OREGON STATE
UNIVERSITY
COMPUTER
SCIENCE
DEPARTMENT

Actionbase System for Manufacturing Control

Toshimi Minoura
Sungwoon Choi
Randall Robinson
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

91-40-2

Actionbase System for Manufacturing Control

Toshimi Minoura, Sungwoon Choi, and Randall Robinson
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602
minoura@cs.orst.edu, choi@cs.orst.edu, robinra@cs.orst.edu

Abstract

A modern computerized manufacturing control system must manage production data, coordinate control actions, and provide user-friendly interfaces. An *actionbase management system* (ABMS) is a *general* software system that facilitates implementations of *actionbase systems* that provide these capabilities for different applications. Besides an ordinary data management facility, our ABMS includes action control and user interface subsystems implemented as *active object systems* (AOSs).

An AOS is a *transition-based* object-oriented system suitable for the design of various concurrent systems. The behavior of each active object is defined by the *transition rules*, the *equational assignment statements*, and the *event routines* provided in its class definition. An active object can be constructed from its component active objects through structural composition as if it were a hardware object.

The user interface management subsystem of the ABMS allows us to provide *declarative* descriptions of *views* for active objects. These views provide user interfaces for an actionbase system.

Key Words and Phrases: manufacturing control, flexible manufacturing system, actionbase system, active-object system, software IC, object-oriented concurrent system, graphical user interface, active-object user interface.

1 Introduction

Numerically-controlled machines, robots, and automatically-guided vehicles (AGVs) are extensively used in modern manufacturing. A software system that coordinates the operations of an automated factory must provide capabilities for data management, control, and man-machine interface. Although the architecture of such a system has been extensively investigated, satisfactory solutions are yet to be found [NAYL-87].

We propose a software system that we call an *actionbase management system* (ABMS). The goal of an ABMS is to achieve for manufacturing-control software development what 4GL systems [INFO88, ORAC87, ORAC88, UNIF87][1] have achieved for business-application software development. We call an application based on an ABMS an *actionbase system.*

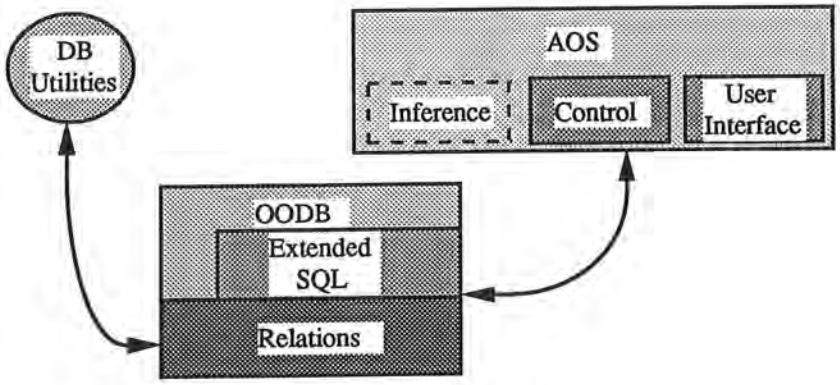

Figure 1: Actionbase system.

Fig. 1 shows the architecture of an actionbase system. The data management function of an actionbase system is handled by an ordinary database management system (DBMS). The records representing such entities as machines and jobs are stored in the database, and they are used in answering queries and producing reports. The central feature of an actionbase system is an *active object system* (AOS) [CHOI91a] that takes control actions on external entities such as robots and machines, responding to their state changes. The AOS can directly manipulate the objects stored in the database, and their states are accessible for such database functions as answering queries and report generation. When AOS objects stored in the database are manipulated by ordinary queries, these changes must be notified to the AOS as *events.*

[1] A 4th-generation language (4GL) system is an application-specific language at a higher level than procedural languages such as Pascal or C, and it ties together its two major components, a database subsystem and a user interface subsystem. A 4GL often achieves several-fold productivity gains.

The idea of active objects originated from 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 [AGHA86, BLAC86, YONE87] have been designed since then by replacing procedure calls of SIMULA with message passing. Some AI systems allow us to create active objects by using active values [KUNZ84, KEHL84]. A KNOs object [TSIC87], which possesses an internal state, a set of operations, and a set of rules, is also active. One approach for implementing a user interface system from active objects is introduced in [COOT88].

Our *active object system* (AOS) uses *transition rules*, *equational assignment statements*, and *event routines* for its behavior description. Transition rules are *condition-action* pairs, and they have been known to be suitable for various concurrent systems that require flexible synchronization [DAVI76, ZISM78]. Equational assignment statements can maintain simple invariant relationships among object states. Event routines are activated by messages. Our AOS supports one-to-many message passing as well as ordinary many-to-one message passing.

We call transition rules, equational assignment statements, and event routines *transition statements*. The behavior of each active object is determined by the transition statements provided in the *class* definition of that object. One key feature of our AOS is that the transition statements provided for each active object can access the states of the other active objects known to it, realizing inter-object communication.

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

The major goal of the AOS approach is to construct a certain class of systems by the hierarchical composition of active objects, where software objects are constructed and modularized like hardware objects. This feature is very useful in constructing manufacturing control software system because we can represent such real-world entities as machines, robots, and AGVs by active objects.

Besides coordination of control actions, the AOS provides a user interface for a manufacturing control system. Fig. 2 shows a simple *view* of a manufacturing line. Through this view an operator can see the movements of jobs and query the statuses of jobs and machines. Such views are supported by an *active-object user interface system* (AOUIS), which is an AOS specialized for user-interface description [CHOI91b].

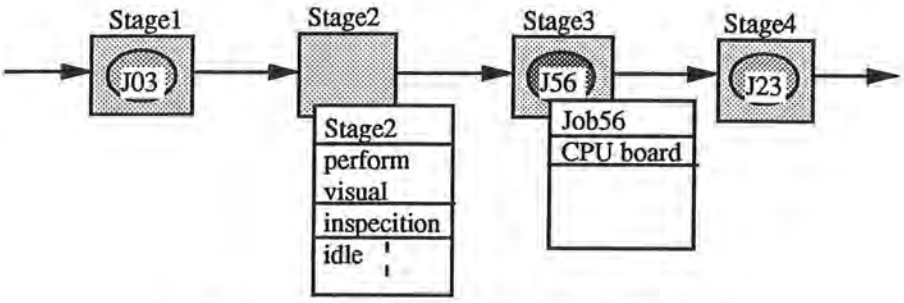

Figure 2: A view of a manufacturing line.

In Section 2, we give a brief overview of an AOS. A flow-line manufacturing system is modeled by an AOS in Section 3, and a flexible manufacturing system 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.

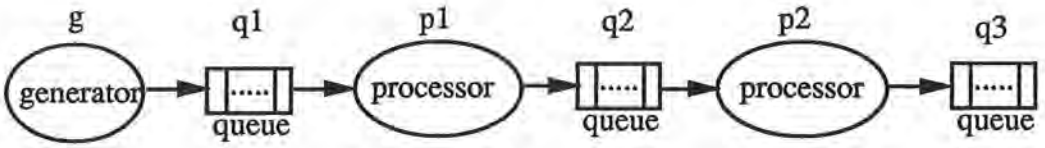

Figure 3: Queuing system with 2 processors and 3 queues.

The queuing system in Fig. 3 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. This queuing system is similar to a flow-line manufacturing system.

Fig. 4 gives an AOS description of the system. It can be easily seen that the code corresponds exactly to the diagram in Fig. 3; the output of `g` is connected to `q1`, the input of `p1` is connected to `q1` and its output to `q2`, and so on.

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.

```
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 4: Main program of a queuing system.

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. 4. 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 5: Timer.

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 Job{
   Instance Variables
      Int ID;
      Job *next;
   /* other information */
}
```

Figure 6: Job.

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

Figure 7: 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 8: 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 */
      when ((input.njobs > 0) and (avail == true)) {
         job = input->dequeue();
         avail = false;
         tm.startTime(random());
      }
      /* Transition Stop */
      when (tm.status == complete) {
         tm.status = reset;
         avail = true;
         output->enqueue(job);
      }
}
```

Figure 9: Processor.

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 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.

We now summarize the behavior description mechanism of an AOS. The details can be found in

[CHOI91a].

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* (`when` statements), *equational assignment statements* (`always` statements), and *event routines* (`on` statements).

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.

An equational assignment statement maintains an invariant relationship among the states of objects. For example, the `output` of an `AND-Gate` can be defined from its `input1` and `input2` as

```
always output = input1 and input2;
```

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. An AOS allows *one-to-many* message-passing as well as *many-to-one* message-passing.

3 Flow-Line Manufacturing

Monitoring and control software for both *flow-line manufacturing* and *flexible manufacturing* can be implemented as an AOS. In this section, we show an AOS description of monitoring and control software for a flow-line manufacturing system by using a simple example shown in Fig. 10.

When a job (or workpiece) goes through this manufacturing line, it is first entered at stage s_1. Then, the job is moved by mover mA_1 to stage s_2, where it is inspected by vision system v. Vision system v gathers information required for the later processing by the other robots and machines. After the inspection of the job is complete, it is moved by mover mA_2 from stage s_2 to stage s_3, where it is worked on by robot rA. Then the job is sent to stage s_4 by mover mA_3.

The same type of robots, rB_1 and rB_2, are installed at stages s_4 and s_5, and one of them must process each job. When an incoming job is placed in stage s_4, mover mB_1 moves it immediately to

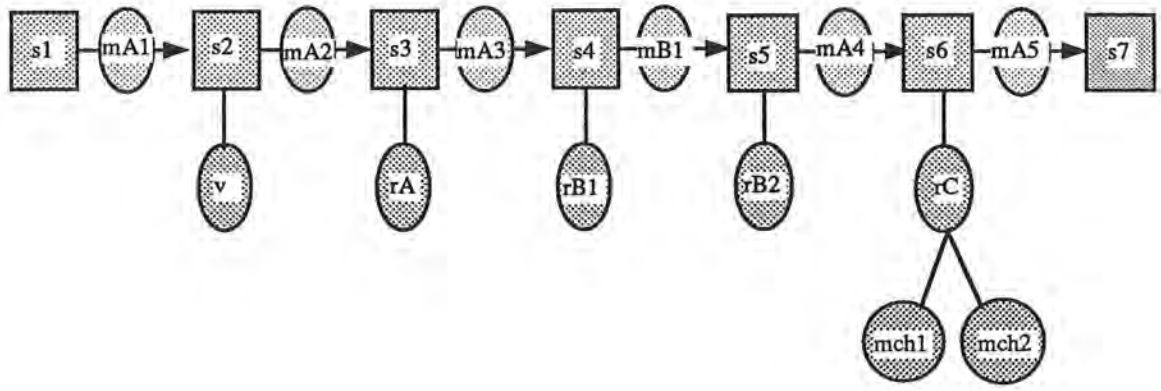

Figure 10: A manufacturing line.

stage s_5 if s_5 is empty. Otherwise, it is processed at stage s_4 by rB_1. Robot rC loads the job from stage s_6 onto machine mch_1 or mch_2, whichever is available. After the machine processing by mch_1 or mch_2 is complete, it returns the job back at stage s_6. Finally, mover mA_5 moves the job to stage s_7.

There is a possibility of a deadlock involving s_6, mch_1, and mch_2. A job must not be moved from s_5 to s_6 when both mch_1 and mch_2 are occupied. Otherwise, none of the jobs loaded on s_6, mch_1, and mch_2 can be moved.

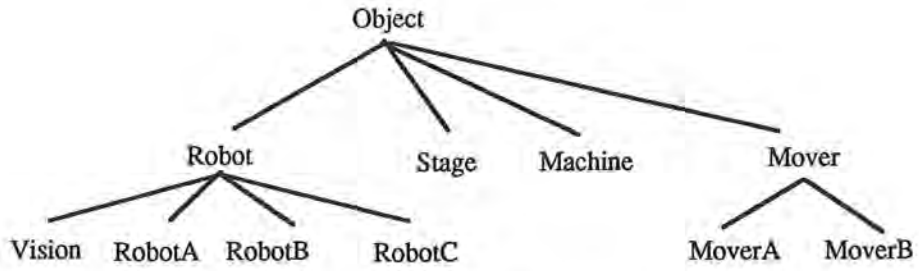

Figure 11: Classes for flow-line manufacturing.

The controller for this manufacturing line can be implemented as an AOS with the following objects: seven instances s_1-s_7 of class `Stage`, five instances mA_1-mA_5 of class `MoverA`, one instance mB_1 of class `MoverB`, one instance v of class `Vision`, one instance rA of class `RobotA`, two instances rB_1 and rB_2 of `RobotB`, one instance rC of class `RobotC`, and two instances of class `Machine`. These classes can be organized into the class hierarchy as shown in Fig. 11.

Fig. 12 is the main program of the AOS. It can be easily seen that the code corresponds exactly to the diagram shown as Fig. 10. The main program of an AOS shows only the structural relationships

```
MFMain {
   public:
      boolean systemRunning = true;
   private:
      Stage s1("s1", "initial stage");
      Stage s2("s2", "visual inspection");
      Stage s3("s3", "assembly1");
      Stage s4("s4", "assembly2");
      Stage s5("s5", "assembly2");
      Stage s6("s6", "packaging");
      Stage s7("s7", "final stage");

      MoverA mA1("mA1", 1000) with {inStage = s1; outStage = s2};
      MoverA mA2("mA2", 1010) with {inStage = s2; outStage = s3};
      MoverA mA3("mA3", 1020) with {inStage = s3; outStage = s4};
      MoverB mB1("mB1", 1030) with {inStage = s4; outStage = s5};
      MoverA mA4("mA4", 1040) with {inStage = s5; outStage = s6};
      MoverA mA5("mA5", 1050) with {inStage = s6; outStage = s7};

      Vision v("v", "visual inspection") with {stage = s2; rqsBlock = ...};
      RobotA rA("rB", "assembly1") with {stage = s3; rqsBlock = ...};
      RobotB rB1("rC1", "assembly2") with {stage = s4; rqsBlock = ...};
      RobotB rB2("rC2", "assembly2") with {stage = s5; rqsBlock = ...};
      RobotC rC("rD", "distributer") with {stage = s6; m1 = mch1; m2 = mch2};

      Machine mch1("m1", "packaging");
      Machine mch2("m2", "packaging");
}
```

Figure 12: Main program for flow-type manufacturing.

among its components which are connected by interface variables. Movers and robots directly move and manipulate jobs in the stages connected to them through their interface variables.

```
class Job {
   public:
      char *name;
      char *description;
      Job(char *n, char *dsc) {strcpy(n, name); strcpy(dsc, description);};
}
```

Figure 13: Class Job.

Fig. 13 shows the definition of class **Job**. It only contains its name and description. In a real system, it may have more detailed information.

```
enum StageState {empty, moving, arrived, ready, processing, processDone, wait};
class Stage {
   public:
      char *name, *description;
      StageState state = empty;
      Job *job = nil;
      Stage(char *n, char *dsc) {strcpy(n, name); strcpy(dsc, description);};
}
```

Figure 14: Class Stage.

Class **Stage** is defined in Fig. 14. It contains four instance variables and does not contain any behavior descriptions. Instance variables **name** and **description** contain the name and description, respectively, of a **Stage**. Instance variable **job** points to a job that is currently at that stage, and **state** represents the current state of the stage.

A robot or machine processes a job based on the state of the stage to which it is connected. A stage can be in six different states. The initial state of a stage without a job is **empty**. While a mover is using a stage to move a job from or to it, the value of **state** is set to **moving**. When a job has arrived, **state** is set to **arrived**, and when the job is ready for processing, **state** is set to **ready**. When a job is being processed by a robot, **state** becomes **processing**, and when the robot finishes the processing, **state** becomes **processDone**. The state **wait** is a state which blocks any movement of a job for some special reason, e.g., prevention of a deadlock.

There are two different kinds of movers in Fig. 10. The common class of these movers is class **Mover** defined in Fig. 15, and it has two subclasses **MoverA** and **MoverB**. The mover state is either **reset**, **moving**, or **moveDone**. A job pointer is also defined in **Mover**.

```
enum MoverState {reset, moving, moveDone};
class Mover {
   interface:
      Stage inStage, outStage;
   public:
      char *name;
      MoverState state = reset;
      IOddress ioAddress;
      Job *job = nil;
      Mover(char *n, IOAddress ioAdd) {strcpy(n, name); ioAddress = siAdd};
}
```

Figure 15: Class Mover.

```
class MoverA : public Mover {
   public:
      MoverA(char *n, IOAddress ioAdd) : (n, ioAdd) {};
   private:
      when ((inStage->state == processDone) && (outStage->state == empty)) {
         state = moving;
         outStage->state = moving;
         job = inStage->job;
         inStage->job = nil;
         inStage->state = empty;
         /* activate external process to move a job */
         activate(moveProgram, inStage, outStage, job);
      }

      when (state == moveDone) {
         outStage->job = job;
         outStage->state = arrived;
         job = nil;
         state = reset;
      }

      when (inStage->state == arrived)
         inStage->state = ready;
}
```

Figure 16: Class MoverA.

Fig. 16 shows a simple mover which just moves `job` from `inStage` to `outStage`. It consists of three transition rules.

First transition rule is activated when a job is processed by a robot in the previous stage and when the next stage is empty. It sets the states of itself and the next stage to `moving`, moves the job object to itself, sets the state of the previous stage to `empty`, and activates an external process to move the job.

When the real job is moved to the next stage, the external process changes the `state` of the mover to `moveDone`. Then the mover moves the job object to the next stage and changes the state of the next stage to `arrived`. The mover finally sets its `job` to `nil` and `state` to `reset`.

The last transition rule sets `state` of the previous stage to `ready` as soon as a job arrives there. Then processing of the job can be started at that stage.

```
class MoverB : public Mover {
   public:
      MoverA(char *n, IOAddress ioAdd) : (n, ioAdd) {};
   private:
      StageState inStageState;
      when ((outStage->state == empty)
          && ((inStage->state == processDone) || (outStage->state == arrived))) {
         state = moving;
         outStage->state = moving;
         job = inStage->job;
         inStage->job = nil;
         inStageState = inStage->state;
         inStage->state = empty;
         /* activate external process to move a job */
         activate(moveProgram, inStage, outStage, job);
      }

      when (state == moveDone) {
         outStage->job = job;
         outStage->state = inStageState;
         job = nil;
         state = reset;
      }

      when ((inStage->state == arrived) && (outStage->state != empty))
         inStage->state = ready;
}
```

Figure 17: Class MoverB.

An instance of class `MoverB` defined in Fig. 17 is a special mover. It schedules jobs for the two stages indicated by `inStage` and `outStage`. Each job need be processed at only one of them. The first

transition rule is activated even when the job is not processed in the first stage if the second stage is `empty`. The last rule changes the `state` of `inStage` to `ready` only if the `outStage` is occupied. The other transition rules are similar to those of `MoverA`.

Classes for robots and machines can be similarly defined.

4 Flexible Manufacturing

In this section, we show an AOS description of monitoring and control software for a *flexible manufacturing system* (FMS). One key difference of an FMS from a flow-line manufacturing system is that the jobstep scheduling must be global. The AOS approach is less suitable to an FMS than to a flow-line manufacturing system, where control is localized. Nonetheless, the AOS approach can be applied to an FMS.

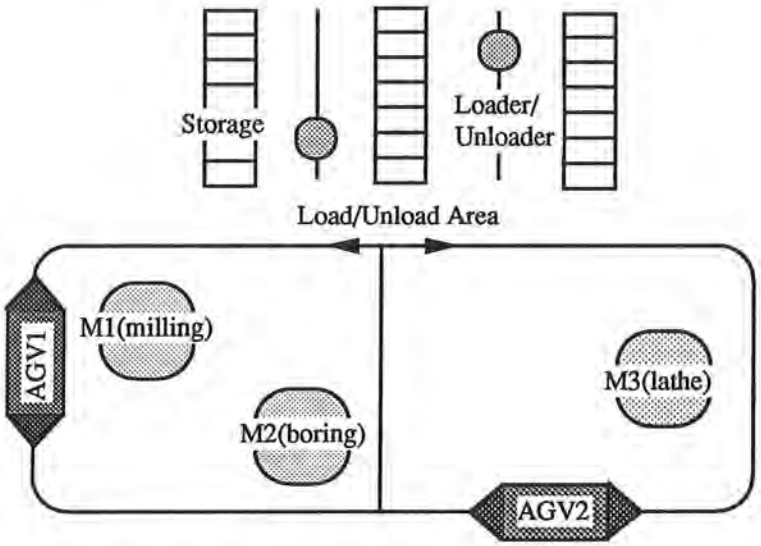

Figure 18: A flexible manufacturing system.

The FMS shown in Fig. 18 consists of three *machines*, two *automatic guided vehicles* (AGVs), and an *automated storage/retrieval* system.

Associated with each *job* is a sequence of *jobsteps* as shown in Fig. 19. After being retrieved from the storage by a *loader*, a job, which is embodied as a workpiece, is transferred by an AGV from a machine to another machine. Each machine performs a jobstep for the job.

Fig. 20 shows the timelines for the sequences of the AGV moves along with the machines' usage. A black area represents a period when a resource (machine, AGV, or loader) is actually processing or

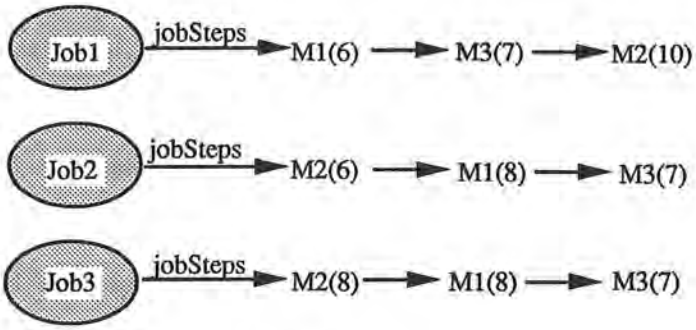

Figure 19: Jobsteps.

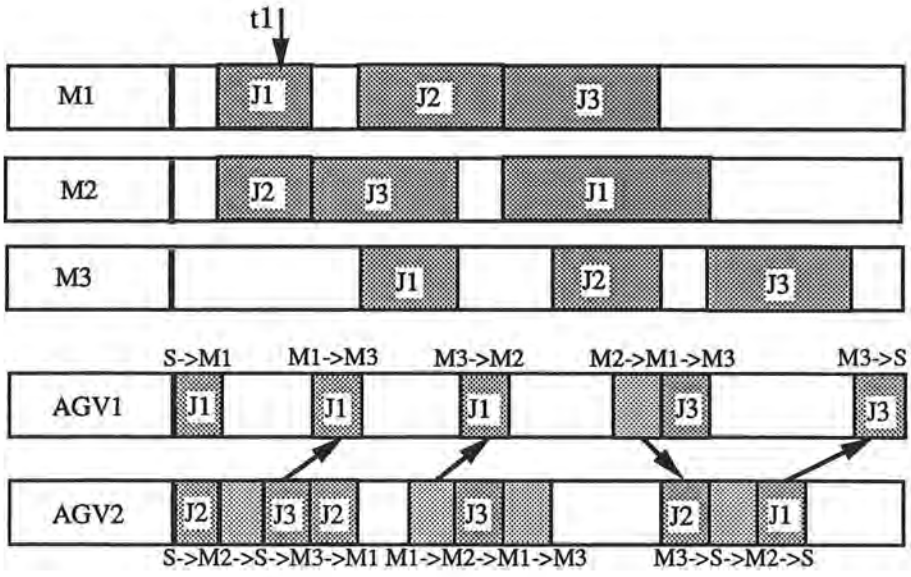

Figure 20: Timelines.

carrying a job, and a clear area represents a period when the resource is free. A grey area for an AGV represents a period when the AGV is moving but is not carrying a job.

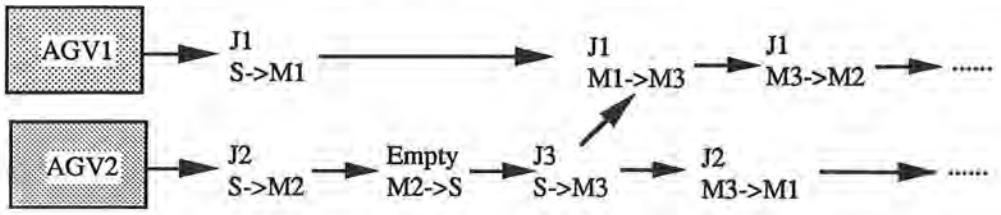

Figure 21: Jobstep schedule.

The sequences of the moves of the AGVs are maintained in the *jobstep schedule* for them. An example of a jobstep schedule is shown in Fig. 21. A jobstep record for an AGV designates the origin and destination of the job to be moved. The AGV jobstep schedule is produced by the *jobstep scheduler*. The jobstep scheduler uses an AI-based heuristic algorithm, which is beyond the scope of this paper, to schedule the moves of the AGVs so that machines and AGVs are utilized efficiently. After all the jobsteps of a job is completed, an AGV returns the job to the storage.

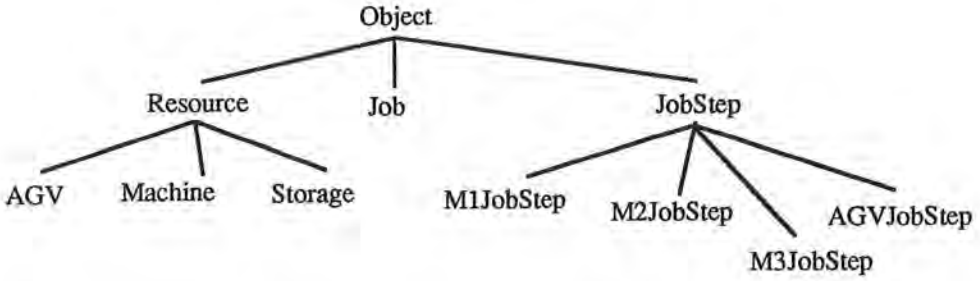

Figure 22: FMS Classes.

Fig. 22 shows the class hierarchy of the AOS classes for our flexible manufacturing system.

The declarative nature of an AOS is illustrated in Figure 23. All that are declared are the top-level components of the system. Three storage objects, three machines, two AGVs, and two loaders are created. Since all the active agents perform their operations according to the global schedule provided by the scheduler. No interconnections are provided among the top-level components.

The definition of class `Job` is given in Fig. 24. Each `Job` contains a pointer `jobsteps`, which points to the list of the `Jobsteps` constituting that job. A `Jobstep`, defined in Figure 25, contains a pointer `next` to the next jobstep and a pointer variable `resource` that identifies the resource (machine) required for that jobstep.

```
MFMain {
   Storage s1("S1", "storage", 100);
   Storage s2("S2", "storage", 100);
   Storage s3("S3", "storage", 100);
   Machine m1("M1", "milling");
   Machine m2("M2", "boring");
   Machine m3("M3", "turning");

   AGV agv1("AGV1", "automatic guided vehicle 1");
   AGV agv2("AGV2", "automatic guided vehicle 2");
   LDR ldr1("LDR1", "loader/unloader 1");
   LDR ldr2("LDR2", "loader/unloader 2");
}
```

Figure 23: FMS main program.

```
class Job {
   public:
      char *name, *description;
      Jobstep *jobsteps, *currentJobstep;
      Job(char *n, char *desc, Jobstep *head) {
         strcpy(n, name);
         strcpy(dsc, description);
         jobsteps = head;
         currentJobstep = head;
      };
}
```

Figure 24: Class Job.

```
class Jobstep {
   public:
      Jobstep *next;
      Resource *resource;
      Jobstep (Resource *r) {resource = r;};
}
```

Figure 25: Class Jobstep.

A subclass of `Jobstep` should be created to represent the unique data requirements of the jobsteps of each type of machines. For example, a drill needs x and y positions, a radius, and a depth to make a hole.

A `Machine` performs an action on a workpiece. Its behavior is defined by two transitions: `Start` and `Stop`. A `Storage` consists of a fixed-number of slots where workpieces are stored. A `Loader` carries a job between a storage slot and a load/unload area.

AGVs move all the jobs between the load/unload areas and the machines and between the machines. The class `AGV` is defined in Fig. 26. An `AGV` has three primary components: `jobsteps`, `job` (defined in `Resource`), and `agvState`. The sequence of the moves of each AGV is determined by `jobsteps`, which is the list of the jobsteps to be performed by the AGV.

An `AGVJobstep` contains `source` and `destination` for the AGV move. An `AGVJobstep` can be initiated only when its `refCnt` is 0, which indicates that all the `AGVJobsteps` that must precede it are complete. See Fig. 27 for the definition of AGVJobstep.

The class `LDR` is similar to `AGV`.

Fig. 28 shows the state of the system at time t_1 indicated in Fig. 20.

5 Conclusions

An actionbase system was proposed as a framework for manufacturing control software. Three major components of an actionbase system are the database, actions control, and user interface subsystems. The actions control and user interface subsystems are implented as active object systems. An active object system is constructed by structural composition of active objects whose behaviors are defined by the transition statements provided in their class definitions.

We showed how manufacturing control systems can be described as active object systems, by using as examples a flow-line manufacturing system and a flexible manufacturing system. Even these simple examples indicated the following advantages of the AOS approach. Active objects, with three kinds of transition statements, show better encapsulation and achieve more flexible inter-object communication than ordinary objects. Support of views makes easy the implementation of a user interface of a manufacturing control system.

Prototypes of manufacturing control systems based on the AOS approach are being implemented.

```
enum AGVState {reset, moving, moveDone};
Class AGV : public Resource {
   public:
      AGVJobstep *jobsteps;
      MoverState state = reset;
      AGV (char *n,  char *d, Resource *cStage) : (n, d) {};
      /* init move without a job */
      when ((state == reset) && jobsteps
          && (jobsteps->state == ready) && (jobsteps->mType == empty)) {
         state = moving;
      };
      /* if the move is done without a job */
      when ((state == moveDone) && (jobsteps->mType == empty)) {
         jobsteps->state = done;
         state = reset;
         /* get new jobstep and remove old jobstep */
         jobsteps *tmpJobstep = jobsteps;
         jobsteps = jobsteps->next;
         delete tmpJobstep;
      };
      /* unload a job from the machine to agv */
      when ((state == reset) && jobsteps && (jobsteps->state == ready)
          && (jobsteps->mType == loaded) && (jobsteps->source->state == processDone)) {
         /* let the source know that the agv is arrived and init unloading */
         jobsteps->source->agv = this;
         jobsteps->source->state = unloading;
      };
      /* if the job is unloaded from the source, init moving */
      when (jobsteps->source->state == unloadDone) {
         /* move to the destination */
         jobsteps->source->agv = nil;
         state = moving;
      };
      /* if move is done */
      when ((state == moveDone) && (jobsteps->mType == loaded)
           && (jobsteps->destination->state == reset)) {
         /* let the source know that the agv is arrived and init loading */
         jobsteps->destination->agv = this;
         jobsteps->destination->state = loading;
      };
      /* if the job is loaded to the destination */
      when (jobsteps->destination->state == loadDone) {
          /* reset the states */
         jobsteps->state = done;
         state = reset;
         jobsteps->destination->agv = nil;
         /* get new jobstep and remove old jobstep */
         jobsteps *tmpJobstep = jobsteps;
         jobsteps = jobsteps->next;
         delete tmpJobstep;
      };
}
```

Figure 26: Class AGV.

```
enum AGVJobstepState {notReady, ready, done};
struct JobstepRefNode {
   AGVJobstep *jobstep;
   JobstepRefNode *next;
};
class AGVJobstep : public Jobstep {
   public :
      Resource *source = nil, *destination = nil;
      MoveType mType;
      AGVJobstepState state = notReady;
      JobstepRefNode *refNodes;
      int refCnt;
      AGVJobstep (Resource *s, Resource *d, MoveType t, int rc, JobstepRefNode rn) : (AGV) {
         source = s;  destination = d;
         mType = t;   refCnt = rc;
         refNodes = rn;
      };
      /* decrement the reference counter of the jobsteps dependent on this */
      on complete() from self do {
         for (JobstepRefNode *tmp=refNodes; tmp; tmp=tmp->next)
            tmp->refCnt--;
      }
      /* if refence counter is 0, then it becomes ready */
      when (refCnt == 0 and state != ready) {
         state = ready;
      }
}
```

Figure 27: Class AGVJobstep.

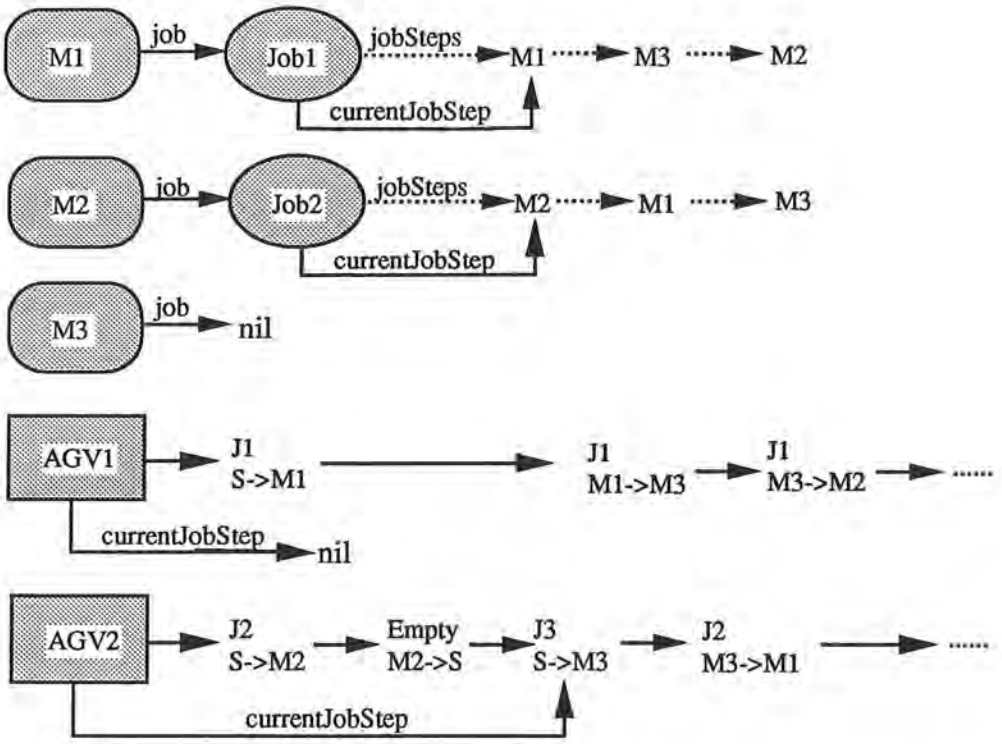

Figure 28: System state.

The major portions of these implementations are user interfaces, for which *active-object user interface system* (AOUIS) [CHOI91b] is being used.

References

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

[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.

[CHOI91a] Choi, S. and Minoura, T. Active object system. Tr. 91-40-1, Dept. of Computer Science, Oregon State Univ., 1991.

[CHOI91b] Choi, S. and Minoura, T. User interface system based on active objects. Tr. 91-60-5, Dept. of Computer Science, Oregon State Univ., 1991.

[COOT88] Coote, S., et al. Graphical and iconic programming languages for distributed process control: An object oriented approach. In Proc. 1988 IEEE WORKSHOP on Visual Languages, 1988, pp. 183-190.

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

[INFO88] *Informix-4GL User's Guide, Version 1.10,* Informix part number 200-501-0004, Informix Corporation, 1988.

[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.

[NAYL87] Naylor, A. W., and Volz, R. A. Design of integrated manufacturing system control software. *IEEE Trans. on Systems, Man, and Cybernetics, SMC-17,* 6, 1987, 881-897.

[ORAC87] *Oracle SQL* Forms Designer's Tutorial,* Oracle Reference Manual Number 3302-V2.0, Oracle Corporation, 1987.

[ORAC88] *Oracle For MacIntosh Manual, Version 1.0,* Part No. 5117-V1.0, Oracle Corporation, 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.

[UNIF87] *Accell Integrated Application Development System,* Unify Reference Manual Number 254A, Release 1.3, Unify Corporation, 1987.

[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.