International Journal of Artificial Intelligence in Education (2000), 11, 1-32

CAROL5: An Agent-Oriented Programming Language for Developing Social Learning Systems

Wen-Cheng Wang, Tak-Wai Chan *Department of Computer Science and Information Engineering, National Central University, Chung-Li, 32054, Taiwan*
email: {cheng, chan}@src.ncu.edu.tw

Abstract. In this paper, we share our experience of designing CAROL5 and how to use it to develop social learning systems. Social learning systems are emerging learning environments that allow multiple students and agents to work at the same computer or across connected machines via various protocols of learning activities. Our experience shows that lacking a good development system is a big obstacle to the advancement of these new breeds of learning environments. We cannot find any existing development system that fulfills all the requirements for developing social learning systems. This motivated us to design a simple general-purpose programming language that is powerful enough to model such systems. To this end, we have designed an agent-oriented programming (AOP) language named CAROL5. The design of CAROL has evolved in five versions and is the continuation of the intention in developing Curriculum-Tree, a simple architecture to support the development of the first learning companion system.

1. INTRODUCTION

Recent research on agents has been rapidly progressing in various subfields of computer science. It seems that agents have suddenly appeared as a hot subject of research in artificial intelligence (AI), network communication, computer-human interface, intelligent computer-assisted learning (ICAL), programming languages, software engineering, etc., despite the fact that the word 'agent' is not new nor well defined in computer science. The notion of agents can be traced back to the early days of distributed AI (DAI) research in the 1970s. Strictly speaking, the common goal of all research in AI is to construct software that will recreate intelligent human behavior in all respects. AI researchers named these kinds of software as 'intelligent agents'. Intelligent agents can perform tasks on behalf of users whether they are present or absent, and intend to reduce users' working load. However, owing to the bottleneck on fully understanding human intelligence, the big dream of creating truly intelligent agents never comes true. It is nevertheless widely believed that agents that can exhibit *some* aspects of human intelligence would still be useful.

The source of the current research stream on agents is closely related to the advancement of network information. In this era of globe-spanning connectivity, computers and the network behind them are becoming important vehicles of information. An increasing number of untrained users are willing to interact with computers to make use of the network information resource. The need to reduce 'information overload' and 'information anxiety' of the population urges researchers of computer science to make computer systems more friendly and easy to use. Agents bear potential to help the population make effective use of the computer and networks, since they perform tasks on the user's behalf. They participate actively in accomplishing tasks, rather than serving as passive tools like today's applications. They assist users in a range of different ways: they hide the complexity of difficult tasks, teach or train the user, help different users in a community collaborate and monitor events and routine jobs.

However, researchers from different subfields often focus on different aspects of agents. For example, AI researchers devote themselves to constructing agents' knowledge bases and enabling them to acquire knowledge by learning or imitating human's behaviors. Network communication researchers concentrate on enabling the 'mobility' of agents, so that they can migrate from one site to another for accessing network resources or meeting other agents at

remote locations. Computer-human interface researchers pay attention to designing ‘social user interfaces’, which will allow agents to make social interaction with users or other agents for collaboration. As they focus on different aspects, the word ‘agent’ makes different senses to different researchers. Interestingly, after several years of arguments on the definition of agents, there is a trend to integrate all these aspects of agents. An obvious indication of this trend is that many recently proposed projects (Gilmore et al. 1995, Bradshaw et al. 1997) intend to merge all these heterogeneous senses of agents into their own integrated architectures. Basically, the original meaning of agents -- computer programs that perform tasks on behalf of human beings -- is still applicable to these integrated architectures. Different from traditional programs, agents are generally considered to possess knowledge of a specific domain, be aware of the user's preference and habit, and have some degree of autonomy, and thus they can actively assist users to accomplish their tasks or directly act on behalf of them. Of course, agents should have the ability to communicate with other agents or even users, so that they can perform more complex tasks by collaboration. In addition, because users will access network resources, mobility and network communication are also considered to be necessary for agents.

It is time to take the role and application of agents in learning systems into overall consideration. In fact, the notion of agents is not new to the field of ICAL. Since ICAL research adopted a large number of AI techniques, the notion of agents was propagated into ICAL research very early. In the late 1970s, computer scientists with a mainly AI research background launched research into intelligent tutoring systems (ITSs). Their approach was to simulate the computer as an intelligent tutor who can understand the student and provide adaptive tutoring. We can view the intelligent tutor residing in an ITS as an agent that plays the role of a human tutor. In the middle of 1980s, another party of researchers (Self 1985, Gilmore and Self 1988) suggested that the computer should be simulated as a co-learner, rather than a tutor as in ITSs, to cooperate with instead of teaching the student. From today’s view, a co-learner is an agent that plays the role of a classmate. Later, learning companion systems (LCSs) were proposed by Chan and Baskin (Chan and Baskin 1988, Chan and Baskin 1990). LCS is a more general notion of agent-based systems for learning with multiple agents. They suggest that the computer can simulate two co-existing agents: a teacher and a learning companion, which will have various interactions with the human student and the computer teacher. For example, the companion and the student can collaborate with the tutor, compete against each other, or the companion can be taught by the student

Social learning systems (Chan 1995, Chan 1996) are emerging learning environments that extend LCSs by allowing multiple students and agents to work at the same computer or across connected machines under different protocols of learning activities. The notion of agents is the central part of various social learning models. Educational agents can act as virtual tutors, virtual students, or virtual learning companions that can help students in learning. In this paradigm, instead of user-initiated interaction via commands, the user is engaged in a cooperative or competitive process in which human and educational agents both initiate communication, monitor events, and perform learning activities. Also, agents can act as virtual personal assistants for students or virtual personal assistants for teachers that can assist them in managing their learning activities, such as scheduling group meeting, reminding the deadline of homework, mining information from a digital library, etc.

The social context has been long considered a catalyst of knowledge cultivation and motivation. Social learning systems have attracted much attention because they socialize computer-assisted learning. However, the lack of a good development tool is a big obstacle to the advancement of social learning systems. We could not find any existing development tool that fulfills the requirements for developing social learning systems. For example, most authoring systems that support networking and multimedia do not provide a good script language or mechanism to develop a complex intelligent educational environment with features of problem solver, courseware control flow, student model, and teaching strategies, not to mention educational agents. This motivated us to design a simple general-purpose programming language that is powerful enough to model such complex learning systems. For this end, we have designed an agent-oriented programming (AOP) language (Shoham 1993) named CAROL5. CAROL5 has been implemented in C++ and has been used in developing some social learning systems for experiment.

Designing a programming language for social learning systems development is somewhat different from designing a normal programming language. Although CAROL5 can be used to develop general applications too, we kept some special requirements of learning systems, especially social learning systems, in mind when designing the language. We believe that with a carefully designed agent-oriented programming language as the core of the developing environment, the complexity of developing social learning systems can be largely reduced. In the following sections, we will share our experience of designing CAROL5 and developing social learning systems.

2. OVERVIEW OF CAROL5

The design of CAROL is the continuation of the intention in developing Curriculum-Tree (Chan, 1992). The goal of developing Curriculum-Tree is to build a knowledge-based framework that allows non-AI experts to construct their own learning companion systems for a complete course. The development of Curriculum-Tree was motivated by the attempt at an effective implementation of Integration-Kid (Chan, 1991), the first learning companion system in the domain of learning indefinite integration. At an early stage of implementing Integration-Kid, it was discovered that the system complexity went far beyond the capability of a simple-minded knowledge system; nor could it be specifically and accurately handled by sophisticated general purpose knowledge-based systems such as KEE. Part of the complexity is due to the additional agent which makes the structure of the learning activities more complex than that in traditional ITS.

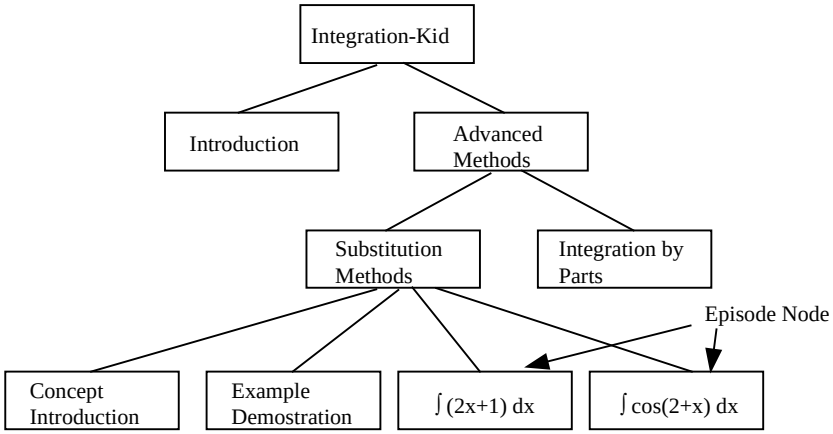

Figure 1. Part of Integration-Kid's Curriculum-Tree

In education, a curriculum is defined as pre-designed teaching goals or learning activities based on the constraints of the domain knowledge. Instructional design is the design of a scheme of learning activities. The basic idea of Curriculum-Tree is that teaching or tutoring can be viewed as the execution of previously planned activities accompanied by monitoring the process. A teacher's plan can be characterized by its global curriculum planning with local decision making in monitoring its execution. To represent the abstraction of a curriculum of learning activities, a curriculum-tree organizes the actual program of the learning activities according to the domain knowledge structure in a tree structure. Figure 1 shows part of Integration-Kid's curriculum-tree.

In a curriculum-tree, each episode node, which represents an episode in the discourse of learning, is a blackboard system in which three separate agents (teacher, companion, and student) communicate through a shared blackboard locally scheduled by an agent scheduler (Figure 2). The agent scheduler acts as a time-sharing mechanism among three agents, so that it looks like they function concurrently. This time-sharing mechanism also guarantees that there is only one agent accessing the blackboard at a time, thus avoiding conflict among agents when they try to update data in the blackboard.

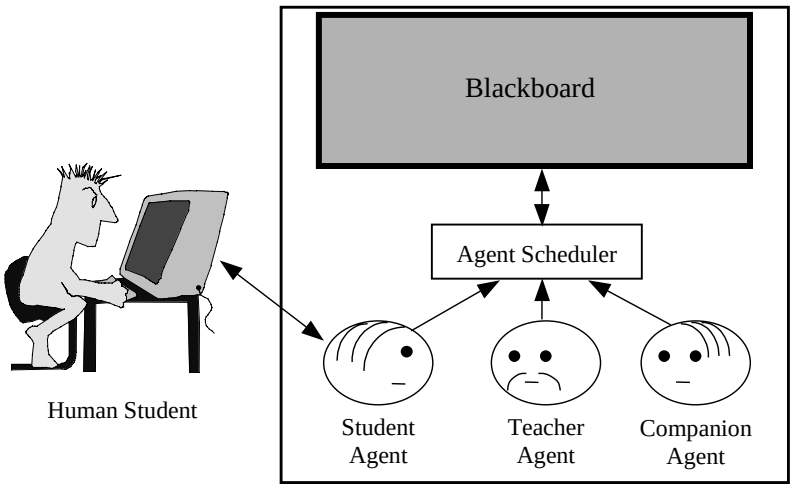

Figure 2. Three agents via a Blackboard

Each agent is implemented as a set of *rules of behavior* which models the behavior of the agent. The behavior of the human student is driven by his/her own intelligence. But, for the teacher and the companion, their behavior is based on their own domain knowledge. The student agent contains those rules that interpret the student's input which is put on the blackboard for the other two agents to react to.

We do not need to store all the relevant rules for each episode in the episode node. Taking advantage of the structure of the curriculum-tree, the rule base of each agent in the episode consists of rules inherited from its ancestor nodes plus some resident rules which are particular to that episode. The curriculum-tree construct is actually a 'divide-and-conquer' mechanism: it helps the author of the system to decompose the curriculum into small episodes, and then focus on the development of rules that are particular to an episode.

Although Curriculum-Tree can largely reduce the complexity of developing social learning systems, it is still far from perfect. The major problem of Curriculum-Tree is that the framework in itself is too complex. Figure 3 shows the system architecture of the early attempt to develop social learning systems with Curriculum-Tree.

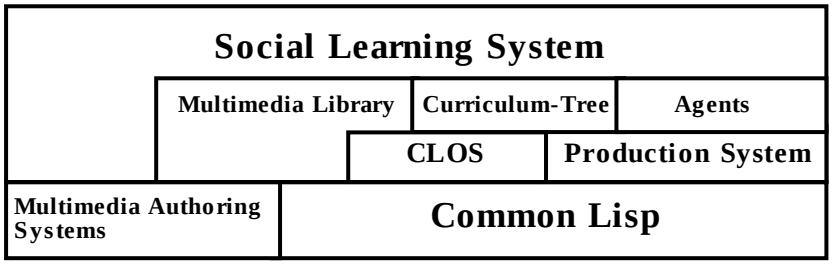

Figure 3. Lisp-based Architecture of Social Learning System

Common Lisp is the core of the programming environment. To simulate agents, a production system was written in Common Lisp for driving rules of behavior. The curriculum-tree is actually a tree-structure built upon an object-oriented extension of Common Lisp, CLOS. Like simulated agents, rules in Curriculum-Tree are also driven by the production system. To support multimedia, a multimedia library upon Common Lisp and CLOS is used as the interface to some multimedia authoring systems. Finally, the target system, a social learning system, is built upon all these components. As can be seen, the system architecture involves several sublanguages and the complexity makes it hard to develop and maintain programs of social learning systems.

Another problem of Curriculum-Tree is that since it is based on a production system, it inherits the maintenance, indexing, and efficiency problems of the production system. Besides the feeling of disorientation when working with hundreds of rules at the same time, there are

problems of complexity and efficiency both in constructing and running the rules. Some of the rule's conditions in one protocol of activity (part of the curriculum) are different from the rules in other protocols but some are the same. Moreover, rules with the same conditions may have different right hand sides. To distinguish rules, one might need to index the rules according to different parts of the curriculum. Such indexing causes complexity on the left hand sides of the rules which would be difficult to understand.

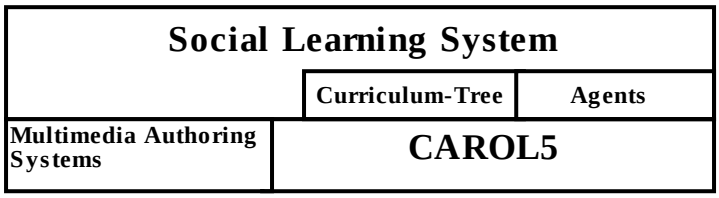

Figure 4. CAROL5-Based Architecture of Social Learning System

We believe that a well-designed dedicated programming language can eliminate most of these problems. This motivated us to design CAROL5 as an all-in-one programming language for developing social learning systems. Ideally, with CAROL5, we can simplify the architecture as shown in Figure 4. To achieve this, several requirements must be satisfied:

1. CAROL5 must provide a hierarchical knowledge sharing mechanism for building the curriculum-tree structure.
2. CAROL5 must support agent-oriented programming, so that it can be used to directly create multiple educational agents.
3. CAROL5 must support rule-based reasoning, so that it can directly drive rules in the curriculum-tree and agents without the help of a production system.
4. CAROL5 must support events, so that it can directly communicate with multimedia authoring systems via user-interface events.

Note that it is possible to eliminate the need of multimedia authoring systems if we build a full multimedia library in CAROL5. However, since building a multimedia library is too labor intensive, we prefer utilizing those of existing multimedia authoring systems to building our own. To avoid maintenance problems, we let CAROL5 and multimedia authoring system be loosely-coupled in the sense that they can only communicate via events, not shared common variables. In this way, the role of a multimedia authoring system is simply an interface builder that provides a graphical user-interface between the human student and CAROL5 programs.

We conclude that to fulfill the requirements of building the curriculum-tree and constructing agents for developing social learning systems, CAROL5 must support prototype-based programming (Liebermann, 1986; Ungar and Smith, 1987; Wegner, 1987), rule-based programming, and event-driven programming all at once. Table 1 shows how these programming techniques meet our need. The prototype-based model provides a simple but powerful object hierarchy for representing knowledge for both agents and Curriculum-Tree. Rule-based reasoning is appropriate for developing AI-intensive software like social learning systems. Finally, with the event mechanism, internal objects such as agents can handle graphical user-interface events. Also, event sending and handling is a straightforward communication mechanism among multiple agents.

The major challenge of designing CAROL5 is how to smoothly integrate these programming paradigms into a single programming language. In the following subsections, we will briefly introduce the design of CAROL5. A more formal semantics of the prototype-based model, methods, and procedures in CAROL5 can be found in another article (Wang and Chan, 1996).

Table 1. Programming Technique vs. Requirement of Social Learning System

Feature / Requirement	Prototype-Based Programming	Rule-Based Programming	Event-Driven Programming
Agent Knowledge Construction and Sharing	√		
Communication among Multiple Agents			√
Curriculum-Tree Knowledge Construction and Sharing	√		
AI-Intensive Software Development		√	
User-Interface Event Handling			√

2.1 Prototype-Based Model

It took us long to search for a knowledge representation best fit for the architecture of social learning systems, and finally we settle with the *prototype-based model* (Liebermann, 1986; Ungar and Smith, 1987; Wegner, 1987) in CAROL5 after several revisions. The idea of the prototype-based model came from that, to grasp new concepts, people usually start by creating concrete examples rather than abstract descriptions. We can use an object as a prototype for creating similar objects by saying how the new objects differ from the prototype. New objects may reuse the knowledge stored in the prototype as its default knowledge.

The prototype-based model was proposed as an alternative to the class-based model. Parallel to class inheritance, which is used to achieve class-based knowledge sharing, *delegation* (Liebermann, 1986; Stien, 1987; Dony, 1992) is usually adopted for prototype-based knowledge sharing. The basic idea of delegation is to forward requests that cannot be handled by an object to the prototype which it is based on

Note that in another sense, delegation can be interpreted as an extension mechanism (Liebermann, 1986; Dony, 1992). We can view object B that delegates to object A as an extension of object A. From this point of view, instead of saying object B delegates to object A, we can say that object B *encloses* object A. That is, we can treat object B as a whole object that contains some private knowledge plus knowledge borrowed from object A.

In CAROL5, the special object named `object` serves as the root of the delegation hierarchy and a place to store global properties. By default, a child object will enclose its parent. That is, the child object will possess all properties of its parent. A child object can override the property value of its parent by specifying a new value. In addition, a child object can add new properties. Thus, each object can have some unique attributes and methods of its own. The syntax of object definition in CAROL5 is:

```
<object-name> := <parent>[ <name1>: <value1>,
                           ...,
                           <namen>: <valuen> ].
```

where `<object-name>` is the name of the object, `<parent>` is the name of the parent object, $\texttt{<name}_1\texttt{>}$...$\texttt{<name}_n\texttt{>}$ are property names, and $\texttt{<value}_1\texttt{>}$...$\texttt{<value}_n\texttt{>}$ are primitive values such as integers, strings, etc. or other complex objects. For example, we can define objects `john` and `tom` as follows:

```
john := object[ name: ”John”,
                sex: ”male”,
                birthdate: [1980, 1, 1],
                school: ”NCU” ].


tom := john[ name: ”Tom”,
             birthdate: [1982, 2, 9] ].
```

The object `tom` overrides the attributes `name` and `birthdate`, and delegate the attributes `sex` and `school` to `john`. This means that tom's sex is also "male" and he studies at the school "NCU" too.

2.2 Rule-Based Methods and Procedures

In CAROL5, rule-based reasoning is well integrated into the prototype-based model; the bodies of methods and procedures are composed of rules. In this way, one can consider the name of a method (procedure) as an abstraction of the goals of the rules in the body of the method (procedure). The decision to adopt rule-based reasoning is orthogonal to the decision to use the prototype-based model. Our main concern is that rules are good for representing heuristics and procedural knowledge. This is very important for developing AI-intensive computer-assisted learning systems. Furthermore, from the computational point of view, most AI algorithms are actually processes of searching and data retrieval, and we know that rule-based reasoning is a powerful searching and data retrieval mechanism (Chan and Wang, 1993). Besides, our experience of representing an agent as a set of rules of behavior in our early curriculum-tree implementation also motivates us to adopt rules.

Rules

In CAROL5, a *rule* is in one of the following forms:

```
<why-part> => <do-part>, <consequence>
<why-part> =>> <do-part>, <consequence>
```

A rule consists of three phases, `why-part`, `do-part` and `consequence`. `Why-part` is the LHS (left hand side) of the rule, which is a sequence of predicates. The RHS (right hand side) of the rule includes `do-part` and `consequence`. `Do-part` is a sequence of statements with side-effects or local variable assignments. `Do-part` is optional. By 'statements with side-effects', we mean those involve I/O or update variable values or object states. `Consequence` is a single statement that may or may not have side-effects. CAROL5 will evaluate the statements in `why-part` one by one. If CAROL5 encounters a statement in `why-part` that is evaluated to be false, the rule evaluation will terminate and return false. If all the statements in `why-part` are evaluated to be true (any non-false value), CAROL5 will evaluate all the statements in `do-parts`, if any, one by one. Finally, CAROL5 will evaluate the `consequence` statement as the returned value of the rule. The execution of a rule is said to be 'successful' if the statements in the `why-part` of the rule were all evaluated to be true. Note that, in CAROL5, statements with side-effects can only appear in the RHS of a rule. In this way, programmers are enforced to separate pure predicates and side-effects. Thus CAROL5 can preserve the declarative style of rules and lead to more readable programming code.

There are two kinds of rules in CAROL5: *singular* rules (the separation mark '=>') and *repeated* rules (with separation mark '=>>'). These two kinds of rules are different in the evaluation process: a repeated rule will exhaustingly try all the possible cases in a backtracking search, like Prolog, and execute the RHS whenever the case is successful, while a singular rule will only attempt to execute the first successful case. Informally, we can interpret the '=>>' mark as 'for all' and the '=>' mark as 'there exists'.

For example, suppose the variable named `students` is a global variable the value of which is a list containing three students, say in order of John, Mary, and Tom; John's score is 85, Mary's score is 91, and Tom's score is 95. The following rule will check, among the three students, if there exists one student whose score is greater than 90.

```
S in students, S.score > 90 => print(S.name).
```

Note that CAROL5 enforces the programmer to distinguish global variables from local variables in the hope that he/she will be more careful and conservative when deciding to use global variables. In CAROL5, the name of a global variable must start with a lower-case letter, while the name of a local variable must start with an upper-case letter. In this example, `S` is an unbounded local variable. The statement `S in students` will retrieve each student from the list one by one. For each student, the statement `S.score > 90` is a predicate for checking whether the student's score is greater than 90. If the predicate is true, then the side-effect statement `print(S.name)` will print the student's name. Since this rule is a singular rule, only the name of the first student whose score is greater than 90 will be printed (i.e., Mary). On the other hand, if we change it into a repeated rule as follows, both Mary and Tom's names will be printed since both of their scores are greater than 90.

```
S in students, S.score > 90 =>> print(S.name).
```

The evaluation of statements in `why-part` (LHS) is a kind of unidirectional pattern-matching. That is, a statement with unbounded variables will be interpreted as an intention of data-retrieval. On the other hand, if all variables in a statement are bounded, the statement will be interpreted as a predicate. For example, if we change the rule as follows:

```
S is mary, S in students, S.score > 90 => print(S.name).
```

when the statement `S is mary` is evaluated, since `S` is still unbounded, the statement is a data-retrieval for letting `S` be bounded to `mary`. However, when the statement `S in students` is evaluated, since `S` is already bound to `mary`, the statement becomes a predicate for checking if `mary` exists in the list of students.

Methods

We can view a *method* as an abstraction to combine multiple rules together to represent different cases respectively. The syntax form of method definition is:

```
<object-name>.<method-name> :=
     method(<arg1>, ..., <argm>) <control-type>
     {
       <rule1>;
       ...;
       <rulen>
     }.
```

For example, suppose `parents` is a property containing a list of one's parents, we can define the following method to check whether `X` is object `john`'s ancestor:

```
john.ancestor? := method(X)
{
  self[parents: S], X in S => t;
  self[parents: S], Z in S, Z.ancestor?(X) => t
}.
```

The statement `self[parents: S]` will bind the local variable S to the value of the property named `parents`. In CAROL5, `self` is a reserved keyword which always refers to the object that is invoking the method; the symbol `t` represents the Boolean true, while the symbol `f` represents the Boolean false. The first rule examines whether `X` is one of `john`'s parents. If so, the predicate `john.ancestor?(X)` will be true. The second rule checks whether `X` is an ancestor of one of his parents. By definition, an ancestor of one's parent is one's ancestor, too. Note that, in CAROL5, if all rules in a method fail to be executed completely, the false value `f` is returned by default. Thus, if `X` does not satisfy both rules, the predicate `john.ancestor?(X)` will be false. Note that we must use singular rules in defining the method `ancestor?` because we only concerned with whether the passed arguments satisfy any one of the cases specified by the rules.

The order of rules in a method is important. By default, when a method is invoked, CAROL5 will execute the rules in the body of the method one by one, until the first successful rule is encountered. By default, the environments among the rules are irrelevant, that is, the bindings of local variables in a rule will not be propagated to the next rule. Programmers can override the default method body control by adding the '^' or '&' control-type specifier when defining methods. A method with the '^' specifier causes CAROL5 to execute all the rules in its body, and the environments among the rules are still irrelevant. A method with the '&' specifier will cause CAROL5 to execute all the rules in its body but the environments among the rules are related, that is, the bindings of local variables in a rule will be propagated to the next rule.

To illustrate the usefulness of these specifiers, let us examine the method for constructing a reverse of a list:

```
list.reverse := method() &
{
  t => RList := [];
  I in self =>> RList = cons(I, RList)
}.
```

where `I` and `RList` are local variables. Note that, in CAROL5, the operator '`:=`' means "define", while '`=`' stands for "update"; both '`:=`' and '`=`' will assign a value to a variable. We distinguish these two operators because we want programmers to declare variables before using them. In this method, the first rule initializes the value of the variable `RList` to be an empty list; the second rule can be viewed as a loop which retrieves each element of the list one by one into the variable `I` and then accumulates them into the variable RList in the reverse order. The '&' specifier is necessary because we want to reserve the value of RList between different rules and among iterations. Without the '&' specifier, the implementation of such iterations would become rather complex in a rule-based language.

Both the '^' and '&' specifiers can be further combined with the 'a' control-type specifier to cause CAROL5 to accumulate all the result of the successful rules into a list. The following example shows the effect of the 'a' specifier:

```
john.foo := method() ^a
{
  self[parents: S], P in S, P[name: N] =>> N;
  self.age() > 60, self[name: N] => N;
}.
```

The first rule retrieves the names of both John's parents, and the second rule retrieves John's name. Here we use the '^' specifier not the '&' specifier because we want the binding values of the variables named N in both rules to be irrelevant. The 'a' specifier will accumulate all those retrieved names. For example, if John's age is greater than 60, and John's parents are Tom and Mary, then the method invocation `john.foo()` will return the list `["Tom", "Mary", "John"]`. Without the 'a' specifier the returned value will be the string `"John"`.

Procedures

CAROL5 supports both *methods* and *procedures*. Although they have similar syntax forms, the semantics of methods and procedures are different: methods use dynamic binding, while procedures adopt static binding. The syntax form of procedure definition is:

```
<object-name>.<procedure-name> :=
     procedure(<arg1>, ..., <argm>) <control-type>
     {
       <rule1>;
       ...;
       <rulen>
     }.
```

To summarize, besides supporting objects, there are many differences between CAROL5 and other rule-based programming languages. First, unlike other rule-based languages in which all rules are flatly distributed in programs, CAROL5 uses methods and procedures as abstraction mechanisms to organizes rules in a better maintainable form. Second, CAROL5 does not adopt an inference engine such as the recognize-act cycle, but CAROL5 does provide a backtracking search and unidirectional pattern-matching on evaluating the LHS of a rule. This is why programmers can still take some important advantages of rule-based programming while implementing AI algorithms. Third, CAROL5 provides two kinds of rule control structures (singular and repeated rules) which, in effect, allows programmers to explicitly specify universal (for all) and existential (there exist) conditions. Fourth, CAROL5 provides three kinds of method body control ('&', '^', and 'a') which may allow programmers to express their thinking in more intuitive way. Finally, CAROL5 allows programmer to use side-effects in a rule with the enforcement that only the RHS of the rule can have side-effects, so that programmer can preserve the declarative-style of rule-based programming.

2.3 Event-Driven Programming

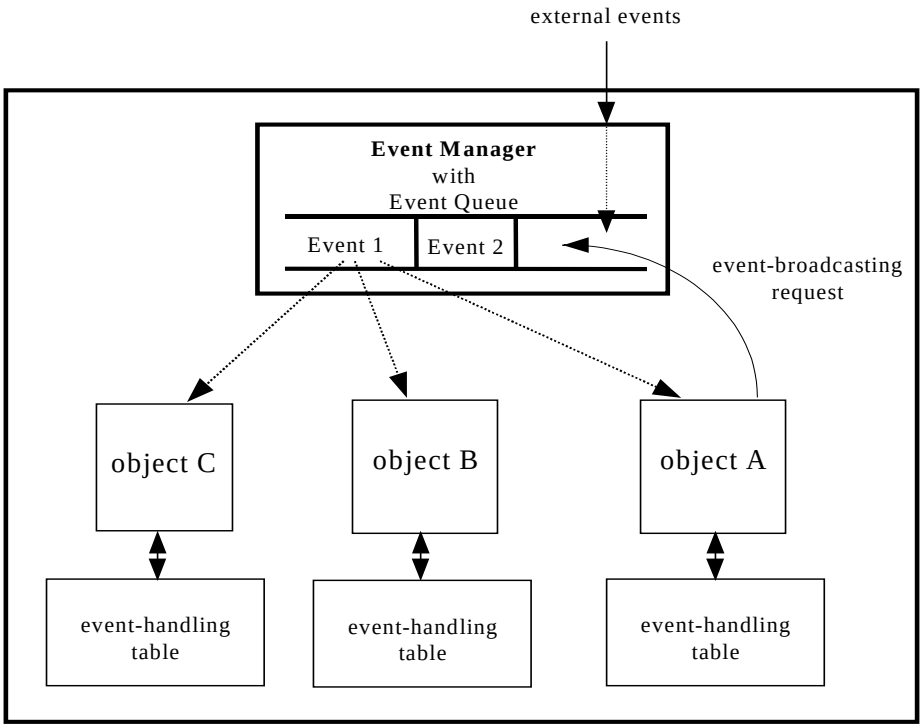

Figure 5. Event Broadcasting as Mechanism of Object Communication

In most traditional object-oriented programming languages, objects communicate by message-passing, which is in fact method invocation. In CAROL5, we adopt event broadcasting, which is similar to the way we communicate in the real world, as the mechanism of object communication. Event-broadcasting is more general than message-passing because message-passing can be viewed as a special case of event-broadcasting with only one event receiver. Figure 5 shows the event-broadcasting mechanism of CAROL5.

In CAROL5, there is a special object named *event manager* which is in charge of mediating event-broadcasting among objects. Since an event can occur while another event is processing, the event manager will schedule unprocessed events in a queue called *event queue*. The current implementation of CAROL5 adopts a simple-minded first-in-first-out mechanism for scheduling events. Since event manager is in charge of scheduling events, it resembles the agent scheduler of a blackboard system that we used to implement an episode node in our early curriculum-tree model.

Event-Broadcasting

Each object can have some event-broadcasting methods or event-broadcasting procedures. An event-broadcasting method is a method with at least one event-broadcasting command in its body and similarly for a procedure. Note that an event-broadcasting command can only be placed in the RHS of a rule, because it may cause some side-effects when executed. The form of an event-broadcasting command is:

```
broadcast(<event-name>, <event-datum>, <receiver-list>)
```

where `<event-name>` is the name of the event, `<event-datum>` is any kind of CAROL5 value sent along to the receivers, and `<receiver-list>` is a list of objects, which is specified as the receivers of the event.

An event-broadcasting method or procedure of an object will run as it usually does for a normal method or procedure, except that whenever an event-broadcasting command in its body is executed, the object will send an event-broadcasting request to the event manager. An event-broadcasting request is a four-tuple of the form:

```
[event-sender, event-name, event-datum, receiver-list]
```

where `event-sender` is the name of the object sending the event-broadcasting request, `event-name`, `receiver-list` and `event-datum` are taken from the arguments of the event-broadcasting command.

After receiving an event-broadcasting request, the event manager will schedule the request in the event queue in a first-in-first-out fashion. When the event queue is not empty, the event manager will broadcast the first event in the queue to each object listed in the receiver-list. Note that when the receiver-list of an event-broadcasting is empty, it represents a request that broadcasts the event to all objects in the system rather than no receivers. When broadcasting an event, some of the receivers of the event may be busy in processing other jobs, and thus they cannot receive the event immediately. In this situation, the event manager will keep trying to broadcast the event to those busy objects until all of them receive the event. After the event has been received by all receivers, the event manager will remove the first event-broadcasting request, and start processing the next event-broadcasting request in the event queue.

Event-Handling

An event forwarded from the event-manager to the receiver is a three-tuple of the form:

```
[event-sender, event-name, event-datum]
```

where each element in the tuple is taken from the event-broadcasting request in the event queue. To handle a received event, an object must have a corresponding event-handler. An event-handler is either an *event-handling method* or an *event-handling procedure*. Any method

(procedure) with only two formal arguments can be used as an event-handling method (procedure). The purpose of these two formal arguments is for passing the `event-sender` and the `event-datum` to the event-handler.

In CAROL5, each object has its own *event-handling table*, which is a list of *event-handling records*. An event-handling record is a three-tuple form:

```
[event-sender, event-name, event-handler]
```

where `event-sender` is the name of the object who may originate the event, `event-name` is the name of the event, and `event-handler` is the name of the event-handler which the object will use to react to the event.

Once an event is received, an object will try to match the event with each event-handling record in its event-handling table. We will say that an event-handling record matches an event, if the event-handling record has the same event-sender and event-name with the event. Once the object finds a matched event-handling record, then the object will execute the event-handler specified in the event-handling record. When the event-handler is executed, it is guaranteed that its first argument will be bound to the event-sender, and the second argument will be bound to the event-datum.

If there is no matching event-handling record in its event-handling table the object will just ignore the event. Also, it is possible that an object broadcasts another event during the process of handling an event, because there may exist some event-broadcasting commands in running the event-handler. In such situation, if any of these event-broadcasting commands is executed, a new event-broadcasting request will be sent to the event queue of the event manager, causing chains of actions and reactions among objects.

Finally, an object can maintain its own event-handling table by using the following two commands:

```
addHandler(<event-sender>, <event-name>, <handler>)
deleteHandler(<event-sender>, <event-name>, <handler>)
```

where `addHandler` is used to add an event-handling record, and `deleteHandler` is used to delete an event-handling record.

External Event

In addition to mediating the communication among CAROL5 objects, the event manager also acts as a mediator between CAROL5 objects and external objects, which may include external multimedia objects, electronic devices probing changes in the external environment, or even the human user itself. When the event manager receives an external event, it will convert the external event into the form of an event-broadcasting request, and then put the request into the event queue.

Objects as Finite State Machines

An object has *state* and *behavior*. (Note that an object, of course, also has its *identity*. However, this paper has no concern with issues related to object identities.) The state of an object in CAROL5 comprises all of the current value of the object's properties. The behavior of an object in CAROL5 is determined by how it acts and reacts in terms of event occurrences and its state changes. When an event occurs, an interested object will invoke the corresponding event handler in response. The execution of the event handler may change values of some of the object's properties, thus causing the objects to transit from one state to another. This gives rise to the idea that we can view an object as a finite state machine. The inputs to the machine are events. That is, the machine will transit from state to state when events occur.

Viewing objects as finite state machines is not a new idea. Many proposed methodologies of object-oriented design such as Booch's method (Booch, 1991) have used *state transition diagrams* for modeling how instances of individual classes behave dynamically. However, it seems that there is no programming language the design of which is directly based on the notion

of objects as finite state machines. Thus, there is usually a gap between software design and implementation. For example, if a programmer codes software in a traditional object-oriented programming language, he will encounter a problem that he need to map events in state transition diagram into corresponding message passing because objects in that language communicate via message passing (or method invocation) not events. On the other hand, in CAROL5, since objects communicate via events, the dynamic behavior of objects can be precisely described in state transition diagrams. That is, with CAROL5, programmers can create objects as though they are constructing corresponding finite state machines.

In the following context, we will use state transition diagrams to explain the behavior of objects. The notation of our state transition diagrams is a modified version of Booch's:

1. A node, drawn as a circle icon with the state name on it, in a state transition diagram represents a single state.
2. A state transition is drawn as a directed line from the initial state to the new state (Figure 6). Such a line must be labeled above the line with an event that causes the state transition and below the line with a handler that the object uses to perform the state transition in response to the event.

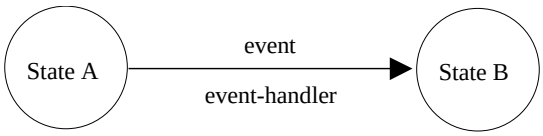

Figure 6. A state transition from state A to state B

As we said before, an event-handler itself may be an event-broadcasting method or procedure. Thus, sometimes, an object may broadcast some events to other objects as a side-effect or consequence of invoking the event-handler. In such a situation, a sequence of corresponding event-broadcasting requests will be added below the event-handler (Figure 7).

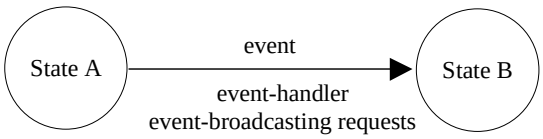

Figure 7. A state transition with an event-broadcasting request

Sometimes, the user may directly command an object to perform some actions such as invoking a method or procedure and thus cause the object to transit from one state to another. In such situation, the state transition will be drawn as shown in the template shown in Figure 8.

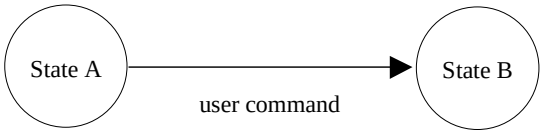

Figure 8. A state transition caused by user initiated command

Finally, the user command may trigger some events from the object. In such case, the template drawn in Figure 9 is used.

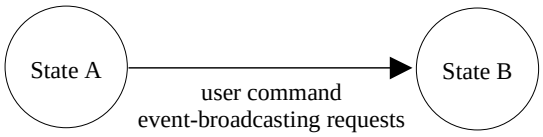

Figure 9. An user initiated command with an event-broadcasting requests

Example

A simple example illustrating how the event-broadcasting mechanism works is provided as follows. This example simulates an episode happening in a classroom. In the classroom, there is a geography teacher and three students. The teacher is quizzing his/her students. The scenario may proceed as follows:

Teacher: Where is the capital of Australia?
Student1: Vienna.
Teacher: Nope, Vienna is the capital of Austria.
You must not confuse Australia with Austria.
Student2: Sydney.
Teacher: Nope, Sydney may be the most famous city of Australia, but it is not the capital.
Student3: Canberra.
Teacher: Correct.

We will create an object named `teacher` to simulate the teacher, and three objects named `student1`, `student2`, and `student3` to simulate student1, student2, and student3 respectively. The `teacher` object is a finite state machine with the state transition diagram shown in Figure 10. In the notation of our state transition diagrams, a start state is denoted by drawing the state icon with an unfilled double line, while a stop state is denoted by drawing the state icon with a filled double line.

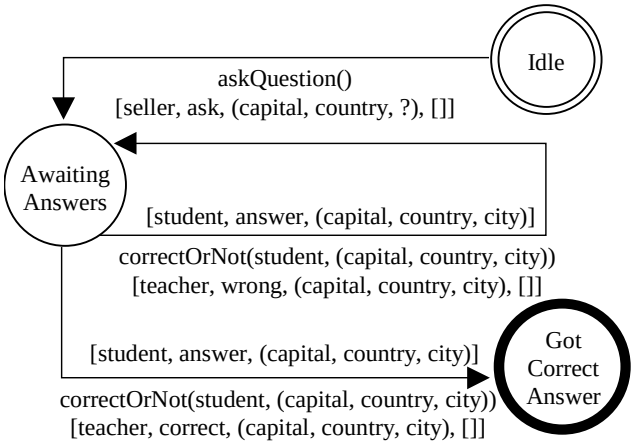

Figure 10. State transition diagram of seller

Initially, `teacher` is in an "Idle" state and he will stay there unless we command it to invoke a method named `askQuestion` to randomly choose a country as the target for quizzing the students. Note that the invocation of method `askQuestion` will cause `teacher` to broadcast an event named `ask` to what the correct combination of "`(capital, country, ?)`" is. After that, `teacher` will change to an "Awaiting Answers" state for awaiting each student object to send an `answer` event to answer the question. Once `teacher` receives an `answer` event from a student object, it will handle this event with the event-handler named `correctOrNot` to decide whether the combination of "`(capital, country, city)`" given by the student object is the correct answer. If the answer given is wrong, `teacher` will broadcast an event named `wrong` to tell all student objects that the combination of "`(capital, country, city)`" is wrong. If the answer is correct, it will send a event name `correct` to tell all student objects that the combination of "`(capital, country, city)`" is correct, and then finally change to an "Got Correct Answer" state.

The `student1` object is a finite state machine with the state transition diagram shown in Figure 11.

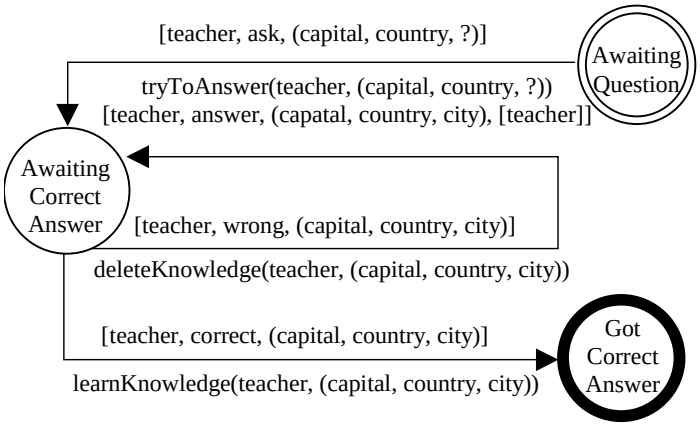

Figure 11. State transition diagram of buyer

Initially, `student1` is in an "Awaiting Question" state for awaiting `teacher` to broadcast a `ask` event. Once he receives a `ask` event from `teacher`, he will invoke an event-handler named `tryToAnswer` to try to answer the question. The invocation of method `tryToAnswer` will cause `student1` to send an event named `answer` to `teacher`. A combination of "`(capital, country, city)`" which represents the answer of `student1` will be sent along with the `answer` event. After sending the `answer` event, `student1` will change to an "Awaiting Correct Answer" state for awaiting `teacher` to tell it that its or other's answer is correct or wrong. If `teacher` say that the combination of "`(capital, country, city)`" is wrong by broadcasting an event named `wrong`, `student1` will invoke an event-handler named `deleteKnowledge` to try to delete the combination from its knowledge base. If `teacher` say that the combination of "`(capital, country, city)`" is correct by broadcasting an event named `correct`, `student1` will invoke an event-handler named `learnKnowledge` to try to store the combination in its knowledge base, and then it will finally change to a "Got Correct Answer" state.

Note that `student2` and `student3` both have the same behavior as `student1`, thus they have the same state transition diagram. In CAROL5, we do not need to create `student2` and `student3` from scratch. With the help of prototype-based delegation, we can simply let both of `student2` and `student3` be extensions of `student1` with different values in some properties.

The implementation of these finite state machines as objects in CAROL5 is quite straightforward because objects in CAROL5 communicate via events by nature. The source code of this geography-quizzing example in CAROL5 is available in

http://www.lisa.src.ncu.edu.tw/~cheng/CAROL5/eventExample.html

To perform the episode, we first let `teacher` call the method `askQuestion()`, then a series of chain reactions will begin among `teacher` and three `students`. Suppose `teacher` happen to choose Australia as the target country, it will broadcast the following event to all objects including `student1`, `student2`, and `student3`:

```
[teacher, ask, (capital, Australia, ?)]
```

After receiving this event, each student object will execute the method `answer` to handle this event. Since `student1` confuses Australia with Austria, it may send following event to the `teacher`:

```
[student1, answer, (capital, Australia, Vienna)]
```

`Student2` thinks it is Sydney, and thus he may reply with:

```
[student2, answer, (capital, Australia, Sydney)]
```

`Student3` does know the answer, and thus he will send the following event to `teacher`:

```
[student3, answer, (capital, Australia, Canberra)]
```

After receiving each `answer` event, `teacher` will handle it with the method `correctOrNot` to decide whether each answer is correct. As a result, `teacher` will reply to the three answer with the following events respectively:

```
[teacher, wrong, (capital, Australia, Vienna)]
[teacher, wrong, (capital, Australia, Sydney)]
[teacher, correct, (capital, Australia, Canberra)]
```

Each student object then execute the method `deleteKnowledge` to handle both the events named `wrong`, and finally execute the method `learnKnowledge` to handle the event named `correct`.

Note that, as shown in this example, although the event manager plays an important role as the mediator of communication among objects in CAROL5, programmers usually need not to be aware of its existence when they are using the event-broadcasting command in CAROL5, just as we usually need not to care about the fact that we use air as the mediator when we talk to other people.

2.4 Multi-Agents in CAROL5

Our main goal of designing CAROL5 is to support the construction of educational agents. AOP support in CAROL5 is a result of integrating prototype-based programming, rule-based programming, and event-driven programming. In this subsection, we will explain how CAROL5 supports AOP.

From the viewpoint of programming, an agent is simply an autonomous object that performs some task based on its own domain knowledge. Indeed, autonomy is the key characteristic that distinguishes an agent from a normal object. By autonomy, we mean that an object cannot force another object to perform a service or any action unless it is willing to accept such a request. The magic to achieve autonomy of an agent is to let the agents communicate via speech acts, such as inform, query, answer, request, etc. In this way, we can view the world of agents as an electronic ecosystem (Maes, 1995b), in which they communicate in a way similar to how we communicate in real world.

Based on the above analysis, we can conclude that, to construct agents, we have to deal with two design issues:

1. How do we construct an agent's domain knowledge?
2. How do we support speech acts among agents?

The prototype-based model and rule-based programming style make CAROL5 an easy and effective language for constructing knowledge, in the sense that the state of an object is actually its factual knowledge, while the rule-based methods and procedures represent its procedural knowledge. Also, since part of the knowledge of the world is objective, a model of agent knowledge construction has to incorporate a mechanism of how agents share knowledge. Fortunately, prototype-based delegation is a powerful knowledge sharing mechanism. Thus, we do not need to invent another one.

As for the second issue, the event broadcasting and handling mechanism in CAROL5 is in fact a general form of speech acts. From this point of view, *programmers can designate some objects in CAROL5 to be agents if they possess this event broadcasting and handling ability*. As we have seen in the seller-buyer example in the previous section, the communications among `seller`, `buyer1`, `buyer2`, and `buyer3` are actually a process of various speech acts. Thus, they are agents by our definition.

The current implementation of CAROL5 supports multiple-agents in a standalone machine. The world of agents is managed by the Event Manager which mediate all communications among agents.

Active Agents

Among agents, some of them may be active objects, which means that they can exhibit some behavior without being operated upon by human or other objects. Like a human user, an active object can initiate activities in its own right, and thus it can participate in actively accomplishing tasks such as assisting students in learning.

To support active objects, we introduce the notion of *missions* in CAROL5. A mission is a method or procedure that an object will automatically execute when it is idle, that is, when it is not handling any event for a period of time. Note that a mission must have no formal argument, since no actual argument will be passed when it is executed. Thus, any method or procedure of an object with no formal argument can be set as a mission.

We say that an agent is active if it has some missions. In CAROL5, we can set an agent a mission by letting it execute the `setMission` command. If necessary, an agent can execute the command `unsetMission` to unset a mission. The form of setMission and unsetMission is shown as follows:

```
setMission(<mission-name>)
unsetMission(<mission-name>)
```

where `mission-name` is the name of the method or procedure which is to be set or unset as a mission.

It is possible that an agent is set to multiple missions. In such a situation, the order of these missions set is significant. When an agent executes its missions, it will execute them one by one in the order we have set them.

Figure 12 shows the flow of activities of an agent. Events and missions both play important roles in the activities of an agent. Initially, an agent will wait for events for a period of time. During the event-waiting, if it receives any event, it will examine whether it is interested in handling that event, that is whether it has a matched event-handling record. If it is interested, it will then execute the corresponding event-handler. However, if it does not receive any event after the period of event-waiting, it will start to execute all it missions. Once an agent starts to execute its missions, it will be unable to accept any event until all its missions are executed.

In CAROL5, events have higher priority than missions. That is, an agent will wait for events first, and then execute its missions if there is no event has occurred. We give events higher priority because in this way agents will be more sensitive to external events, and thus the user can get responses from agents more quickly. As can be seen, there is a trade-off between how sensitively an agent responds to events and how often an agent will execute its missions. If necessary, programmers or users can configure the CAROL5 interpreter to get a reasonable length of this event-waiting period.

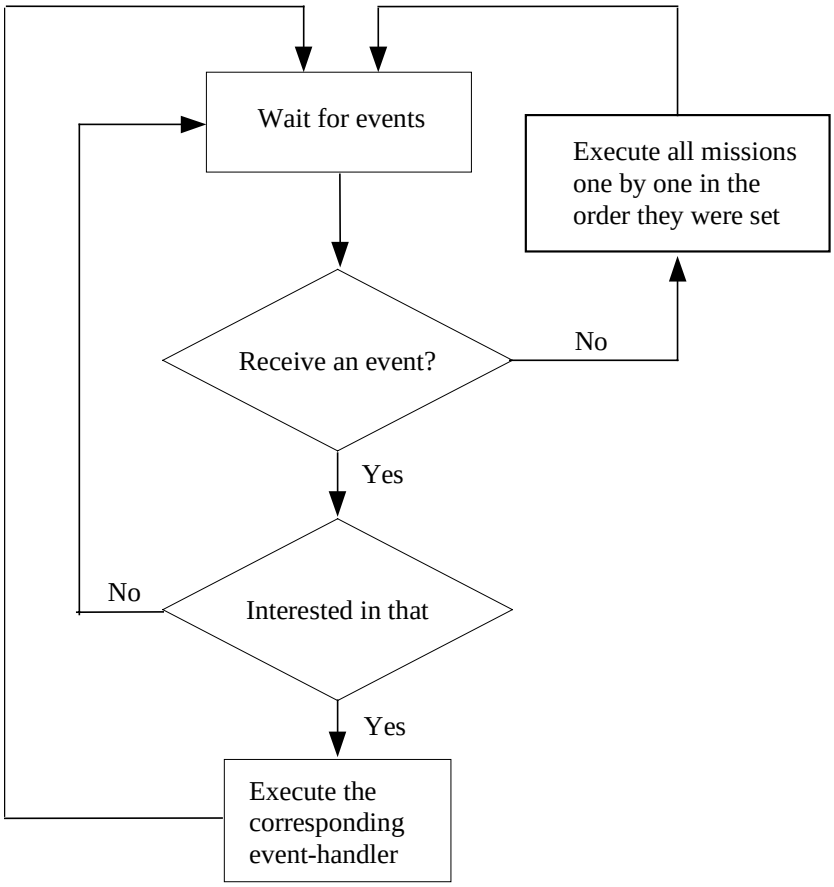

Figure 12. The flow of activities of an agent

Events and missions are two orthogonal features, although they are often used together. In CAROL5, any object that communicates with others via events can be viewed as an agent, so there can exist agents without missions. Agents without missions are always passively waiting for events to response to, so we call them *passive agents*. Agents with missions are *active agents*. For example, `teacher` in the seller-buyer example is a passive agent because he will always stay at the "Idle" state unless users initiate a `teacher.askQuestion()` command to let it ask a question . However, if we set the method `askQuestion` as a mission of `teacher` by executing the following command:

```
teacher.setMission(askQuestion)
```

then `teacher` will become an active agent. From then on, since `askQuestion` is `a mission of teacher`, it will periodically and automatically execute the method `askQuestion` to randomly ask some question.

3 EXAMPLE: THREE'S COMPANY SYSTEM

Since the notion of agents is the central part of social learning models, it should be straightforward to implement social learning systems in an agent-oriented programming language such as CAROL5. Three's Company (Lin, 1993) is a direct extension of the student-teacher-companion model of Integration-Kid by including one more companion agent, so that there are three students (one human students and two simulated students) learning together. An interesting issue raised by the Three's Company system is the possible variations of the performance patterns among three students and their effects on the human student. For example, if the performance of the human student lies between that of the two companions, how would his/her motivation be affected? What would happen if their pattern of performance varies during

the learning process? How should we control the competence of the companions in order to facilitate the human student's motivation? In this section, we will illustrate how to implement a simplified version of Three's Company in CAROL5.

3.1 Overview of Simplified Three's Company System

The original Three's Company is a system teaching recursion in Lisp. However, since our purpose is only to demonstrate how AOP techniques can be helpful for implementing social learning systems, we will change the domain to a simpler one, letting students practise adding and subtracting two integers. For convenience, we will simply refer to this simplified version as 'Three's Company' hereafter.

In Three's Company there is a teacher who is in charge of controlling the progress of the curriculum, asking questions, checking student answers, and evaluating user performance. In addition, there are two simulated students, named John and Mary. John is simulated as a more capable learning companion, and Mary is basically the same as John except that she is a less capable learning companion.

When using the system, the user will go through three learning phases: introduction, practice, and quiz. In the introduction phase, the system presents a movie introducing some concepts to the user. However, for simplicity, we assume that the user has learned the necessary concepts of addition and subtraction from a textbook or class before using the system, and the introduction movie is omitted in this simplified version. Instead, the system will only show a welcome message on the screen. In the practice phase, the user will practise by answering some question given by the teacher. While the user is practising, the teacher evaluates the performance of the student and continues to ask questions until the user is qualified to enter the quiz phase. In the quiz phase, the teacher will continue to ask twenty questions. However, unlike the practice phase in which the user was alone, the two simulated students, John and Mary, will now compete with the user in the quiz phase. The scoring rule is simple: for each question, the first student who gives the correct answer scores one point. Finally, the student who gets the highest score is the winner.

3.2 Implementation

The implementation of Three's Company in CAROL5 is quite straightforward because the design of the language is tailored to the design of this kind of systems with multiple agents and well-defined curricular structure. The source code of Three's Company in CAROL5 is avaiable in

http://www.lisa.src.ncu.edu/~cheng/CAROL5/3Company.html.

The implementation comprises two main tasks: constructing a curriculum-tree and creating educational agents.

Curriculum-Tree

The curriculum-tree of Three's Company is shown as Figure 13, which is a sub-tree of the prototype–based delegation hierarchy in CAROL5. Each episode in the curriculum-tree is an individual CAROL5 object.

The episode node named `three` is the root of the curriculum-tree. It is an object containing knowledge, such as the name of the system or a method for ending an episode, shared by all episodes.

The three learning phases are directly implemented as three episode nodes. The `introduction` episode is a child of the root episode whose behavior is shown as the state transition diagram in Figure 14.

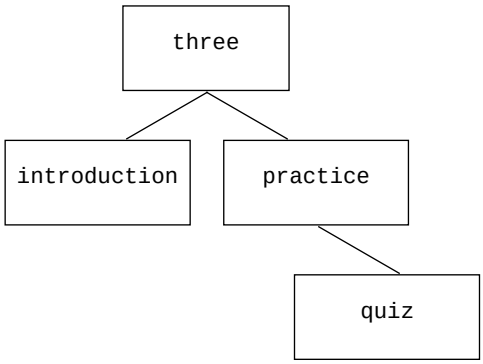

Figure 13. Curriculum-Tree of Three's Company

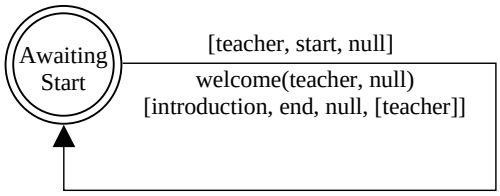

Figure 14. State transition diagram of introduction episode

The `introduction` episode is an object whose only task is to show a welcome message. Whenever it receives a `start` event from `teacher` it will show the following welcome message on the screen:

```
Welcome to Three's Company. This system is for
reinforcing your knowledge on integer addition and
subtraction. First, in the practice phase, the
teacher will ask you at least five questions until
you are qualified to take the quiz. Then, in the quiz
phase, the teacher will continue to ask you twenty
questions. Be prepared that two other students, John
and Mary, will compete with you in the quiz phase.
```

After showing the welcome message, it will send an `end` event to `teacher` to inform him of the end of the `introduction` episode. As shown in this case, we will usually use a special value name `null` whenever there is no event datum needed to be passed between the event sender and receivers.

The `practice` episode is also a child of the root episode. It contains a question base which comprises many questions for integer addition or subtraction. The teacher will retrieve questions from the question base for asking students. The behavior of the `practice` episode is shown as the state transition diagram in Figure 15.

Initially, the practice episode is in an "Awaiting Start" state for awaiting a `start` event from `teacher`. Once the `start` event is received, it will retrieve a question from its question base, and then send the question to `teacher` via a `sendQuestion` event. After sending the question, it will stay at the "Retrieving Questions" state to wait for some `moreQuestion` events sent from `teacher` for retrieving more questions. Finally, if it receives a `noMoreQuestion` event sent from `teacher`, it will invoke a method named `end` to end itself and then send an `end` event to `teacher`.

The `quiz` episode is a child of the `practice` episode. Like the `practice` episode, the task of the `quiz` episode is to provide a question base to the teacher, too. Since it is a child of

the `practice` episode, it reuses the question base of the `practice` episode. Furthermore, the behavior of the `quiz` episode is also exactly the same as the `practice` episode. Although it is possible to let the `practice` episode handle the question retrieving requests from the teacher in both the practice and quiz phases, the adding of the quiz episode makes the control of curriculum flow clearer. Thus, it is worth adding this additional episode. Fortunately, with the help of the prototype-based delegation mechanism, we do not need to create this episode from scratch.

The question base shared by the `practice` and `quiz` episodes is a question generator which will dynamically generate two random integers and randomly choose the operation of addition or subtraction for constructing a question. However, we treat this question generator as if it was a real database containing questions. In this way, we can easily replace it with a question base for a new domain in the future.

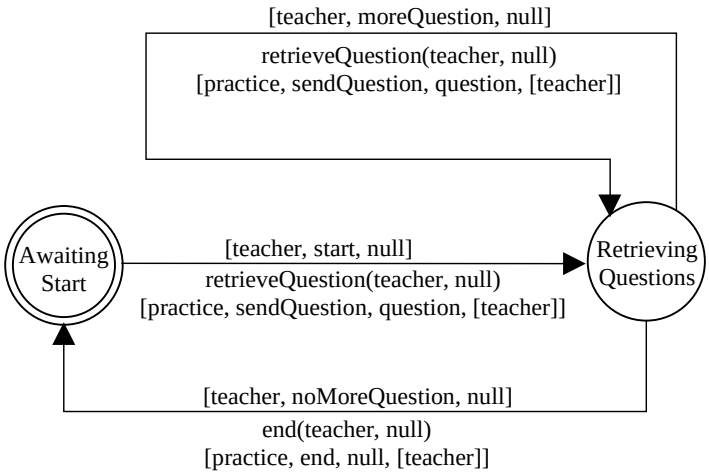

Figure 15. State transition diagram of practice episode

Educational Agents

In this subsection, we will present how we create educational agents for Three's Company. As shown in Figure 16, the teacher and the two simulated students, John and Mary, are implemented as individual agents named `teacher`, `john`, and `mary` respectively. Although agents in CAROL5 communicate in a way similar to speech acts among human beings, their native language is composed of events, which is too low level for a human user to understand. Thus, there is a need for translating events into dialogues in natural language when they are sent to the human user. In our implementation, instead of adding some special code to let every agent perform the translation by themselves, we add an additional `user` agent for acting as a translator between agents and the human user.

As shown in Figure 16, whenever an agent wants to communicate with the human user, it will send an event to the `user` agent instead. The `user` agent will then translate the event into a corresponding dialogue in natural language and wait for the human user's response. After the `user` agent receives the human user's response, it will pack the response into an event for sending back to the original event sender. Since other agents are facing the `user` agent rather than the real human user, from their viewpoint, the human user is capable of communicating in events like a normal agent. The advantage of this approach is that it allows all agents to communicate in a uniform way, and thus they usually do not bother to distinguish the human user from agents. This uniformity not only make the flow control of the program clearer but is also helpful for increasing readability and maintainability of the program code. Furthermore, this approach also makes the job of designing and modifying the human interface easier since all the related methods and procedures are centralized in the `user` agent. Now, let us use state transition diagrams to describe each agent's behavior:

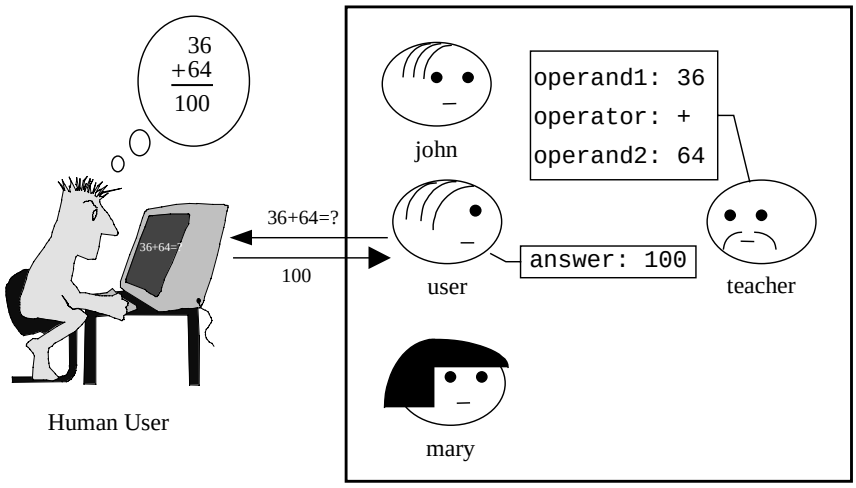

Figure 16. Human user and agents in Three's Company System

I. *teacher*:

In Three's Company, `teacher`'s behavior is the most complicated one. Thus, we will first outline its behavior as the state transition diagram shown in Figure 17. Later, we will detail `teacher`'s behavior in the practice and episode phases.

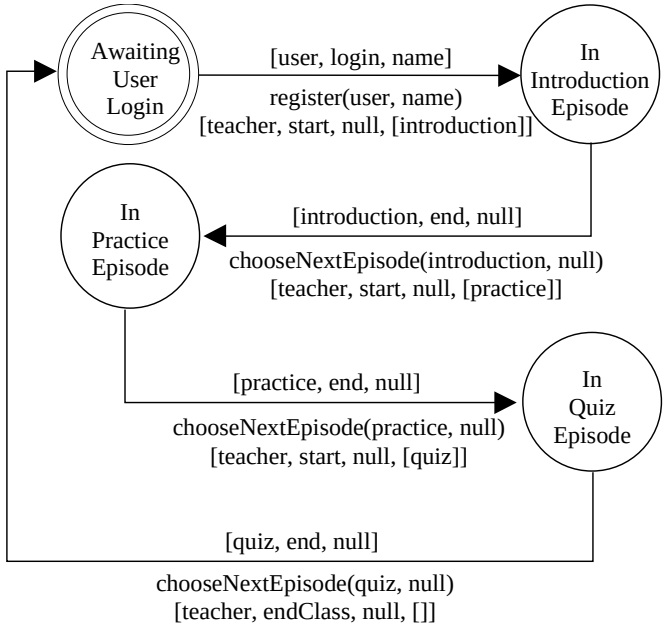

Figure 17. Outline of `teacher`'s state transition diagram

Basically, `teacher`'s task is to control the curriculum flow. Thus, the outline of its activity is to lead the system and other agents to walk through the three learning phases. At the beginning, `teacher` is in an "Awaiting User Login" state for awaiting a `login` event from the `user` agent for informing that the human user is on-line. When `teacher` receives a `login` event, it will invoke the `register` method to register the name of the human user, and then send a `start` event to activate the `introduction` episode and then change to an "In Introduction Episode" state. At the end of each activated episode, `teacher` will receive an `end` event from the episode. The `end` event will cause `teacher` to execute a method named `chooseNextEpisode` and send another `start` event to activate the next episode. Thus, at

the end of the `introductory` episode, the `practice` episode will be activated. The `quiz` episode comes after the `practice` episode. Finally, `teacher` will go back to the "Awaiting User Login" state for awaiting another `login` event.

The "In Practice Episode" and "In Quiz Episode" are actually two *meta-states*, because there are some learning activities inside them. By meta-state, we mean a state that contains several sub-states. Thus, the "In Practice Episode" state and "In Quiz Episode" state can be further divided into several sub-states respectively. The detail of `teacher`'s transition diagram in the `practice` episode is shown in Figure 18.

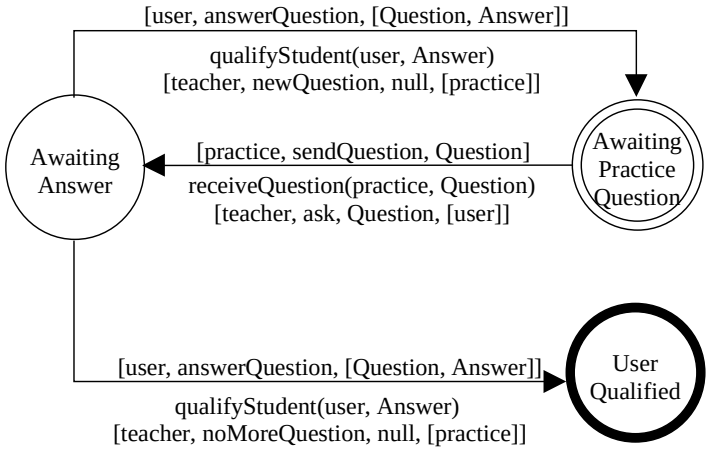

Figure 18. Teacher's state transition diagram in practice episode

In the practice phase, `teacher`'s task is to ask user some questions and qualify him/her for entering the quiz phase. When the `practice` episode starts, `teacher` is in an "Awaiting Practice Question" for awaiting the `practice` episode to send it a `question` by a `sendQuestion` event. Once a `sendQuestion` event is received, `teacher` will invoke a method named `receiveQuestion` to receive the question, and it will ask the user to answer the question by sending an `ask` event to the `user` agent. After sending the `ask` event, `teacher` will stay in an "Awaiting Answer" state for awaiting the `user` agent to send back the human user's answer by an `answerQuestion` event. Once `teacher` receives the answer, he will invoke a method named `qualifyStudent` to check the answer and to determine whether the user is qualified to enter the quiz phase. The heuristic rule set for determining the qualification of the user is described as follows:

if the total number of asked questions is less than five
then the qualification of the user is unknown;
if less than 60% of the user's answers are correct
then the user is unqualified
if more than or equal 60% of the user's answers are correct
then the user is qualified

After invoking the `qualifyStudent` method, if the qualification of the user is *unknown* or *nqualified*, `teacher` will send a `moreQuestion` event for asking the `practice` episode to retrieve another question from its question base and go back to the "Awaiting Practice Question" state to await the new question. However, if the user is *qualified*, `teacher` will send a `noMoreQuestion` event to the `practice` episode for ending the practice phase.

The detail of `teacher`'s transition diagram in the quiz episode is shown in Figure 19. In this diagram a `student` could be any one of the `user` agent, `john`, or `mary`, because all the three students have competed for answering questions in the quiz phase.

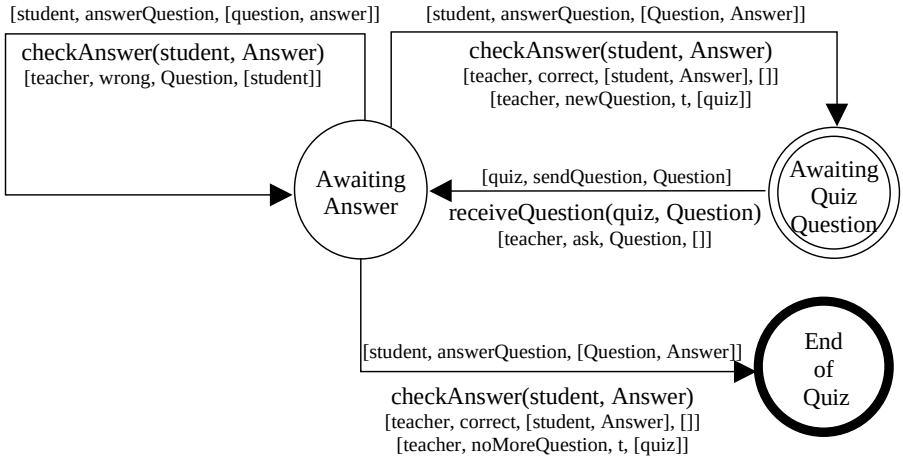

Figure 19. Detail of teacher's transition diagram in quiz episode

In the quiz phase, `teacher` will ask twenty questions to three students. When the `quiz` episode starts, `teacher` is in an "Awaiting Quiz Question" for awaiting the `quiz` episode to send it a `question` by a `sendQuestion` event. Once a `sendQuestion` event is received, `teacher` will invoke a method named `receiveQuestion` to receive the question, and it will broadcast an `ask` event to everyone to ask all the three student agents to answer the question. After sending the `ask` event, `teacher` will stay in an "Awaiting Answer" state to wait for any `student` to send back its/his/her answer by an `answerQuestion` event. Once `teacher` receives the answer, it will invoke a method named `checkAnswer` to check if the answer is correct. If the answer is incorrect, `teacher` will send an event named `wrong` to the `student` to tell it/him/her that its/his/her answer is wrong. In this case, `teacher` will stay in the "Awaiting Answer" state to wait for next answer from any `student`. Once a `student` gives a correct answer, `teacher` will broadcast a `correct` event to inform everybody that the `student` has given a correct answer. In addition, `teacher` will either send a `moreQuestion` event for asking the `quiz` episode to retrieve another question from its question base and go back to the "Awaiting Quiz Question" state if the number of asked question is still less than twenty, or it will send a `noMoreQuestion` to the `quiz` episode for ending the quiz phase if the number of asked question has reached twenty.

Note that since `teacher` needs to check answers from students in both practice and quiz phases, he must act like an expert who always knows the correct answer. To make `teacher` an expert, we simply let it delegate the task of calculating the answer to the CAROL5 interpreter. That is, whenever `teacher` wants to check a student's answer, it will translate the integer addition or subtraction into an CAROL5 expression, and then ask the CAROL5 interpreter to calculate the exact answer.

II. *john*:

The behavior of `john` is simple. It waits for `teacher` to ask a question, then it takes a short time to solve the question, then gives its answer to `teacher`, and then waits for `teacher` to inform it if its answer is correct or wrong. If its answer is wrong, it will try to solve the question again, unless someone has already given the correct answer, and `teacher` starts to ask another question. The state transition diagram of `john` is shown in Figure 20.

Initially, `john` is in an "Awaiting Question" state for awaiting `teacher` to ask a question by the `ask` event. After receiving a question, `john` will invoke a method named `solve` to solve the question and consequently send an answer to `teacher` by an `answerQuestion` event. After giving the answer, `john` will change to an "Awaiting Correction" state for awaiting `teacher` to check the correctness of the answer. In the "Awaiting Correction" state, `john` may receive an event from `teacher` for telling it that its answer is wrong, or it may receive a broadcast from `teacher` for informing it that a student has given the correct answer.

If `john` is told that his answer is wrong, it will continue to make efforts to solve the question. However, if the correct answer has been given, `john` will invoke a method named `updateScore` to determine whether it is the first student who gave the correct answer. If it is, then its score should be increased.

Unlike a simulated teacher, which usually acts like an expert with authority, a simulated student is usually supposed to be a peer of the human user. Thus, to be more realistic, a simulated student such as `john` will take some time for solving a question and may make a mistake sometimes.

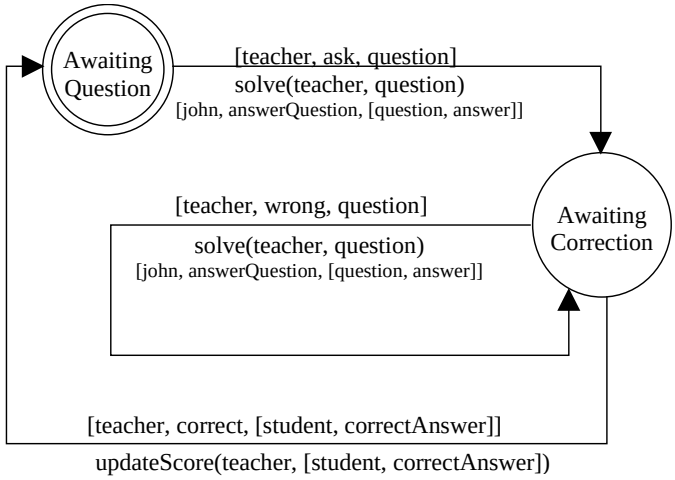

Figure 20. State transition diagram of `john`

III. *mary*:

The behavior of `mary` is exactly the same as that of `john` except that it is simulated as a less capable student. In the implementation, `mary` is a child object of `john`. The only difference between `mary` and `john` is two attribute values which control the response-time and probability of correctness respectively. The human user will feel that `mary` is less capable than `john` because it usually takes a longer time to solve a question and the probability that its answer is correct is lower.

IV. *user*:

The state transition diagram of the `user` agent is shown in Figure 21.

Although its state transition diagram looks similar to the one of `john`, the `user` agent is not implemented as a child object of `john` or `mary` because the event handlers of the `user` agent are totally different from those of `john` and `mary`. Unlike `john` and `mary`, the `user` agent does not solve any question by itself. Instead, it only acts as an interface agent between other agents and the human user. Whenever the `user` agent receives a new question which comes with an `ask` event from `teacher`, it will translate the question into a user-readable form and ask the human user to solve it. Once the human user enters his answer, it will pack the answer into an agent-understandable event and send it back to `teacher`.

In addition to acting as an interface agent, the `user` agent is in charge of recording the history and present state of the human user such as the score he/she got in the practice episode. `Teacher` uses these data to check the performance of the human user. Thus, the `user` agent can also be viewed as a simple student model of the human user.

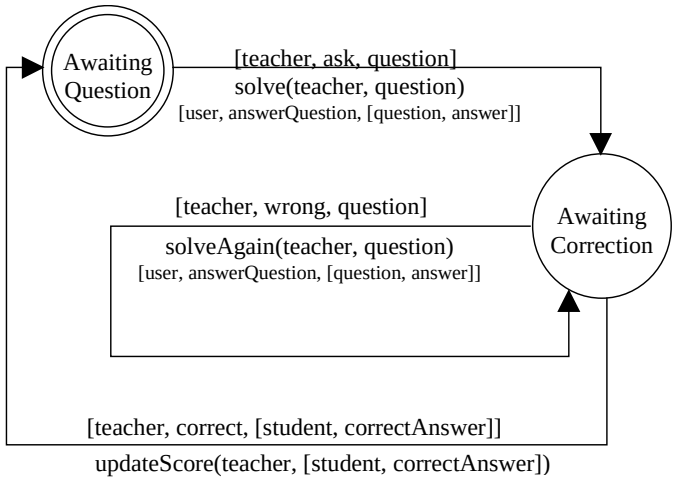

Figure 21. State transition diagram of `user` agent

Putting Them All Together

The overall object hierarchy of Three's Company is shown as Figure 22. As can be seen, the curriculum-tree is a sub-tree of the object hierarchy. The `user` agent, `teacher` and `john` are three children of the root object, while `mary` is a child of `john`.

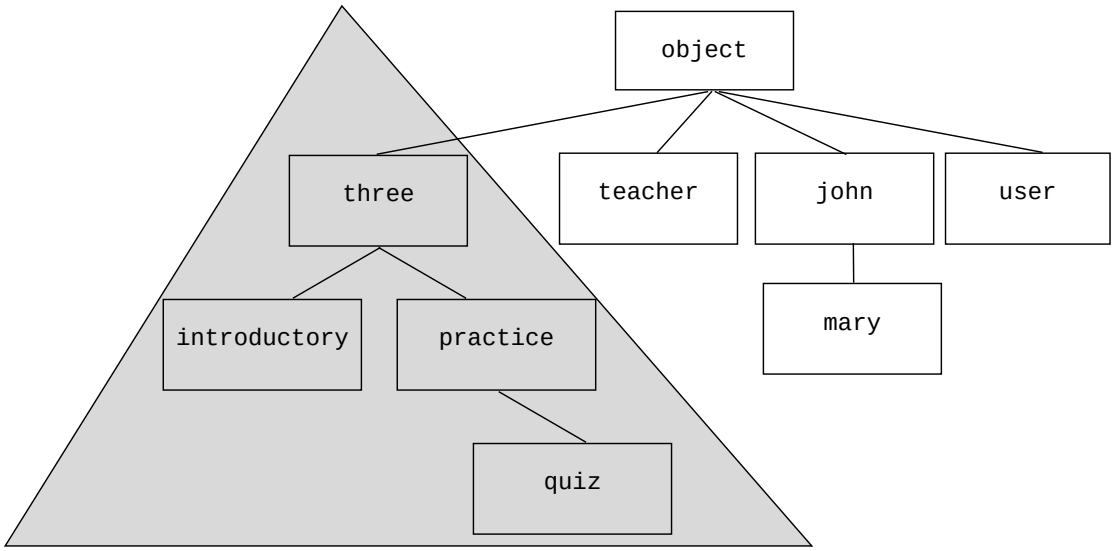

Figure 22. Overall object hierarchy of Three's Company

Although Three's Company is just a simple system, the design process shown above has illustrated a typical framework for developing social learning systems based on a curriculum-tree and educational agents. More importantly, we have demonstrated that CAROL5 supports this framework in a natural way.

4 DISCUSSION

In this section, we will clarify some design decisions related to supporting the development of social learning systems in CAROL5.

4.1 Advantages of Prototype-Based Model

In comparison with the class-based model, the prototype-based model can simplify the relationships among objects. In the class–based model, we must grasp two relationships, the "is a" relationship, which indicates that an object is an instance of some class, and the "kind of" relationship, which indicates that an object's class is a subclass of some other object's class. In the prototype–based model, there is only one relationship, "delegate to", that describes how objects share knowledge.

In the class-based model, one must describe the abstract properties of a class that an object would belong to before creating the object, and then create similar objects by instantiation of that class. The knowledge (state and behavior) related to an object is distributed between the object itself and the class it belongs to: the class holds the behavior of the object, while the object holds its own state. The disadvantage of the class-based model is its difficulty of dynamic knowledge manipulation. For example, suppose we want to change the behavior of an object in the class-based model by changing a method definition in its class. Most class-based programming languages do not allow us to redefine the method definition of a class at run-time. In the prototype-based model, on the other hand, the knowledge related to an individual object is held directly by the object itself. Therefore, every object can be viewed as a self-sufficient entity with its own state and behavior, and its properties can be changed at any time we want to. That is, we can dynamically alter the knowledge of an object by adding or deleting its property.

The main reason that we prefer the prototype-based model to the class-based model is that the former is more flexible for manipulating knowledge. The capability of incremental and dynamic knowledge construction in the prototype-based model is crucial for building and manipulating knowledge bases in learning systems. For example, the knowledge of the student model must evolve to reflect the learning state of students. The simplicity and flexibility of the prototype-based model makes knowledge construction and manipulation easier than in the class-based model, and thus is more suitable in developing systems that incrementally acquire knowledge.

In addition, the prototype-based model allows programmers to directly create concrete objects without defining an abstract class in advance. From our experience of developing social learning systems, the capability of directly creating concrete objects is very convenient for constructing educational agents. The class mechanism in the class-based model is useful for developing some software systems such as computer-aided design systems or database systems, which require a large number of objects of the same type, an ICAL system usually does not require a large number objects of the same type. In the field of social learning systems, although agents may have some common behavior, almost every agent is unique to some specific system. This means that if we adopt the class-based model, programmers would need to create a class for almost every agent. This will be a burden. Thus, we conclude that the prototype-based model is more intuitive for developing social learning systems whose success mainly depends on the intelligence of individual educational agents rather than the capability of manipulating a large amount of objects of the same type. The prototype-based model is preferable because it allows programmers to start by creating concrete objects, and then fine tune the behavior of each individual object.

Also, remember that our original motivation of designing CAROL5 was in developing the Curriculum-Tree. One of the characteristics of Curriculum-Tree is that knowledge stored at a higher-level node of the tree is shared by all children of the node, while knowledge stored at a lower-level node overrides knowledge stored at its parent. The capability of representing default knowledge in the prototype-based model answers that purpose. As shown in the Three's Company example, we can directly represent the curriculum-tree as part of the delegation hierarchy in CAROL5.

4.2 Advantages of Integrating Prototype-Based and Rule-Based Programming

Unlike other rule-based programming languages in which rules are their principal elements and rule-based reasoning is their global problem-solving mechanism, rule-based reasoning in CAROL5 is a powerful but “humble” feature in the sense that rules are packed into methods and procedures. In CAROL5, interactions among objects compose the main rhythm of global

problem-solving, while rule-based reasoning is the underlying mechanism helping each object perform every single step of its actions. One will think CAROL5 is a normal prototype-based programming language unless one takes a close look at the bodies of methods or procedures. The advantage of this kind of integration is that the prototype-based model helps programmers to organize rules and rule-based reasoning adds AI power into the prototype-based model.

The gain of integrating rule-based programming and prototype-based programming is more than the sum of the parts. On one side, rules are powerful for developing AI-intensive software such as ICAL systems. On the other side, by organizing rules into the body of methods and procedures, we not only increase the readability and maintainability of methods and procedures but also solve the indexing problem in traditional rule-based systems such as production systems.

4.3 Distinction between Methods and Procedures

Unlike some prototype-based programming languages, in which procedures are not supported, CAROL5 supports both methods and procedures. We decided to supports both methods and procedures in CAROL5, not only because their semantics are different, but also because they represent different kinds of procedural knowledge. Methods are mainly used to represent private behavior of an object, while procedures usually represent the capability of general problem-solving that is nothing to do with the state of the object itself.

Another reason to support procedures is that, in some cases, procedures are more natural than methods. Methods force programmers to distinguish a 'subject' from some 'objects' among its arguments: the receiver is the 'subject', and the rest are those 'objects'. However sometimes, none of the arguments should be the 'subject'. For example, suppose that we want to calculate the greatest common divisor (GCD) of two integers, say M and N. It seems to be a little hard to choose a 'subject' between M and N, since they play equal roles here. Furthermore, since the capability of calculating GCD is not their behavior by nature, it seems to be a little odd to implement `gcd` as a method of M or N. It seems to be preferable to define it as a procedure. Also, based on the consideration of performance, some system-defined primitives in CAROL5 are implemented as procedures. This also prevents us from removing the notion of procedures.

4.4 Distinction between Fetching and Executing

Unlike some prototype-based programming languages, in which a reference to a method will execute it, CAROL5 has a distinction between fetching and executing a method or a procedure. For consistency, a variable reference always returns the content of a property in an object as a value. In the case that the content is a method or a procedure, a value representing it will be returned. This distinction makes sense since methods and procedures are both first-class entities. In CAROL5, to execute a method or a procedure, one must make an explicit call to it.

Many programming language designers (Ungar and Smith, 1987; Smith, 1994a) insist that for a prototype-based programming language, a reference to a method will execute it, and thus it can provide better information-hiding. For example, when we refer to an object `john`'s attribute, say `age`, we may get the same result, say 25, no matter whether it is directly stored as an attribute value or it is actually a method calculating the age based on the birthdate. In CAROL5, if the attribute `age` is implemented as a method, we must invoke it explicitly. Indeed, there is trade-off between better information-hiding and more flexible knowledge manipulation. The concern of developing AI-intensive learning systems leads us to provide better flexibility.

The capability of fetching methods or procedures enables us to manipulate them. For example, this capability enables us to create higher-order methods or procedures, which have been proved to be a powerful feature by the community of functional programming. In addition, since procedures adopt static binding, higher-order procedures can be used to produce closures (Field and Harrison, 1988; Wang and Chan, 1996), which will remember the run-time environment. Closures can be viewed as suspended computations, which can be resumed anytime for the needs of the problem-solving. Thus, we believe that closures can be used to implement 'promises' (Shoham, 1993) among agents. However, this issue is still under investigation, so it is not further discussed in this paper.

4.5 Advantages of Event-Driven Programming

Our ultimate goal is to develop a multimedia authoring system based on CAROL5 for developing learning system. The support of event-driven programming in CAROL5 will be beneficial to the development of our multimedia authoring system. When designing GETMAS (Chan 1991; Wong, Chan, Cheng, and Peng, 1996; Wong and Chan, 1997), a Goal-Episode Tree based Multimedia Authoring System, we found that the most laborious work in implementing a multimedia authoring system is building its multimedia interface builder. The problem is that if we want to make our system useful we must provide an interface builder with a large multimedia library. Thus, unless the manpower can be largely increased, it seems to be impossible to accomplish such a complex multimedia authoring tool in a short time. This motivates us to take advantage of the multimedia interface builder already available from some commercial multimedia development tools. In this way, to construct a multimedia authoring tool suitable for designing intelligent educational programs, all we need to do is to integrate CAROL5 with a commercial multimedia development tool.

The event-driven programming style makes it easier to incorporate CAROL5 into a commercial multimedia development tool that we can use to create event-driven multimedia objects. The result of this incorporation is a multimedia authoring system in which CAROL5 serves as the script language and the multimedia development tool functions as the interface builder.

5 FUTURE WORK

In this paper, we have discussed how CAROL5 has evolved as an agent-oriented programming language through our experience of developing social learning systems. CAROL5 provides some features needed to construct intelligent educational agents, but refinements on CAROL5 are needed in the future.

AOP features of CAROL5 are natural for developing social learning systems, where there are multiple human students and simulated students involving social interactions. However, the current implementation of CAROL5 does not support networked agents yet. We found that without support of networked agents, CAROL5 can only be used for developing a limited range of small social learning systems working on a standalone machine. To be practical, the support of networked agents must be added to CAROL5 in the future. Educational agents should have the ability to communicate with other educational agents or real students on the network. Two kinds of communication models are possible for us to extend CAROL5 to support networking:

Peer-to-Peer

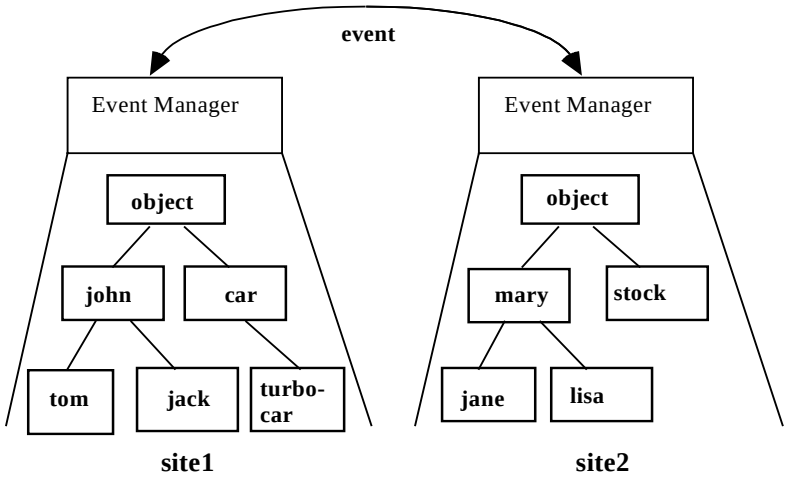

Figure 23. Peer-to-Peer Model

In this model, every standalone machine has its own object hierarchy as shown in Figure 23. Here, the reference from `mary` on `site2` to `john` on `site1` must include `site1` in the head of the reference name, such as `site1.object.john.spouse`.

Client-Server

In this model, there is a server that acts as the root of the object hierarchy. The client machines must register every creation of objects to the server. To avoid naming conflicts, if the name of the object is the same an an object on other machine, the creation of the new object will be rejected. The whole object hierarchy on the network is shown as Figure 24. The advantage of the client-server model is that the name reference is apparent. For example, the reference from mary to john's spouse is `john.spouse`, which is the same in CAROL5.

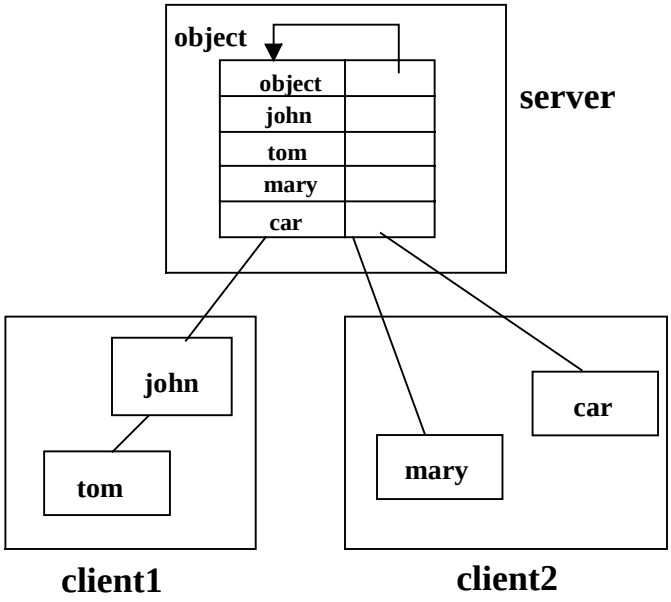

Figure 24. Client-Server Model

In addition to the distributed object hierarchy, the following issues are also worth careful consideration for more complete support to networked educational agents:

- Mobility: Mobile agents can move from one computer to another in the network. Although mobility is neither a necessary nor sufficient condition for agenthood (Nwana 1996), it is attractive in some environments such as social learning systems. For example, a teacher may want to monitor the learning activities of all on-line students. In traditional client-server environments, the teacher needs to setup a 'centralized' server to log the activities of all students' client programs. However, this approach may fail as the required network bandwidth will increase in proportion to the number of students. With the support of mobility, however, the teacher can send an agent to each student's computer and let the agent monitor the learning activities. Once an agent finds a student with a problem, it will inform the teacher immediately. The point is that the teacher does not need to send these agents simultaneously. Instead, he can send them out in sequence. In fact, he can even send them out a week ago (or anytime in advance), and let them sleep in a student's computer until the class starts. The advantage is that the once mobile agents get into a student's computer, they perform 'local communication', and thus do not waste any network bandwidth. Furthermore, there is no need to setup a high-performance centralized server.
- Authentication: Since an educational agent performs tasks on behalf of teachers or students, there must be an authentication mechanism to ensure it is representing who it claims to be representing. Sometimes, the ability to identifying the owner of an agent is

very important in a social learning system. For example, in a learning activity of group discussion, the agents of a student may only be permitted to communicate with agents of students from the same discussion group.

- Security: All systems with networked agents may involve security problems, such as a computer virus, Trojan horse, etc. In a social learning system, one should also be careful about security problems such as unauthorized access to files. In addition, an educational may contain some personal information. A cryptographic mechanism such as public/private key is also necessary for use in agent communication to ensure privacy.

A project of extending CAROL5 to support networked agents is in progress. The experience of designing CAROL5 is helpful for us to extend it for supporting distributed learning systems.

In addition to adding features for supporting networked agents, some improvements on the current implementation of CAROL5 should be made. First, the current implementation of CAROL5 is for experimental usage only, and thus we have not paid much effort to improving the efficiency of the interpreter. However, experience tell us that the efficiency of a programming language is as important as its features. In the future, many optimization techniques, such as byte code compilation, should be applied in implementing the CAROL5 interpreter.

Second, the current implementation of CAROL5 has not yet adopted multi-thread technique supported by most modern operating system. Instead, we use our own scheduling mechanism to coordinate time-sharing among multiple agents. In the future, the implementation of the CAROL5 interpreter should utilize the multi-thread technique, since this powerful technique is very natural and efficient for supporting multiple agents which work concurrently.

Third, a visual programming environment is helpful to programmers. Thus it is worth adding a tree editor to the programming environment of CAROL5 to provide a visualized way for editing the prototype-based object hierarchy. Also, as can be seen, the notion of objects as finite state machines is powerful and state transition diagrams are useful tools for both designing and explaining behavior of objects. It is worth adding a state transition diagram editor to the programming environment of CAROL5, not only for visualizing dynamic behavior of objects but also for reinforcing the notion of objects as finite state machines.

Finally, a mechanism of control event priority should be incorporated into CAROL5, so that programmers can specify higher priority to some emergent events rather than being enforced to accept the simple-minded first-in-first-out event scheduling in the current implementation. For example, most programmers will usually let user-interface events have higher priority, so that systems will be more responsive to users.

References

Booch, G. (1991), *Object-Oriented Design with Applications*, Readings, Benjamin/Cummings Publishing Company, Inc., 1991.

Bradshaw, J. M., Dutfield, S., Benoit, P., and Woolley, J. D. (1997), "KaoS: Toward an industrial-strength generic agent architecture", in *Software Agents*, ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Chan, T. W. and Baskin, A. B. (1988), "Studying with the Prince: The Computer as a Learning Companion", in *Proceedings of International Conference of Intelligent Tutoring Systems*, 1988, June, Montreal, Canada, 194-200.

Chan, T. W. and Baskin, A. B. (1990), "Learning Companion Systems", in C. Frasser and G. Gauthier (Eds.) *Intelligent Tutoring Systems: At the Crossroads of Artificial Intelligence and Education*, Chapter 1, New Jersey: Ablex Publishing Corpration.

Chan, T. W. (1991), "Integration-Kid: A Learning Companion System", in the*Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, Morgan Kaufman Publishers, Inc., 1094-1099.

Chan, T. W. (1992), "Curriculum Tree: A Knowledge-Based Architecture for Intelligent Tutoring Systems", in *2nd International Conference of Intelligent Tutoring Systems*, Lecture Notes in Computer Science, 608, Springer-Verlag, 140-147.

Chan, T. W. and Wang, W. C., (1993), "DARET: A Logic-Based language for Object-Oriented Databases", *In Proceedings of The 5th International Conference on Software Engineering and Knowledge Engineering*, IEEE, San Francisco, 1993.

Chan, T. W. (1995), "Social Learning Systems: An Overview", *Innovating Adult Learning with Innovative Technologies*, B. Collis and G. Davies (eds), IFIP Transactions A-61, North-Holland, 102-122, Also appeared in "Tutorial on Social Learning Systems in *Emerging Technologies in Education*, T. W. Chan & J. Self (eds), AACE, 71-96.

Chan, T. W. (1996), "Learning Companion Systems, Social Learning Systems, and Social Learning Club", Invited Talk, *World Conference on Artificial Intelligence in Education*, Vol. 7, No. 2, 125-159.

Dony, C., Malenfant, J., and Cointe, P. (1992), "Prototype-Based Language: From a New Taxonomy to Constructive Proposals and Their Validation", in *OOPSLA '92 Proceedings*.

Field, A. J. and Harrison, P. E. (1988), *Functional Programming*, Wokingham: Addison-Wesley.

Genesereth, M. R. (1995), "An Agent-Based Framework", *AI Expert*, Mar. 1995, 30-40.

Gilbert, D., Aparicio, M., Artkison, B., Brady, S., Ciccarino, J. Grosof, B., O'Connor, P., Osisek, D., Pritko, S., Spagna, R., and Wilson, L. (1995), "IBM Intelligent Agent Strategy", IBM Corporation.

Gilmore, D. and Self, J. (1988), "The Application of Machine Learning to Intelligent Tutoring Systems", in J. Self, (Ed.) Artificial Intelligence and Human Learning, *Intelligent computer-assisted instruction*, New York: Chapman and Hall, 179-196.

Liebermann, H. (1986), "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", in *OOPSLA '86 Proceedings*.

Lin, G.L. (1993), Three's Company: A social Learning System. Master Thesis, Institute of Computer Science and Electronic Engineering, National Central University, Taiwan.

Maes, P. (1995a), "Artificial Life Meets Entertainment: Lifelike Autonomous Agents, in *CACM*, 38, 11, 108-114.

Maes, P. (1995b), "Intelligent Software", in *Science American*, Vol. 273, No. 3, 84-86, Science American, Inc.

Nwana, H. S. (1996), "Software Agents: An Overview", in *Knowledge Engineering Review*, Vol. 11, No 3, pp. 205-244, October/November 1996.

Rich, C. (1996), "Window sharing with Collaborative Interface Agent", *SIGCHI Bulletin 28*, 1 ,70-78.

Self, J. (1985), "A Perspective on Intelligent Computer-Assisted Learning", *Journal of the Learning Sciences*, 2(3), 235-276.

Shoham, Y., (1993), "Agent-Oriented Programming", in *Artificial Intelligence 1993*.

Smith, R. B. (1994), "Prototype-Based Languages: Object Lessons from class-Free Programming (Panel)". In OOPSLA '94 Proceedings. Panel summary in OOPSLA '94 Addendum to the Proceedings.

Stein, L. A. (1987), "Delegation is Inheritance", in *OOPSLA '87 Proceedings*.

Ungar, D. and Smith, R. (1987), "Self: The Power of Simplicity", in *OOPSLA '87 Proceedings*.

Wegner, P. (1987), "Dimensions of Object-Based Language Design", in *OOPSLA '87 Proceedings*.

Wang, W. C. and Chan, T. W. (1996), "The Support of Higher-Order Methods and Procedures in Prototype-Based Programming Languages", in *Proceedings of 7th Workshop on Object-Oriented Technology and Applications*, Taiwan, 1996.

Wong, W. K., Chan, T. W., Cheng, Y. S., and Peng, S. S. (1996), "A Multimedia Authoring System for Building Intelligent Learning Systems", in *Proceedings of Educational Multimedia and HyerMedia*, ED-MEDIA 1996.

Wong, W.K. and Chan, T.W. (1997), "A multimedia authoring system for crafting topic hierarchy, learning strategies, and intelligent models", *International Journal of Artificial Intelligence in Education*, 8, 71-96.