Active Object System

Sungoon Choi
Toshimi Minoura
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

91-40-1

Active Object System

Sungoon Choi and Toshimi Minoura
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602
choi@cs.orst.edu, minoura@cs.orst.edu (Research)

Abstract

An active object system is a transition-based object-oriented system suitable for the design of various concurrent systems. An AOS consists of a collection of interacting objects, where the behavior of each object is determined by the transition statements provided for the class of that object. A transition statement is a condition-action pair, an equational assignment statement, or an event routine. The transition statements provided for each object can access, besides the state of that object, the states of the other objects known to it through its interface variables. Interface variables are bound to objects when objects are instantiated so that desired connections among objects are established. The major benefit of the AOS approach is that an *active* system can be hierarchically composed from its *active* software components as if it were a hardware system.

Key Words and Phrases: **object-oriented concurrent programming, software IC, production system, hierarchical composition, behavioral composition.**

1 Introduction

An *active object system* (AOS) is an object-oriented concurrent system using transition (production) rules, equational assignment statements, and event routines for its behavior description. Production systems have been known to be suitable for various concurrent systems that require flexible synchronization [DAVI76, ZISM78]. On the other hand, object-oriented programming [GOLD80, MEYE87] enables us to achieve higher levels of modularization and code sharing. These two paradigms are combined in the AOS approach.

The major goal of the AOS approach is to enable us to construct a certain class of concurrent systems by hierarchical composition of active objects, in which software objects are constructed and modularized like hardware objects. Note that a hardware object is an active autonomous object. In an AOS, a higher level object is constructed by putting component objects together with some connections. Systems that can be best handled by the AOS approach are those that are normally represented graphically. Hardware circuits, simulation diagrams, and control system diagrams are examples.

The behavior of each object in an AOS is determined by the *transition statements* provided for the *class* of that object. Each transition statement is a transition rule, which is a *condition-action* pair, an equational assignment statement, or an event routine. Equational assignment statements pioneered by VISICALC can maintain simple invariant relationships among object states. Event routines are activated by messages. We support one-to-many message passing as well as ordinary many-to-one message passing.

One key feature of our AOS is that the transition statements provided for each object can access, besides the state of that object, the states of the other objects known to the object, thus realizing inter-object communication. Desired connections among objects can be established through *interface variables* which can be bound to proper objects when objects are instantiated.

The AOS approach is an object-oriented programming paradigm based on active objects with standardized *structural* interfaces, whereas conventional object-oriented languages support passive objects with standardized procedural interfaces. As active objects provide a higher level of modularity than passive objects [ELLI89], AOS's are easier to design, implement, and maintain than ordinary object-oriented programs.

The idea of active objects originated with the first object-oriented language SIMULA [BIRT73], where active objects are cooperating sequential processes that communicate with each other through procedure calls. Several active object systems have been designed since then by replacing procedure calls used by SIMULA with message passing. Actors introduced active computational agents that carry out their actions in response to incoming messages [AGHA86]. ABCL objects [YONE87] and Emerald objects [BLAC86] may be sequential processes that exchange messages among them.

Researchers recently started to emphasize behavioral composition of objects. Inheritance and subtyping in the parallel object-oriented language POOL-I are addressed in [AMER90], including composition of heterogeneous objects. *Contracts* [HELM90] provides a basis for interaction-oriented design that facilitates identification, abstraction and reuse of patterns of behavior in programs. *Collaborations graphs* for analyzing paths of communication and identifying potential subsystems are introduced in [WIRF90].

Some AI systems allow us to create active objects by using active values [KUNZ84, KEHL84]. A KNOs object has an internal state, a set of operations, and a set of rules [TSIC87]. Operations on KNOs objects are invoked by messages and state-driven transition rules.

In Section 2, we give an overview of an AOS by using a simple queuing system. AOS features are further explained in Section 3, and implementation issues are discussed in Section 4. Section 5 concludes this paper.

2 Simple Queuing System

In this section, we introduce an AOS program by using a simple example of a queuing system.

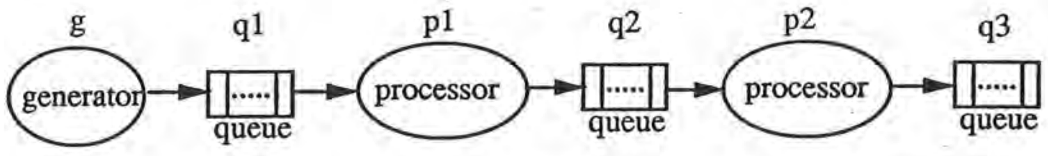

Figure 1: Queuing system with 2 processors and 3 queues.

The queuing system in Fig. 1 consists of a generator g that generates a stream of jobs, two processors p1 and p2 that process jobs, and three queues q1, q2, and q3 that hold jobs. Fig. 2 gives an AOS description of the system. It can be easily seen that the code corresponds exactly to the diagram in Fig. 1; the output of g is connected to q1, the input of p1 is connected to q1 and its output to q2, and so on.

```
QueuingSystem {
   Private
      Generator g  with {output = q1;};
      Queue q1;
      Processor p1 with {input = q1;  output = q2;};
      Queue q2;
      Processor p2 with {input = q2;  output = q3;};
      Queue q3;
   Public
      boolean systemRunning = true;
}
```

Figure 2: Queuing system main.

An AOS program consists of a main program and classes. A class contains three parts: an interface, a set of instance variables, and a behavior description, Instance variables and behavior can be *private*, *protected*, or *public*, as in C++ [STRO86]. The main program is defined in the same way as a class but is an instance.

In order to construct the above system according to the AOS approach, we first define the four classes used by the system: class `Generator`, class `Queue`, class `Processor`, and class `Job`. Second, we create one instance g of Generator, three instances q1, q2 and q3 of Queue and, two instances p1 and p2 of Processor and provide the static interconnections among them as specified in Fig. 2. The interconnections between Job instances and the other system components cannot be defined statically, since Job instances are created dynamically by g and are moved to the processors and the queues during the execution of the system.

We now show the definitions of the classes used by the system.

```
Class Timer {
   Instance Variables
      enum {reset,running,complete} status;
   Behavior Description
      void startTimer(int delay) {
         /* Sets status to running.
            When the delay time passes,
            status automatically changes to complete. */
   }
}
```

Figure 3: Timer.

```
Class Job{
   Instance Variables
      Int ID;
      Job *next;
}
```

Figure 4: Job.

Class Timer. Class Timer is a system-defined class. A Timer is used by a Generator or Processor to measure a time interval. Initially its status is reset. After it starts running, its status changes to complete when the specified delay time expires.

Class Job. Class Job has instance variables ID and next. The next field is used to point to the next Job when Jobs form a queue. It has no behavior description.

Class Generator. Class Generator generates a stream of jobs whose inter-arrival times are randomly distributed. Interface variable output of class Queue designates the queue to which this generator feeds jobs. Transition rule Start initiates the timer tm, whose expiration time designates the next job's generation time. When the timer expires, transition rule Stop that generates a job and resets the timer is activated.

```
Class Generator {
   Interface
      Queue output; /* imported reference */
   Instance Variables
      Timer tm with status = reset;
      Job newJob;
      int jobID = 0;
   Behavior Description
      /* Transition Rule Start */
      if (system_running and (tm.status == reset))
         tm.startTimer(random());
      /* Transition rule Stop */
      if (tm.status == complete) {
         newJob = new Job;
         newJob->ID = jobID++;
         newJob->next = nil;
         output->enqueue(newJob);
         tm.status = reset;
      }
}
```

Figure 5: Generator.

```
Class Queue {
   Interface
      /* none */
   Instance Variables
      int njobs;
      Job *head = nil, *tail = nil, *temp;
   Behavior Description
      void enqueue (Job* job) {
         /* put the job at the end of the queue */
         if (tail == nil) {
            tail = job; head = job;
         }
         else {
            tail-> next = job; tail = tail->next;
         }
         njobs++;
      }

      Job* dequeue () {
         /* get the job in front of the queue */
         if (head) {
            njobs--;
            temp = head; head = head->next; temp->next = nil;
         }
         return (temp);
      }
}
```

Figure 6: Queue.

```
Class Processor {
   Interface
      Queue input, output; /* imported reference */
   Instance Variables
      boolean avail = true;
      Timer tm with status = reset;
      Job *job = nil;
   Behavior Description
      /* Transition Start */
      if ((input.njobs > 0) and (avail == true)) {
         job = input->dequeue();
         avail = false;
         tm.startTime(random());
      }
      /* Transition Stop */
      if (tm.status == complete) {
         tm.status = reset;
         avail = true;
         output->enqueue(job);
      }
}
```

Figure 7: Processor.

Class Queue. Class `Queue` has two methods, **enqueue** and **dequeue**. Instance variable `njobs` is directly accessed by `Processors`.

Class Processor. A `Processor` processes jobs found in the `Queue` designated by interface variable `input` one at a time. When the processing of each job is complete, the job is placed in the `Queue` designated by interface variable `output`. The Timer `tm` is used to measure the processing time, which is randomly generated. Instance variable `avail` indicates if the processor is free or busy. Transition rule `Start` is activated when the processor is free and when there is at least one job in the input queue. Once activated, the processor removes one job from the input queue and starts the timer. When the timer expires, transition rule `Stop`, which resets the timer and moves the job to the output queue, is activated.

3 AOS Features

In this section, we discuss further details of an AOS system. Our discussions centers around its key feature, i.e., *structural composition* of *active* objects. Inter-object communications among the structurally composed active objects are realized by *interface variables* and *transitions*. Besides *event routines* that respond to messages, *transition rules*, which are *condition-action* pairs, and `always` *statements*, which are *equational assignment statements*, can be used as *transition statements* that describe the behaviors of active objects.

3.1 Structural Composition

We can compose a new AOS object from its component objects by providing proper connections among them. *Interface variables*, which act like terminals of hardware components, are used for this purpose. Interface variables are basically pointers through which remote objects are accessed. The object bound to an interface variable may be dynamically changed. Dynamic bindings of objects to interface variables are needed to describe time-dependent structural relationships among active objects. As we discuss in Section 4, operations on dynamically bound objects cannot be handled as efficiently as those on statically bound objects. The latter can be compiled into efficient code.

There are two kinds of hierarchies in an AOS. One is the *class hierarchy*, and the other is the *component hierarchy*. Although the component hierarchy exists in ordinary object-oriented programming, we attach more importance to it. Fig. 8 shows the component hierarchy in the queuing system discussed in Section 2.

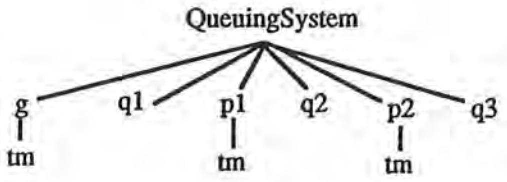

Figure 8: Component hierarchy of the queuing system given in Section 3.

As in ordinary OOP, public and protected instance variables and transition statements are inherited from a superclass to its subclasses according to the class hierarchy. Furthermore, according to the component hierarchy, public instance variables are visible to its components. For example, the instance variable *systemRunning* of the main program *QueuingSystem* is visible to the instance g of class Generator. The inheritance according to the class hierarchy precedes the inheritance according to the component hierarchy.

3.2 Behavior Description

Objects in Smalltalk or C++ are passive in the sense that they only respond to the messages sent to them. On the other hand, the behaviors of AOS objects can be specified by three kinds of transition statements: *transition rules*, `always` *statements*, and *event routines.*

3.2.1 Transition Rule

Each transition rule is a condition-action pair, and its action part is executed when its condition part is satisfied. An execution of a transition rule should be atomic. The implementation details of the execution mechanism of transition rules are discussed in Section 4.

3.2.2 Always Statements

A simple mechanism for describing a behavior of an object is an equational assignment statement that maintains an invariant relationship among the states of objects. `always` statements are used for this purpose.

For example, the class `AND-Gate` can be defined as follows. Note that interface variables `input1` and `input2` are bound to instances of `Gate`, each of which has an instance variable `output`.

```
class Gate {
  Instance Variables
    bool output;
}

class AND-Gate: public Gate {
  Interface Variables
    Gate input1, input2;
  Behavior Description
    always state = input1->output and input2->output;
}
```

An `always` statement can be implemented similarly to a transition rule. The execution of the assignment statement should be triggered whenever any of the variables used in the expression of the `always` statement changes.

3.2.3 Event Routines

The activations of transition rules are, at least conceptually, state-driven. Active objects can communicate with each other by directly accessing the states of other objects rather than sending messages to them. Although this mechanism often eliminates the necessity of explicit message passing, some events are more efficiently handled by messages. An AOS supports event-driven activations of procedures. We consider *messages* as extended events that include some data as parameters. The unique feature of our message passing mechanism is that it supports *one-to-many* message-passing as well as *many-to-one* message-passing.

We provide two constructs that support message passing among objects. The statement `say` $message(p_1, ..., p_n)$ `[to` *receiver*`]` is used to send a message (to the receiver specified as *receiver*). The receiver may or may not be specified. The statement `on` $message(p_1, ..., p_n)$ `[from` *sender*`]` is used to receive a message (from the sender specified as *sender*). The sender may or may not be specified. For a pair of `say` and `on` statements between which a message is passed, if the sender is not specified in the `say` statement, the receiver must be specified in the `on` statement, and vice versa. The receiver object of a `say` statement and the sender object of an `on` statement, if specified, should be visible to the objects issuing these statements.

The message-passing mechanism in ordinary OOP can be regarded as *many-to-one* message-passing, where each `say` statement specifies exactly one receiver of the message, while one `on` statement can receive messages from different senders.

If the receiver is not specified in a `say` statement, the message can be received by any objects to which the sender is visible. Hence, we can implement *one-to-many* message passing. In the example given later, we show how this mode of message passing can simplify programming.

3.3 Examples

To illustrate the usefulness of the features discussed in this section, we show the controller of a radio-button group as shown in Fig. 9 as various AOS programs. Our button group has three push buttons of which at most one button can be *on* at any time.

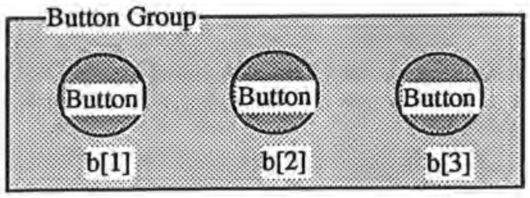

Figure 9: Button Group.

In Fig. 10, two basic classes `GraphicalObject` and `Button` are shown. `GraphicalObject` is the super-class of `Button`. A `GraphicalObject` catches every `mouseDown` event, and if it detects that the event has happened inside its `region`, it broadcasts the message `pressed()`. Class `Button` inherits all the properties of class `GraphicalObject` and has one instance variable `status` which indicates whether the button is on or `off`.

```
class GraphicalObject {
   Region region;

   on mouseDown(Position position) from mouse
      if (in(position, region))
         say pressed();
}

class Button: public GraphicalObject {
   Status status = off;
}
```

Figure 10: Classes GraphicalObject and Button.

In the first implementation called `ButtonGroupA` shown in Fig. 11, class `ButtonA`, which is defined as a subclass of class `Button`, is responsible for the whole operations of the button group. When each instance of `ButtonA` detects message `pressed()` from itself (super-object), it changes its `status` to on and sets the parent's `buttonID` to the value of its `myID`. Each of the other buttons makes its `status` field `off`, detecting its `myID` to be different from the value in the parent's `buttonID`.

```
class ButtonA: public Button {
     int myID;
     on pressed() from this do {
        status = on;
        parent->buttonID = myID;
     }

     if (parent->buttonID <> myID)
       status = off;
}
class ButtonGroupA {
     int buttonID;
     ButtonA b[MAXBUTTON];
     for (i=0; i<MAXBUTTON; i++)
         b[i].myID = i;
}
```

Figure 11: Button group A.

In the implementation shown in Fig. 12, only event routines are used for behavior descriptions, while a transition rule is used by button group A. If a `ButtonB` receives `pressed()` from itself, it changes its `status` to on and sends message `pressed(myID)` to its parent. When the parent object, which is a ButtonGroupB, receives pressed() from its component, it sets the status of the previously pressed button to off and sets the ID of the newly pressed button newButtonID to its instance variable oldButtonID.

```
class ButtonB : public Button {
     int myID;
     on pressed() from this do {
        status = on;
        say pressed(myID) to parent;
     }
}

class ButtonGroupB {
     int oldButtonID, newButtonID;
     ButtonA b[MAXBUTTON];
     for (i=0; i<MAXBUTTON; i++)
         b[i].myID = i;

     on pressed (newButtonID) do {
        b[oldButtonID]->status = off;
        oldButtonID = newButtonID;
     }
}
```

Figure 12: Button group B.

In the last implementation shown in Fig. 13, ButtonGroupC takes care of all of the operations of the button group. The pressed() message generated by any Button is caught by its parent ButtonGroupC instance. We could program ButtonGroupC without introducing a specialized class of Button because on statements can designate their sources of messages. We consider it important not to proliferate class definitions.

4 Implementation Issues

The AOS translator translates AOS descriptions into C++. In this section, we describe the implementation issues of the AOS runtime environment and the translator. The major problem is the mechanism for activating behavior description routines such as transition rules, *always* statements, and event procedures. As these routines are activated by triggers, setting up the *triggers* for them is the main subject of this section.

```
class ButtonGroupC {
     Button b[MAXBUTTON];

     on pressed () from b[1] {
        b[1]->status = on;
        b[2]->status = off;
        b[3]->status = off;
    }
     on pressed () from b[2] {
        b[1]->status = off;
        b[2]->status = on;
        b[3]->status = off;
    }
     on pressed () from b[3] {
        b[1]->status = off;
        b[2]->status = off;
        b[3]->status = on;
    }
}
```

Figure 13: Button group C.

4.1 Event Scheduling

An AOS computation is a sequence of executions of transitions, which transform the states of objects and generate events. These new states of objects and events may trigger executions of other transitions. We call an object that contains a transition a *source object*, since a transition contains the sources of references to other object states. The objects referenced by the condition parts of transitions are called *trigger objects*, since their state changes must trigger the executions of the transitions in source objects. An AOS computation can be best understood in terms of trigger objects and triggered transitions. Fig. 14 shows this interaction between the trigger objects and the triggered transitions in the simple queuing system discussed in Section 2.

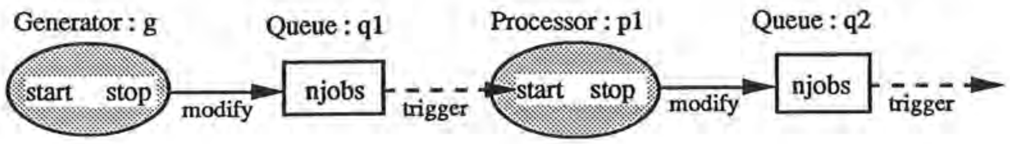

Figure 14: Interaction between trigger objects and triggered transitions.

In an AOS, triggered transitions are executed one at a time. Since an execution of a transition may cause activations of multiple transitions, multiple triggered transitions may be pending at any given time. The transitions triggered simultaneously should not interfere with each other, or their interference should not cause any undesirable effects. When simultaneously activated transitions cause undesirable effects, the programmer must take care of the problem by introducing, for example, interlocking variables or programmer-defined queues.

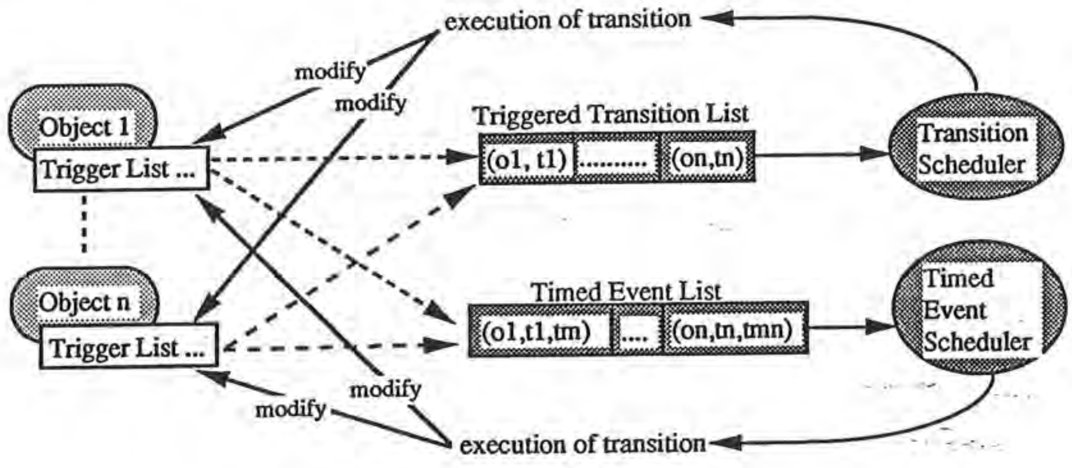

Figure 15: AOS runtime environment.

Fig. 15 shows the run time environment for an AOS. The identities of the transitions to be activated when the state of an object changes are stored in the *trigger list* TL associated with it. Each entry of TL associated with a trigger object o is a pair (s, t), which indicates that transition t of object s should be activated when the state of o changes. We call such a pair a *trigger element*. Activations of transitions are handled as follows. When the state of a trigger object is changed, the trigger elements in its TL are added to the *triggered-transition-list* TTL. TTL, which is global, maintains the trigger elements for all the triggered transitions. Transitions designated by the trigger elements in TTL are executed one by one in first-come-first-serve basis by the *transition scheduler* TS.

Although it is possible to let individual objects produce events at future times, computation resources can be saved if *timed events* (or future events) are directly supported by the system. A timed event is a triple (s, t, tm) which indicates that transition t of object s should be executed at time tm. When a timed event is posted, it is added to the *timed event list* TEL, and when the time specified in it is reached, its execution is scheduled by the *timed-events scheduler* TES.

4.2 Trigger Setup

In order for the transitions in an AOS to work properly, whenever the state of an object changes, all the transitions that refer to that state in their condition parts should be activated. Hence, triggers should be setup for the objects whose states are referenced by transitions.

We now introduce some definitions used in our discussions.

Reference Path. A *reference expression* such as `x.p->q->y.z`, where `x`, `y`, `z`, `p`, and `q` are field names, and furthermore `p` and `q` are pointers, is used to access the state of an object. A reference expression, when evaluated, produces a *reference path* $(v_0, v_1, v_2, ..., v_n)$, where v_0 is a source object, v_i is a composite object containing v_{i+1} or a pointer object pointing to v_{i+1}, and v_n is a trigger object. The reference path for the reference expression `x.p->q->y.z` is (`x`, `x.p`, `*(x.p)`, `x.p->q`, `*(x.p->q)`, `x.p->q->y`, `x.p->q->y.z`). We designate each reference by (*source-object-ID*, *reference-ID*). Reference IDs are integers unique relative to each object.

Dynamic Path Pointer. A pointer object that is an element of a reference path is called a *path pointer*. When the value of a path pointer is changed during the execution of an AOS program, it is called a *dynamic path pointer*.

4.2.1 Static Triggers

In most cases, the reference path for a reference does not involve any dynamic pointers. In this case, the trigger for that reference need not be changed during the execution of the program. Since we can know the exact object that will be accessed when the source object is instantiated, the trigger for the reference need be setup only once when the source object is instantiated.

4.2.2 Dynamic Triggers

When the reference path of a reference involves a dynamic pointer, the trigger object for that reference changes when the value of that pointer is manipulated. If this happens, the trigger in the old trigger object should be dropped, and the new trigger should be added to the new trigger object. It is generally impossible to tell which trigger object is accessed until the operation that modifies the dynamic trigger object is actually executed.

When the value of a dynamic path pointer is changed, all the reference paths that involve the dynamic path pointer are affected. We associate with each dynamic path pointer a *path list* that contains the set of *path elements* each of which indicates a reference path that includes the dynamic path pointer. A path element is a pair (*source-object-ID*, *reference-ID*).

Fig. 16 shows the trigger lists and the path lists maintained by trigger objects and dynamic path objects, respectively. The situation depicted is as follows. Transition t_1 of object a accesses the state of

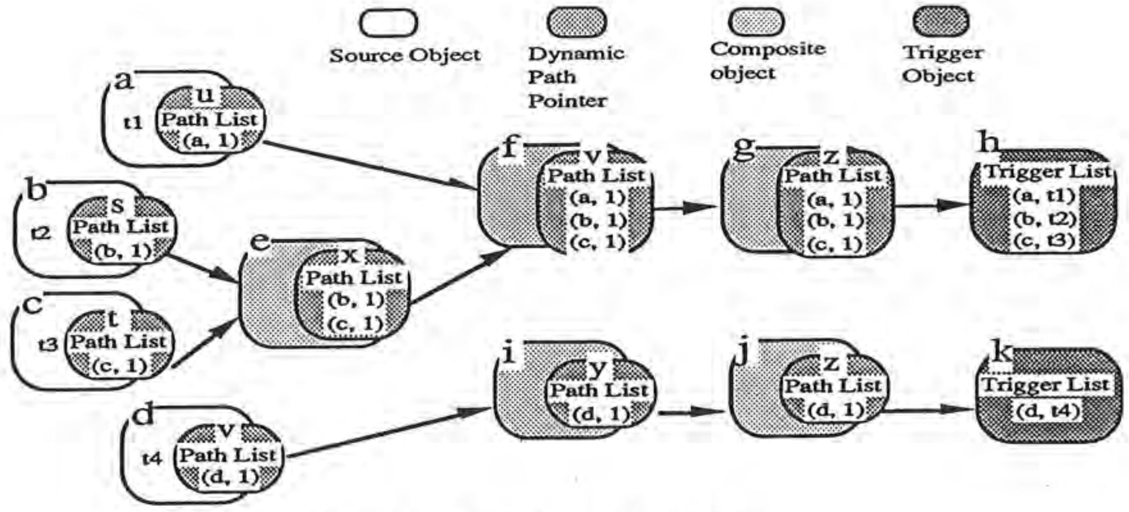

Figure 16: Trigger lists and path lists.

object h through reference expression `*(u->v->z)`. Transition t_2 of object b and transition t_3 of object c also access the state of h, and transition t_4 of object d accesses that of object k. The reference path IDs of all of these reference paths happen to be 1. Note that reference path IDs are relative to each source object. The path list of dynamic path pointer y in object f, for example, is ((a, 1), (b, 1) and (c, 1)), which indicates that y of f is involved in the three reference paths originating from a, b, and c.

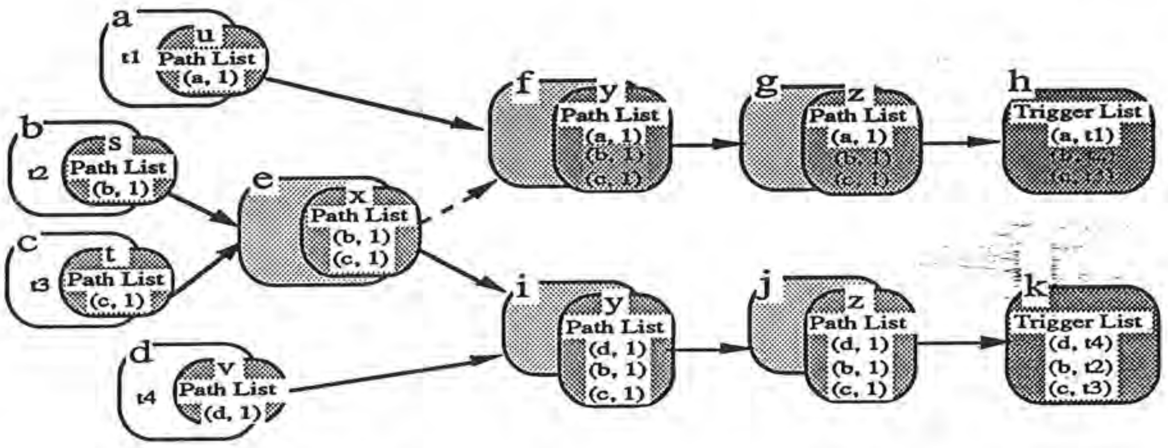

Figure 17: A change of a dynamic path pointer value.

We now show how the trigger lists and the path lists should be updated when the object pointed to by a dynamic path pointer is changed. In Fig. 17, the dynamic path pointer x in e, which initially points to f as in Fig. 16, is changed to point to object i. Now the trigger elements (b, t_1) and (c, t_3) are removed from h, and they are added to k. Path elements (b, 1) and (c, 1) are deleted from y of f and z of g, and they are added to y of i and z of j.

```
for each path element (s, i) in the path list of p do
  begin
   remove path element (s, i) from every dynamic
   path pointer in the old reference path (s, i),
   and remove the trigger element (s, tj) from the old trigger object;
   change p;
   add path element (s, i) to every dynamic
   path pointer in the new reference path (s, i),
   and add the trigger element (s, tj) to the new trigger object;
end;
```

Figure 18: Changing a dynamic path pointer value.

When the value of a dynamic path pointer *p* is to be changed, the procedure given in Fig. 18 must be executed. In order to allow dynamic trigger setup, methods `addElements(i, t)` and `removeElements(i)`, where `i` is a reference-path ID, and `t` a transition ID, should be provided in the class definition of each source object *s*. Method `addElements(i, t)` adds path element (*s*, *i*) to each dynamic path pointer in the *i*th reference path originating from *s* by calling method `addPathElement(this, i)` for that dynamic path pointer, and it adds trigger element (*s*, *t*) to the trigger object of the *i*th reference path originating from *s* by calling method `addTriggerElement(s, t)` for the trigger object. Method `removeElements(i)` applied to a source object *s* removes the path element (*s*, *i*) from each dynamic path pointer in the *i*th reference path originating from *s*, and the trigger element (*s*, *t*) from the trigger object of that reference path.

Fig. 19 shows the `addElements(i, t)` and `removeElements(i)` methods defined in the class for object *b*.

5 Conclusions

The AOS approach provides a new framework for constructing object-oriented concurrent systems. AOS objects are self-contained and active, and their behaviors are defined by the transition rules, `always` statements, and event routines provided in the classes from which they are instantiated. They interact with other objects connected through their interface variables. These features allow an AOS class to be constructed from its component classes by a pick-and-place method. Thus AOS approach facilitates the reusability of software components.

The simple queuing system discussed in Section 2 was implemented as an AOS system. We are now implementing more complex AOS systems and designing a graphical user interface subsystem. The

```
addElements(i, t) {
  switch (i) {
    1: { s.addPathElement(this, i);
         s->x.addPathElement(this, i);
         s->x->y.addPathElement(this, i);
         s->x->y->z.addPathElement(this, i);
         *(s->x->y->z).addTriggerElement(this, t2); }
  }
}
removeElements(i) {
  switch (i) {
    1: { s.removePathElement(this, i);
         s->x.removePathElement(this, i);
         s->x->y.removePathElement(this, i);
         s->x->y->z.removePathElement(this, i);
         *(s->x->y->z).removeTriggerElement(this, t2); }
  }
}
```

Figure 19: The methods addElements and removeElements.

graphical user interface subsystem displays the state of an AOS and allows its user to interact with it. One application area of AOS's is computer-integrated manufacturing.

References

[AGHA86] Agha, G. A. *Actors: A model of concurrent computation in distributed Systems.* The MIT Press, 1986.

[AMER90] America, P., and Linden, F. A Parallel Object-Oriented Language with Inheritence and Subtyping. In Proc. ECOOP/OOPSLA'90 Conf. on Object-Oriented programming, 1986, pp. 161-168.

[BIRT73] Birtwistle, G., Dahl, O. J., Byhrhang, B., and Nygard, K. *SIMULA BEGIN*, Auerbach, 1973.

[BLAC86] Black, A., Hutchinson, N., Jul, E., and Levy, H. Object structure in the Emerald System. In Proc. OOPSLA'86 Conf. on Object-Oriented programming, 1986, pp. 78-86.

[DAVI76] Davis, R., and King, J. An overview of production systems. *Machine Intelligence, 8*, 1976, 300-332.

[ELLI89] Ellis, C. A., and Gibbs, S. J. Active objects: realities and possibilities. In *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. H. Lochovsky (Eds), ACM Press, 1989, pp. 561-572.

[GOLD80] Goldberg, A., Robson, D. *Smalltalk-80 The language and its implementation* Addsison-Wesley, 1983.

[HELM90] Helm, R., Holland, I. M., and Gangopadhyay, D. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In Proc. ECOOP/OOPSLA'90 Conf. on Object-Oriented programming, 1986, pp. 169-180.

[KEHL84] Kehler, T. P., and Clemenson, G. D. An Application Development System for Expert-Systems. *Systems and Software 34*, 1984, 212-224.

[KUNZ84] Kunz, J. C., Kehler, T. P., and Williams, M. D. Applications development using a hybrid AI development system. *The AI Magazine, 5*, 3, 1984, 41-54.

[MEYE88] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 1988.

[STRO86] Stroustrup, B. *The C++ Programming Language*, Addison-Wesly, 1986.

[TSIC87] Tsichritzis, D., Flume, E., Gibbs, S., and Nierstrasz, O. KNOs: Knowledge acquisition, disemination, and manipulation objects. *ACM Trans. on Office Information Systems, 5*, 1, 1987, 96-112.

[WIRF90] Wirfs-Brock, R. J., and Johnson, R. E., Surveying Current Research in Object-Oriented Design. *CACM, 33*, 9, 1990, 105-124.

[YONE87] Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y. Modelling and programming in an object oriented concurrent language ABCL/I. In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (Eds), The MIT press, 1987, pp. 55-90.

[ZISM78] Zisman, M. D. Use of production systems for modelling asynchronous, concurrent processes. In *Pattern-Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (Eds), Academic Press, 1978, pp. 53-69.