_2, \Box E, \Diamond E, \neg E \in \mathcal{T}$

The semantics of $\mathcal{T}$ is given with respect to a trace (as for $\mathcal{E}$) and an index into that trace (unlike for $\mathcal{E}$). This semantics characterizes progress along a given computation to determine the decision on each event. It has important differences from traditional linear temporal logics. One, our traces are sequences of events, not of states. Two, most of our semantic definitions are given in terms of a pair of indices, that is, intervals, rather than a single index. For $0 \leq i \leq k, u \models_{i,k} E$ means that E satisfied over the subsequence of u between i and k. For $k \geq 0, u \models_{i,k} E$ means that E is satisfied on u at index k—implicitly, i is default to 0.

A trace, u, is *maximal* iff for each event, either the event or its complement occurs on u. $U_{\mathcal{T}}$ is defined as the set of maximal traces. We assume $\Xi \neq \emptyset$, hence, $\Gamma \neq \emptyset$.

Semantics **6** $u \models_i E$ iff $u \models_{0,i} E$

Semantics **7** $u \models_{i,k} f$ iff $(\exists j : i \leq j \leq k$ and $u_j = f)$, where $f \in \Gamma$

Semantics **8** $u \models_{i,k} E \vee F$ iff $u \models_{i,k} E$ or $u \models_{i,k} F$

Semantics **9** $u \models_{i,k} E \wedge F$ iff $u \models_{i,k} E$ and $u \models_{i,k} F$

Semantics **10** $u \models_{i,k} E \cdot F$ iff $(\exists j : i \leq j \leq k$ and $u \models_{i,j} E$ and $u \models_{j+1,k} F)$

Semantics **11** $u \models_{i,k} \top$

Semantics **12** $u \models_{i,k} \neg E$ iff $u \not\models_{i,k} E$

Semantics **13** $u \models_{i,k} \Box E$ iff $(\forall j : k \leq j \Rightarrow u \models_{i,j} E)$

Semantics **14** $u \models_{i,k} \Diamond E$ iff $(\exists j : k \leq j$ and $u \models_{i,j} E)$

EXAMPLE **3** Let $t = \langle e_1 e_2 e_3 e_4 \rangle$ be a trace in U_Γ, we can verify that

$u \models_0 \Diamond e_4$

$u \models_0 \neg e_1 \wedge \neg e_2 \wedge \neg e_3 \wedge \neg e_4$

$u \models_0 \Diamond(e_2 \cdot e_3)$

$u \not\models_0 \Diamond(e_3 \cdot e_2)$

$u \models_1 \Box e_1 \wedge \neg e_2 \wedge \neg e_3 \wedge \neg e_4$

$u \not\models_1 e_1 \cdot e_4$

$u \models_2 e_1 \cdot e_2$

$u \models_3 e_1 \cdot e_3$

$u \models_4 e_1 \cdot e_2 \cdot e_3 \cdot e_4$ ▮

Given an intertask dependency specified in temporal logic expression, it is important to find the correct guard for the dependency. Obviously wrong guards will lead the system into incorrect states even if our scheduler works well. In this chapter, a guard derivation method offered by [Singh, 1996] will be briefly discussed. We can get correct temporal guards for specified intertask dependencies either by manual derivation or by a compiler who automatically does exactly what manual derivation does.

We term the derivation *residuation* and notate it by an operator $/ : \mathcal{E} \times \Xi \mapsto \mathcal{E}$, which is not in $\mathcal{E}$ or $\mathcal{T}$. Roughly, given a dependency D and event e, D/e gives the residual or "remnant" of D after e occurs.

$/$ can be computed symbolically. We propose a set of equations to compute the residual of a dependency with respect to an event.

EQUATION **1** $\bot/e = \bot$

EQUATION **2** $\top/e = \top$

EQUATION **3** $(E_1 \wedge E_2)/e = ((E_1/e) \wedge (E_2/e))$

EQUATION **4** $(E_1 \vee E_2)/e = ((E_1/e) \vee (E_2/e))$

EQUATION **5** $(e \cdot E)/e = E,$ if $e \notin \Gamma_E$

EQUATION **6** $D/e = D,$ if $e \notin \Gamma_D$

EQUATION **7** $(e' \cdot E)/e = 0,$ if $e \in \Gamma_E$

EQUATION **8** $(\overline{e} \cdot E)/e = 0$

The temporal guards are given by

DEFINITION **2** The operator G for computing guards is defined as $G : \mathcal{E} \times \Xi \mapsto \mathcal{T}$:

- $G(D_1 \vee D_2, e) \underline{\triangle} G(D_1, e) \vee G(D_2, e)$
- $G(D_1 \wedge D_2, e) \underline{\triangle} G(D_1, e) \wedge G(D_2, e)$
- $G(e_1 \cdot \ldots \cdot e_i \cdot \ldots \cdot e_n, e_i) \underline{\triangle} \Box e_1 \wedge \ldots \wedge \Box e_{i-1} \wedge \Diamond(e_{i+1} \cdot \ldots \cdot e_n)$
- $G(e_1 \cdot \ldots \cdot e_i \cdot \ldots \cdot e_n, \overline{e_i}) \underline{\triangle} \bot$
- $G(e_1 \cdot \ldots \cdot e_i \cdot \ldots \cdot e_n, e) \underline{\triangle} \Diamond(e_1 \cdot \ldots \cdot e_n),$ if $e, \overline{e} \not\subseteq \{e_1, \overline{e_1}, \ldots, e_n, \overline{e_n}\}$

- $G(\perp, e) \;\underline{\triangle}\; \perp$

- $G(\top, e) \;\underline{\triangle}\; \top$

$\blacksquare$

Now let's use the definition to compute temporal guards for the two types of intertask dependencies used in MEP. The first one is "require" $D_{\rightarrow} = \overline{e} \vee f$:

- $G(D_{\rightarrow}, e) = G(\overline{e}, e) \vee G(f, e) = \perp \vee \Diamond f = \Diamond f$

- $G(D_{\rightarrow}, \overline{e}) = G(\overline{e}, \overline{e}) \vee G(f, \overline{e}) = \top \vee \Diamond f = \top$

- $G(D_{\rightarrow}, f) = G(\overline{e}, f) \vee G(f, f) = \Diamond \overline{e} \vee \top = \top$

- $G(D_{\rightarrow}, \overline{f}) = G(\overline{e}, \overline{f}) \vee G(f, \overline{f}) = \Diamond \overline{e} \vee \perp = \Diamond \overline{e}$

and the second one is "precede" $D_{<} = \overline{e} \vee \overline{f} \vee e \cdot f$:

- $G(D_{<}, e) = G(\overline{e}, e) \vee G(\overline{f}, e) \vee G(e \cdot f, e) = \perp \vee \Diamond \overline{f} \vee \Diamond f = \Diamond \overline{f} \vee \Diamond f = \neg f$

- $G(D_{<}, \overline{e}) = G(\overline{e}, \overline{e}) \vee G(\overline{f}, \overline{e}) \vee G(e \cdot f, \overline{e}) = \top \vee \Diamond \overline{f} \vee \perp = \top$

- $G(D_{<}, f) = G(\overline{e}, f) \vee G(\overline{f}, f) \vee G(e \cdot f, f) = \Diamond \overline{e} \vee \perp \vee \Box e = \Diamond \overline{e} \vee \Box e$

- $G(D_{<}, \overline{f}) = G(\overline{e}, \overline{f}) \vee G(\overline{f}, \overline{f}) \vee G(e \cdot f, \overline{f}) = \Diamond \overline{e} \vee \top \vee \perp = \top$

Table A.1 illustrates how a guard message is assimilated so that an event can get its new guard evaluated. The assimilation process is captured by operator $\div$.

Old Guard G	**Message** M	**New Guard** $G \div M$
$G_1 \vee G_2$	M	$G_1 \div M \vee G_2 \div M$
$G_1 \wedge G_2$	M	$G_1 \div M \wedge G_2 \div M$
$\Box e$	$\Box e$	$\top$
$\Box \overline{e}$	$\Box e$ or $\Diamond e$	$\bot$
$\Diamond e$	$\Box e$ or $\Diamond e$	$\top$
$\Diamond \overline{e}$	$\Box e$ or $\Diamond e$	$\bot$
$\Box(e_1 \cdot e_2)$	$\Box e_1 \wedge \neg e_2$	$\Box e_2$
$\Box(e_1 \cdot e_2)$	$\Box e_2 \wedge \neg e_1$	$\bot$
$\Box(e_1 \cdot e_2)$	$\Box \overline{e_i}$ or $\Diamond \overline{e_i},\ i \in \{1,2\}$	$\bot$
$\Diamond(e_1 \cdot e_2)$	$\Box e_1 \wedge \neg e_2$	$\Diamond e_2$
$\Diamond(e_1 \cdot e_2)$	$\Box e_2 \wedge \neg e_1$	$\bot$
$\Diamond(e_1 \cdot e_2)$	$\Box \overline{e_i},\ i \in \{1,2\}$	$\bot$
$\neg e$	$\Box e$	$\bot$
$\neg \overline{e}$	$\Box e$ or $\Diamond e$	$\top$
G	M	G,otherwise

Table A.1: Guard message assimilation

Bibliography

[ABE, 1998] IBM's Agent Building Environment–ABE. http://www.networking.ibm.com/iag/iagsoft.htm.

[Attie *et al.*, 1993] Attie, Paul C.; Singh, Munindar P.; and Sheth, A.; 1993. Using flexible transactions to support multi-system telecommunication applications. In *Proceedings of the 19th VLDB Conference*. 134–145.

[Austin, 1962] Austin, J. L.; 1962. *How to Do Things with Words*. Oxford University Press.

[Ayala & Yano, 1997] Ayala, Gerardo and Yano, Yoneo; 1997. Evaluating the performance of agents that support the effective collaboration of learners in a CSCL environment. *IEICE Transactions on Information and Systems* E80-D(2):125–134.

[Ayala & Yano, 1998] Ayala, Gerardo and Yano, Yoneo; 1998. A collaborative learning environment based on intelligent agents. *Expert Systems with Applications* 14:129–137.

[Barbuceanu & Fox, 1995] Barbuceanu, Mihai and Fox, Mark S.; 1995. COOL: A language for describing coordination in multi agent systems. In *Proceedings of the International Conference on Multiagent Systems.* 17–24.

[Deitel, 1990] Deitel, Harvey M.; 1990. *An Introduction to Operating Systems.* Addison-Wesley, second edition.

[Emerson, 1990] Emerson, E. Allen; 1990. Temporal and modal logic. In Leeuwen, Jan van, editor, *Handbook of Theoretical Computer Science*, volume B. North-Holland, Amsterdam. 995–1072.

[Enderton, 1972] Enderton, Herbert B.; 1972. *A Mathematical Introduction to Logic.* Academic Press, San Diego.

[FIPA, 1997a] Foundation for Intelligent Physical Agents, Geneva, Switzerland. *FIPA 97 Specification, Part II, Agent Communication Language.*

[FIPA, 1997b] Foundation for Intelligent Physical Agents, Geneva, Switzerland. *FIPA 97 Specification, Part III, Agent Software Integration.*

[JKQML, 1997] Object Technology Systems, IBM Yamato Laboratory, Japan. *JKQML Specification.*

[Kick, 1995] Kick, Alexander; 1995. An agent algebra for the formal description and verification of multi agent systems. In *Proceedings of the International Conference on Multiagent Systems.* Poster.

[Kuwabara, 1996] Kuwabara, Kazuhiro; 1996. Meta-level control of coordination protocols. In *Proceedings of the International Conference on Multiagent Systems*. 165–172.

[Lin, 1998] Lin, Chubin; 1998. Design and implementation of a distributed workflow scheduler. Master's thesis, North Carolina State University.

[Lynch *et al.*, 1994] Lynch, Nancy; Merritt, Michael; Weihl, William; and Fekete, Alan; 1994. *Atomic Transactions*. Morgan Kaufmann, San Mateo.

[Parunak, 1996] Parunak, H. Van Dyke; 1996. Visualizing agent conversations: Using enhanced Dooley graphs for agent design and analysis. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS)*. 275–282.

[Project25, 1997] Project 25: NC State University Courses on the Internet. http://courses.ncsu.edu/.

[Silberschatz & Gavin, 1994] Silberschatz, Arbaham and Gavin, Peter B.; 1994. *Operating System Concepts*. Addison-Wesley, fourth edition.

[Singh & Tomlinson, 1994] Singh, Munindar P. and Tomlinson, Christine; 1994. Workflow execution through distributed events. In *Proceedings of the 6th International Conference on Management of Data*. 117–132.

[Singh, 1996] Singh, Munindar P.; 1996. Distributed scheduling of workflow computations. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 61–75.

[Singh, 1997a] Singh, Munindar P.; 1997a. Formal aspects of workflow management. TR 97-04 and 97-05, Department of Computer Science, North Carolina State University, Raleigh.

[Singh, 1997b] Singh, Munindar P.; 1997b. Formal aspects of workflow management, part 2: Distributed scheduling. TR 97-05, Department of Computer Science, North Carolina State University, Raleigh.

[Stallings, 1995] Stallings, William; 1995. *Operating Systems*. Prentice-Hall, second edition.

[Tanenbaum & Woodhull, 1997] Tanenbaum, Andrew S. and Woodhull, Albert S.; 1997. *Operating Systems: Design and Implementation*. Prentice-Hall, second edition.

[Tanenbaum, 1992] Tanenbaum, Andrew S.; 1992. *Modern Operating Systems*. Prentice-Hall.

[Wegner, 1997] Wegner, Peter; 1997. Why interaction is more powerful than algorithms. *Communications of the ACM* 40(5):80–91.

[Wegner, 1998] Wegner, Peter; 1998. Interactive foundations of computing. *Theoretical Computer Science* 192(2):315–351.