

AD-A262 964



DTIC
ELECTE
APR 15 1993
S C D

FINAL REPORT

DLA900-87-D-0017, DO 0019

ANALYSIS AND IMPROVEMENT OF PINNING AND
CUTTING RESOURCE SCHEDULING

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

John D. McGregor, Principal Investigator
and David A. Sykes
Department of Computer Science
Clemson University
Clemson, SC 29634-1906

93 3 23 095

93-06055
 146PR

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE 2/10/93		3. REPORT TYPE AND DATES COVERED 7/1/91 - 12/31/92	
4. TITLE AND SUBTITLE "Analysis and Improvement of Pinning and Cutting Resource Scheduling"				5. FUNDING NUMBERS DLA900-87-D-0017 DO 0019 (C)	
6. AUTHOR(S) John D. McGregor David A. Sykes					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Clemson Apparel Research 500 Lebanon Road Pendleton, SC 29670				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Logistics Agency Cameron Station Alexandria, VA 22304-6100				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The Clemson Interactive Planner is a software decision support tool useful in planning the use of pinning and cutting resources in a cutting room. The current version of the software is implemented in Objectworks/Smalltalk, an implementation of the Smalltalk-80 programming language that is available on a variety of platforms, including the IBM PC, Apple Macintosh, and Sun SPARCStation. The current implementation was done on a Macintosh and the examples in the following manual are taken from that environment. All environments are essentially the same; only the "look and feel" of the windowing system varies across platforms.					
14. SUBJECT TERMS Clemson Interactive Planner, pinning & Cutting				15. NUMBER OF PAGES 143	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL		

Clemson Interactive Planner (CIP) User's Manual

John D. McGregor, Principal Investigator
and
David A. Sykes
Department of Computer Science
Clemson University
Clemson, SC 29634-1906

December 1992

DTIC QUALITY ASSURANCE

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

St-A per telecon, Ms. Kerlin, DLA
Alex., VA 22304
4-15-93 JK

Contents

1	Introduction	3
1.1	Purpose	3
1.2	Scope	3
1.3	Background	4
2	Running a Simulation	8
2.1	Creating a Scenario	8
2.1.1	Tasks and Work Assignments	13
2.1.2	Plans	15
2.1.3	Cutting Room	16
2.1.4	Simulation Runs	17
2.2	Running Simulations	17
2.3	Customizing the System	28
A	Installation	29

List of Figures

1.1	A sample cutting room.	5
2.1	The initial screen for a simulation run.	18
2.2	The screen for a simulation run after restart.	20
2.3	The screen for a simulation run after one step.	21
2.4	The screen for a simulation run after two steps.	22
2.5	The screen for a simulation run after three steps.	23
2.6	The screen for a simulation run after four steps.	24
2.7	The screen for a simulation run after five steps.	25
2.8	The screen for a simulation run after eight steps.	26
2.9	The screen for the end of a simulation run.	27

Chapter 1

Introduction

1.1 Purpose

The Clemson Interactive Planner (**CIP**) is a software decision support tool useful in planning the use of pinning and cutting resources in a cutting room. This manual describes how to use **CIP**.

The current version of the software is implemented in Objectworks\Smalltalk, an implementation of the Smalltalk-80 programming language that is available on a variety of platforms, including the IBM PC, Apple Macintosh, and Sun SPARCStation. The current implementation was done on a Macintosh and the examples in this manual are taken from that environment. All environments are essentially the same; only the "look and feel" of the windowing system varies across platforms.

1.2 Scope

The information in this document describes the the current implementation developed under the sponsorship of the Defense Logistics Agency and Clemson Apparel Research. The current implementation is a prototype and contains only a relative few of the features the final system should contain. It is not suitable for production use, primarily because the user interface is primitive, but also because some of the class definitions required are incomplete.

This prototype has been produced as part of a research effort to use object-oriented technologies to adapt simulations easily to a wide variety of cutting room environments. The implementation—and this manual—focus on those portions that have been implemented and provide a look at how a production system might look. Please note that software still needs to be written. The information in this manual corresponds to the state of the system at the end of current funding of the project under which it was developed. This document

and the document *C/IP System Architecture* serve to document the status of the project at that time.

Instructions on installing the prototype system are provided in Appendix A.

1.3 Background

C/IP is a simulation system intended as a tool for assisting in the planning of the allocation of resources in a cutting room. Pinning and cutting operations in an apparel plant are responsible for producing the pieces of fabric that go into the construction of an apparel item. These operations are performed in a cutting room using a variety of automatic and manually-operated machines designed to spread fabric in multiple layers and cut them. A cutting room manager must make a variety of decisions with respect to how the various resources—operators, workstations, and materials—are scheduled to produce cut bundles on time and at low cost.

The cutting room's function includes the processing of rolls of fabric into the pieces required for a garment's manufacture. A typical men's dress shirt, for example, comprises about sixteen pieces that are sewn together before buttons are attached. Each of these pieces is produced by the cutting room in accordance with a set of requirements derived from a wide range of parameters such as fabric, fabric pattern, garment size, and garment style. The primary goal of the cutting room manager is to supply these pieces by the time they are needed by the sewing room, but to supply them at the lowest cost both in terms of labor and of fabric utilization (i.e., minimize the amount of scrap).

The scheduling of cutting room resources is a difficult problem complicated by many factors. A cutting room contains equipment designed to spread and cut fabric. Machines have been developed to spread fabric in multiple layers on a table. Other machines have been developed to cut through the layers of spread fabric, thereby producing dozens of cut pieces at a time. Some of these machines are operated by hand and others are operated under computer control. A spread up to sixty feet long can be fed automatically through a cutter under computer control to produce the cut pieces processed by the sewing rooms. Some specialized dye cutters can both spread and cut, needing only a single operator to monitor the machine.

A variety of machines can be found in a typical cutting room. Figure 1.1 shows a sample cutting room with three tables supporting automatic spreading, a BITE cutter that can be positioned at the end of any of these three tables and process a spread of arbitrary length, a table that supports automatic spreading and pinning but manual cutting, and a smaller table that supports manual spreading and manual cutting. Pinning is required for garments that have "engineered placements"—for example, plaid trousers for which the plaid pattern must match between the left and right halves of the garment, meaning that the cuts must be matched to the pattern. Pins are used to keep the pattern aligned

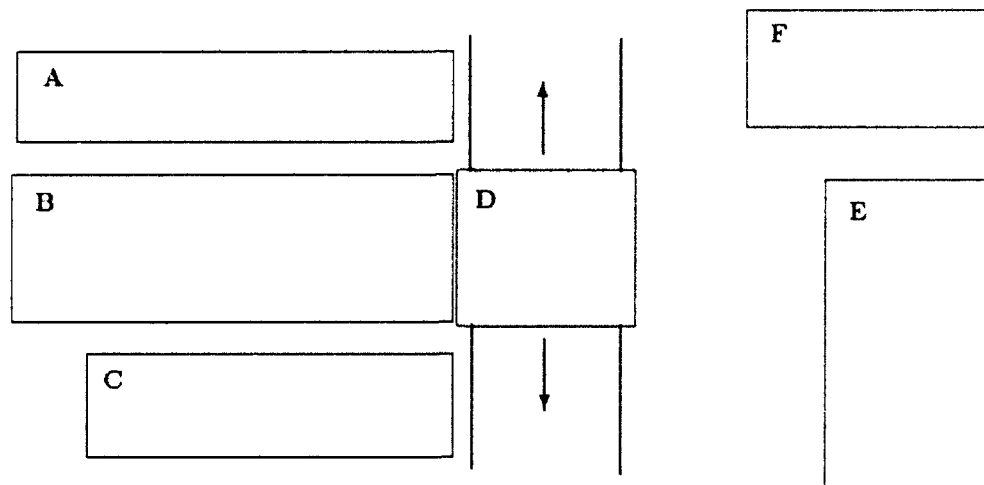


Figure 1.1: A sample cutting room. A, B, C: tables with automatic spreaders; D: a BITE cutter that can be positioned at the end of any of three tables and process a spread of arbitrary length; E: a table that supports automatic spreading and pinning but manual cutting; and F: a smaller table that supports manual spreading and manual cutting.

on all of the layers of fabric in a spread.

Among the decisions that a cutting room manager makes are:

- which operator to assign to which task. Each operator has different skills and efficiencies that can be applied to the tasks that need to be done.
- whether operators should be scheduled to work overtime or perhaps whether another shift should be added. If the workload is heavy, then the cutting room might have to work more hours. Overtime raises production costs.
- how to reallocate resources when an operator doesn't show up for work, an operator leaves work early, or a machine goes down unexpectedly.
- which fabric to spread on which table. Once a spread is begun, it cannot be moved to another table without introducing wrinkles, although most spreads can be slid to another part of the same table. Most spreads do not occupy the entire length of a table and more than one spread can be placed on a table. However, if the cutting is to be performed by an automatic cutter, then the cutting can only be done in the order that the

spreads lie on the table since the cutter is fed a spread from one end of the spreading table. Often a style—for example, jeans, tee shirts, dress shirts, etc.—requires several fabrics and spreads. Consider a shirt with a striped front and a polka-dot collar. Spreading must be planned such that the cutting for all the striped parts can be done near the time that the cutting for all the polka-dot parts is done so that the shirts can be sewn without a need to store any cut parts for them for very long.

Some spreads are limited to certain tables because of the fabric qualities or how cutting must be done. For example, only one automatic spreader in a cutting room might be suitable for knit fabrics or a pinning table must be used for spreading a plaid fabric to meet engineered placement specifications.

- which fabric to spread next. Some spreads take a long time—for example, spreads that require pinning for engineered placements, spreads that have many plies, spreads of very thin fabrics, or spreads that cannot be done using automatic spreading equipment—and tie up resources for a long period of time.
- how much variance to use for a cut. If the fabric available is not sufficient to produce an exact order, a decision must be made concerning how much more or less to cut. If a relatively small amount of fabric will be left after a cut, then a decision must be made concerning how many more pieces to cut (and which size(s)) in order to minimize scrap.
- which production orders can be combined. Fabric utilization can be improved by combining two or more production orders that use the same fabric as long as there is sufficient fabric available. This combination can be achieved either by placing markers end-to-end on a spread or by combining the individual markers into a single marker¹. Fabric utilization is usually improved if the latter technique is used, but this technique takes more lead time in order to produce a marker. Lying markers end-to-end produces fabric savings by reducing waste on the cut ends of a spread.
Production orders requiring the same cuts can sometimes be incorporated into a single spread. This is desirable because it reduces cutting time and can save fabric.
- which spread to cut next. As illustrated in figure 1.1, a cutter may be shared by multiple spreading tables. A decision must be made as to which spread to cut based on the time that will be required for cutting and subsequent needs for the spreading resources that will be freed up after the spread is cut.

¹ A marker is a layout of the pieces to be cut from a given length of fabric, usually plotted on a large piece of paper and placed atop a spread to identify the cuts. Labels printed on the paper pieces identify—or *mark*—the cuts that are produced.

- what to do if a production order for a sample arrives. The production of sample garments generally takes priority over regular production and preempts jobs in progress.

The development of a cutting plan for some period of time—for example, a day or a week—is an art generally based on a cutting manager's experience. Good cutting room managers are difficult to find and, as a relatively scarce commodity, difficult to keep at a plant. Our goal is to help a cutting room manager make decisions about cutting room resource utilization such that costs are minimized and fabric utilization is maximized. Our solution is a decision support tool that simulates the cutting room in order to predict the outcome of various plans.

Chapter 2

Running a Simulation

In this chapter, we explain how to create a scenario. A scenario consists of a cutting room and a plan for allocating the resources—equipment, materials, and operators—to the work assignments to be accomplished.

2.1 Creating a Scenario

Putting a scenario together requires describing a cutting room, defining resources, and creating a plan to be used as the basis for a simulation. In this section, we will describe the construction of a scenario involving a cutting room with two tables that can be used for both spreading and cutting. Three operators, designated as Huey, Dewey, and Louie, are available for work. Rolls of fabric are available for the various tasks to be performed. Construction of other scenarios parallel construction of this example. [The file `scenario.txt` on the distribution diskette contains the code necessary to construct and run this scenario.]

A scenario is constructed by a sequence of Smalltalk statements. The current implementation does not include an explicit class to represent a scenario, though perhaps such a class should be provided. Nor does this implementation provide any interactive support for describing a cutting room¹. The approach to defining a scenario in Smalltalk provides greater flexibility during prototype development, but probably more flexibility than is required for the typical user. We expect that, in practice, the need to use Smalltalk will be supplanted by the use of interactive tools and dictionaries of resources.

¹Ideally, a cutting room would be described by arranging pieces of equipment on an area of a computer monitor screen.

Resources

The first step in the construction of a scenario is the definition of the resources. Resources comprise the equipment, operators, and rolls of fabric and other materials that are found in the cutting room.

The segment of code below defines the resources in the example scenario.

Allocate equipment resources:

```
table1 := Table makePair: 'Table 1'.  
table2 := Table makePair: 'Table 2'.  
equipment := OrderedCollection  
    with: table1 with: table2.
```

We define the two pieces of equipment and put them in an ordered collection. This collection will be used to establish priorities in the simulation, but the ordering is really unimportant.

The message takes a string designating the name of the table as its argument. The message designation is taken from DEVS, and is admittedly not very user-friendly.

Define three operators, Huey, Dewey, and Louie.

```
huey := Operator new
  name: 'Huey'.
  number: 1.
  status: #idle.
  skills: (Dictionary new
    at: table1 put: 1 00.
    at: table2 put: 1 00.
    yourself).
  image: (Image
    extent: 16 @ 16
    depth: 1
    palette: MappedPalette whiteBlack
    bits: #[
      16r01 16rc0 16r02 16r20
      16r04 16r10 16r09 16r48
      16r08 16r08 16r09 16rc8
      16r05 16rd0 16r02 16r20
      16r01 16rc0 16r07 16r70
      16r18 16r0c 16r61 16r43
      16r45 16r51 16r49 16rc9
      16r49 16r49 16r49 16r49]
    pad: 16) reflectedInX.
  yourself.
```

*Define operator Huey,
employee number 1, having
100% efficiency at both
tables. The image is used
to represent this operator
on the screen. (Refer to
Objectworks documentation
for details on image
creation.)*

```

dewey := Operator new
  name: 'Dewey';
  number: 2;
  status: #idle;
  skills: (Dictionary new
    at: table1 put: 1.00;
    at: table2 put: 1.00;
    yourself);
  image: (Image
    extent: 16 @ 16
    depth: 1
    palette: MappedPalette whiteBlack
    bits: #[
      16r01 16rc0 16r02 16r20
      16r04 16r10 16r09 16r48
      16r08 16r08 16r09 16rc8
      16r05 16rd0 16r02 16r20
      16r01 16rc0 16r07 16r70
      16r18 16r0c 16r60 16rc3
      16r45 16r51 16r49 16r49
      16r49 16r49 16r48 16rc9 ]
    pad: 16) reflectedInX.
yourself.

```

*Similarly, define operator
Dewey*

```

louie := Operator new
  name: 'Louie';
  number: 3;
  status: #idle;
  skills: (Dictionary new
    at: table1 put: 1.00;
    at: table2 put: 1.00;
    yourself);
  image: (Image
    extent: 16 @ 16
    depth: 1
    palette: MappedPalette whiteBlack
    bits: #[
      16r01 16rc0 16r02 16r20
      16r04 16r10 16r09 16r48
      16r08 16r08 16r09 16rc8
      16r05 16rd0 16r02 16r20
      16r01 16rc0 16r07 16r70
      16r18 16r0c 16r60 16r43
      16r44 16r51 16r48 16r49
      16r48 16r49 16r49 16rc9]
    pad: 16) reflectedInX;
  yourself.

```

And operator Louie

```

operators := ResourceCollection
  with: huey with: dewey with: louie.

```

*A resource collection is a
collection of resources that
is able to display itself as a
collection in a view.*

Create some fabrics and rolls of fabrics:

(cotton := Material new)
description: '100% cotton'.

All fabrics are 100% cotton

redCotton := Fabric
color: Color red weave: nil
weight: 1 thickness: 0.0625 inches
material: cotton defectRate: 0.0.

greenCotton := Fabric
color: Color green weave: nil
weight: 1 thickness: 0.0625 inches
material: cotton defectRate: 0.0.

blueCotton := Fabric
color: Color blue weave: nil
weight: 1 thickness: 0.0625 inches
material: cotton defectRate: 0.0.

*Create three rolls of cotton,
one green, one blue, and
one red.*

greenRoll := FabricRoll
fabric: greenCotton
length: 500 feet width: 54 inches.

blueRoll := FabricRoll
fabric: blueCotton
length: 200 feet width: 54 inches.

redRoll := FabricRoll
fabric: redCotton
length: 1000 feet width: 60 inches.

greenRoll name: 'green cotton'.
blueRoll name: 'blue cotton'.
redRoll name: 'red cotton'.

*The length and width of
each roll is specified, and
each roll is given a name.
Note that units of feet,
yards, and inches are used.
All lengths must include
the units of measure.*

materials := ResourceCollection
with: greenRoll with: blueRoll with: redRoll.

*Put the rolls of fabric in a
resource collection.*

2.1.1 Tasks and Work Assignments

Once resources have been defined, they can be used in work assignments. A work assignment is the association of a task with the equipment, operators, and

materials. There are currently four kinds of tasks in *C&IP* : spreading, cutting, moving, and bundling. Each task has a different class defined for it. The results of a work assignment is represented by a ticket. So, for example, if fabric is to be spread and then cut, the result of the spreading task is associated with a ticket and that ticket serves as one of the materials required for the cutting task. Tickets, among other things, prescribe an ordering to work assignments.

```
ticket1 := Ticket new: 'Ticket 1'.
ticket2 := Ticket new: 'Ticket 2'.
ticket3 := Ticket new: 'Ticket 3'.
```

Create three tickets now for convenience.

```
spreadingTask1 := SpreadingTask
  spreadTemplate: (SpreadTemplate
    new: 4 layersOf: greenCotton
    ofWidth: 54 inches
    ofLength: 60 feet)
  rolls: (ResourceCollection
    with: greenRoll with: blueRoll)
  marker: nil.
```

There are four tasks to be performed:

```
spreadingTask2 := SpreadingTask
  spreadTemplate: (SpreadTemplate
    new: 10 layersOf: redCotton
    ofWidth: 60 inches
    ofLength: 30 feet)
  rolls: (ResourceCollection with: redRoll)
  marker: nil.
```

- *spread 4 layers of green cotton*
- *spread 10 layers of red cotton*
- *spread 6 layers of red cotton*
- *spread 6 layers of blue cotton*

```
spreadingTask3 := SpreadingTask
  spreadTemplate: (SpreadTemplate
    new: 6 layersOf: redCotton
    ofWidth: 60 inches
    ofLength: 3 yards)
  rolls: (ResourceCollection with: redRoll)
  marker: nil.
```

A spread template defines the length, width, number of plies, and composition of each ply.

```
spreadingTask4 := SpreadingTask
  spreadTemplate: (SpreadTemplate
    new: 6 layersOf: blueCotton
    ofWidth: 54 inches
    ofLength: 50 feet)
  rolls: (ResourceCollection with: blueRoll)
  marker: nil.
```

2.1.2 Plans

Once the tasks are set and the resource sets are defined, a plan is constructed. A plan comprises a sequence of work assignments, each designating a task, a set of one or more operators to work together on the task, and a workstation at which the task is to be performed. A workstation is a section of a piece of equipment and can vary from assignment to assignment. For example, a spread might be constructed on two-thirds of the left end of a table and another spread might be constructed on the right two-thirds. Two workstations at the same table are involved. Obviously, one of these tasks cannot be started until the other spread has been moved. Thus, the demand for workstations at the same piece of equipment specifies a partial ordering of work assignments.

A workstation is designated as two percentages of the width and length of a piece of equipment, which is always assumed to have some rectangular shape. This area is referred to as a *section*. Thus, the entire equipment can be a single workstation, designated as width: 0-100%, length: 0-100%. The left half of a workstation is designated as width: 0-100%, length: 0-50%.

A plan must be given a start time. This time sets the context for a simulation run. The time is significant because workers have work schedules (default is 8:00 A.M. to 5:00 P.M. with an hour for lunch at noon and 15-minute breaks at 10:00 and 3:00).

```

plan := Plan new.
plan startTime: (SimulationTime
  date: (Date newDay: 1
    month: #October year: 1992)
    time: (Time fromSeconds: 0)).
wa1 := WorkAssignment
  workstation: (Workstation
    at: table1 section:
      (Section origin: 0 0 corner: 0.5 1))
  operators: (Set with: huey)
  task: spreadingTask1
  ticket: ticket1.
wa2 := WorkAssignment
  workstation: (Workstation
    at: table2 section:
      (Section origin: 0.25 0.25
        corner: 0.75 0.75) )
  operators: (Set with: louie)
  task: spreadingTask2
  ticket: ticket2.
wa3 := WorkAssignment
  workstation: (Workstation at: table2)
  operators: (Set with: dewey)
  task: spreadingTask3
  ticket: ticket3.
wa4 := WorkAssignment
  workstation: (Workstation
    at: table1 section:
      (Section origin: 0.5 0 corner: 1 1))
  operators: (Set with: huey)
  task: spreadingTask4.

plan add: wa1; add: wa2; add: wa3; add: wa4.

```

Define four work assignments, one for each task.

Put the work assignments in a plan. Order matters.

2.1.3 Cutting Room

The final step in the construction of a scenario is the placing of the plan and resources in a cutting room. The cutting room itself is a DEVS model, and the argument to `makePair` is the name of the cutting room.

```

cuttingRoom := (CuttingRoom makePair: 'ACME Cutting Room'
    containing: equipment
    operators: operators
    materials: materials).
cuttingRoom plan: plan.

```

2.1.4 Simulation Runs

A simulation run is created by associating a root coordinator with the cutting room model, specifying a starting time for the simulation, and specifying the time at which simulation should stop. Finally, the model is associated with a simulation window, an instance of the class `SimulationRunView`, from which simulation is controlled as explained in Section Running Simulations.

```

scenario := RootCoordinator new: 'RC:ACME Cutting Room'.
startTime := (SimulationTime
    date: (Date newDay: 1 month: #October year: 1992)
    time: (Time fromSeconds: 0)).
scenario
    startTime: startTime;
    timeLimit: (startTime addDuration: 5 days);
    linkToParent: (cuttingRoom processor).
Cursor wait
    showWhile: [SimulationRunView tryOn: scenario ].

```

2.2 Running Simulations

We have just described the specification of a scenario and the method for creating of a window to control simulation. The simulation view provided by the current implementation is shown in Figure 2.1.

The window labeled "CLIP: RC:ACME Cutting Room" contains the following components:

- a **Restart** button used to reset all the models in the run.
- a **Go** button used to start or resume a simulation run.
- a **Pause** button used to interrupt a simulation run during execution.

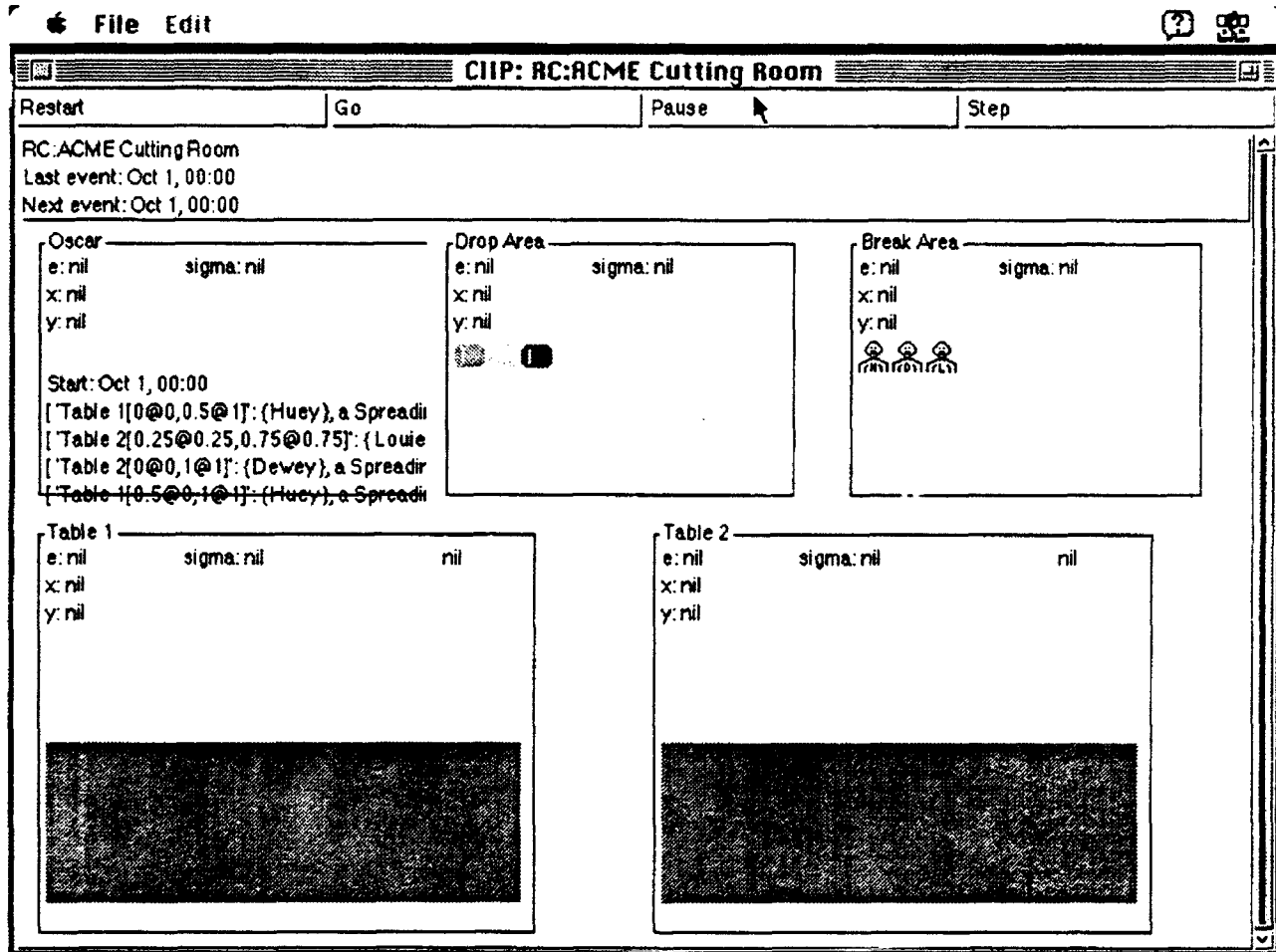


Figure 2.1: The initial screen for a simulation run.

- a Step button used to single-step a simulation, meaning to execute one iteration of the loop being performed by the root coordinator controlling the simulation.
- a text view (under the buttons) that provides the time of the last simulation event and of the next simulation event. These values are maintained by the root coordinator.
- a set of views, one for each atomic model in the system—that is, one for the various components of the cutting room. The default arrangement is for the planner and dispatchers to show across the top line and for the equipment to show across the bottom line. Each view is labeled by its name and each provides a graphical representation of its current status. All models show the values of e , x , y , and σ associated with the management of atomic models. Below this information, each view reflects model-specific information:
 - Oscar is a planner and shows the current plan. [A graphical representation would be preferable as would a scrollable view, but time did not permit their implementation.]
 - Drop Area is a dispatcher of materials. The three cylindrical objects shown represent rolls of fabric. The horizontal representation of each roll shows that the roll is currently not in use. A vertical orientation designates that the roll is in use (or about to be since more than one iteration of the root coordinator can apply to a given simulation time).
 - Break Area is a dispatcher of operator resources. The three icons represent Huey, Dewey, and Louie. These operators are shown idle since they are outlined in black.
 - Table 1 shows a rectangular area shaded gray. This represents the surface of the table. An active workstation at a table is shown shaded black. The nil showing in the upper right-hand corner reflects the current phase of the model.
 - Table 2 is similar to Table 1 and is, in fact, of the same class.

When a window for a simulation run is first displayed, no restart has been done on the root coordinator. Consequently, all values for simulation-related information shows nil.

Figure 2.2 shows the window after the Restart button has been pressed. Note that some of the fields have changed from nil to duration values, indicating the time to the next internal transition. The tables show they are in the #passive phase—that is, no work is in progress.

Once a run has been restarted, either of the Go or Step buttons may be used to run the simulation. The Go button is equivalent to repeated pressing of the

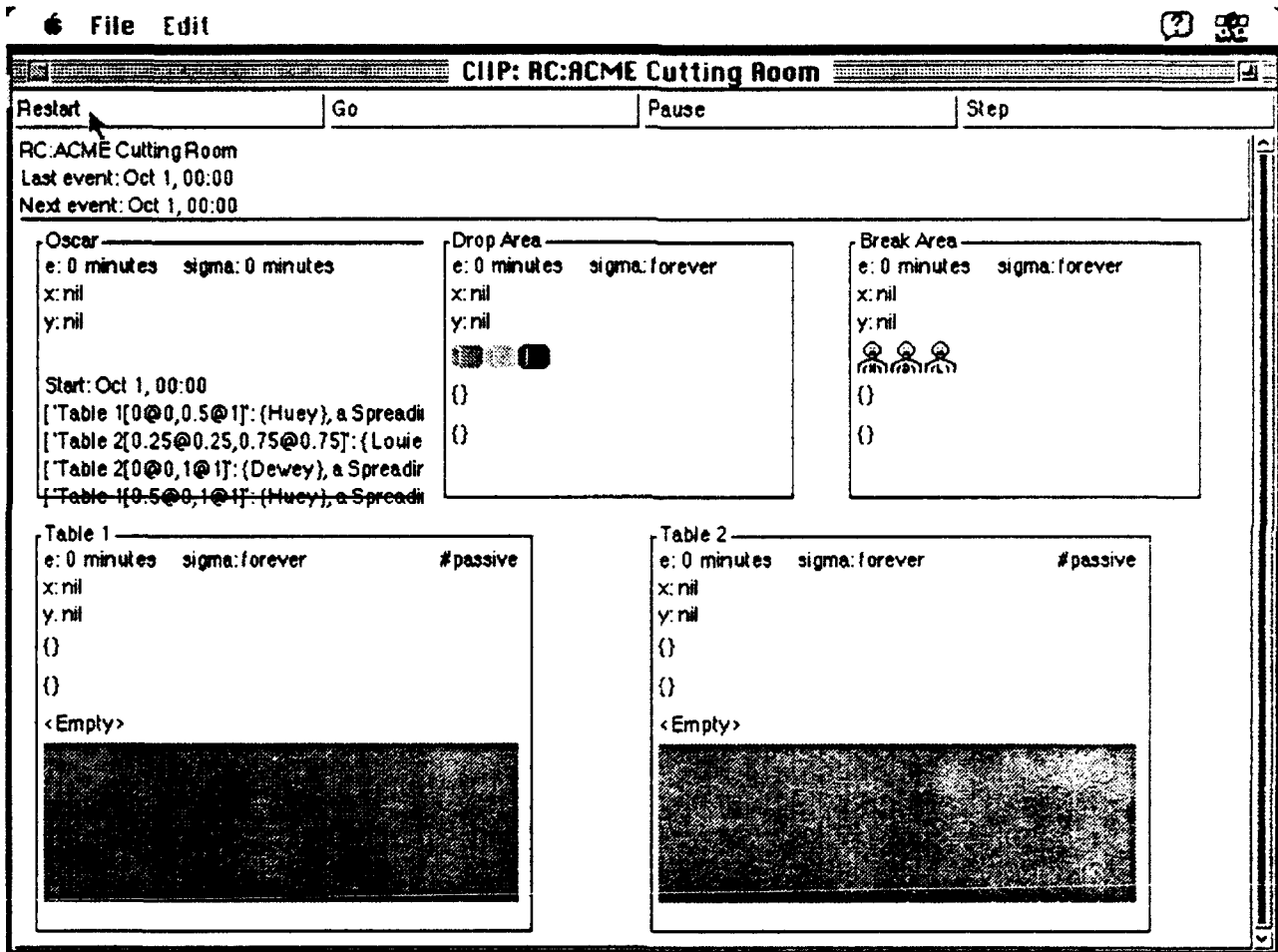


Figure 2.2: The screen for a simulation run after restart.

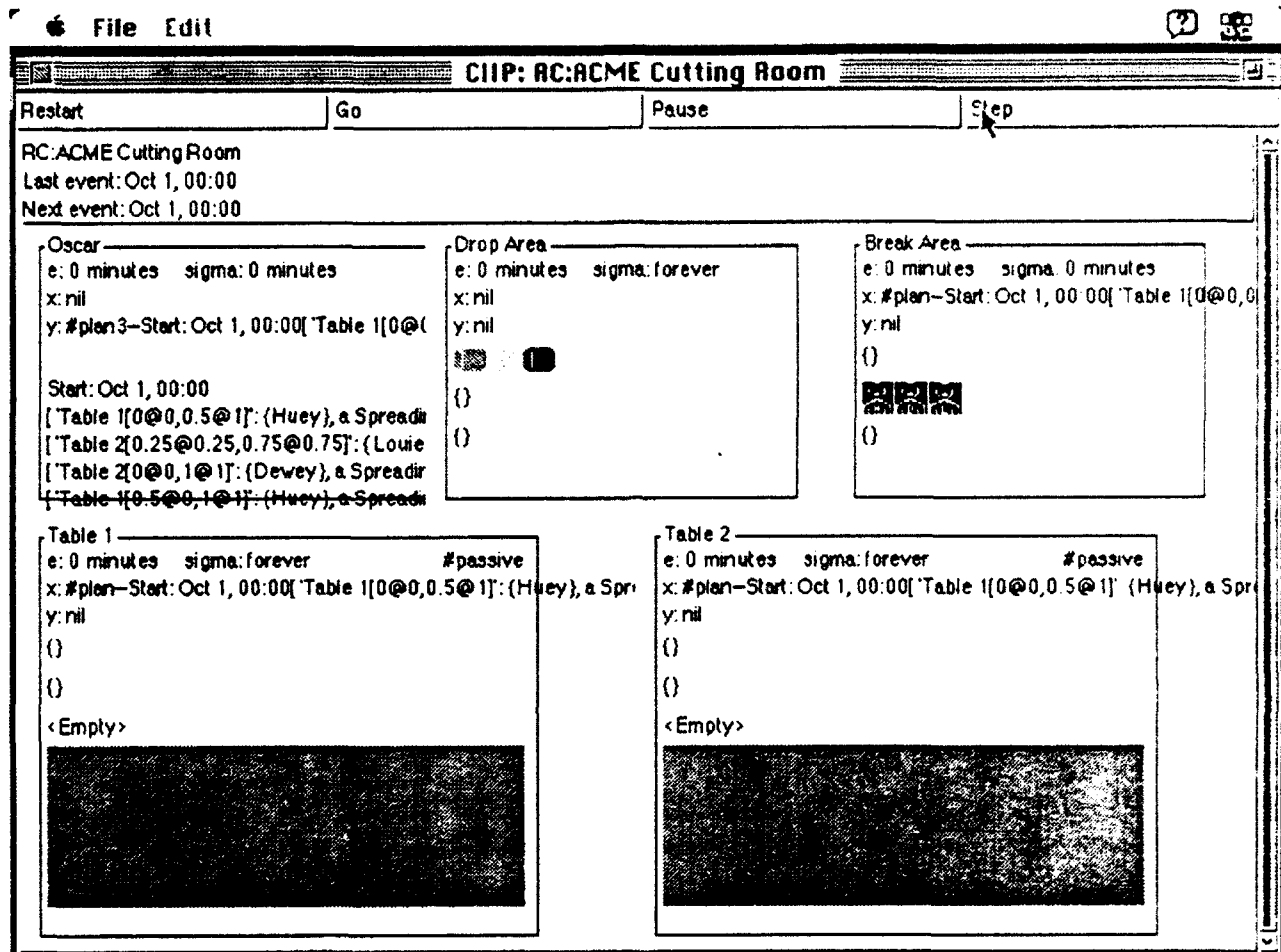


Figure 2.3: The screen for a simulation run after one step.

Step button. The Pause button may be used to interrupt a "Go"-ing simulation run.

Figure 2.3 shows the window after the Step button has been pressed once. On this first step, the break area has determined that all three operators are on duty and can be dispatched to the various tables to await the arrival of materials and the availability of workstations. This determination is reflected in the fact that the operator icons are outlined in white.

Figure 2.4 shows the window after the Step button has been pressed again. Note how operator Huey (designated by the "H" on the icon) has been dispatched from the break area to Table 1, the equipment at which his next work

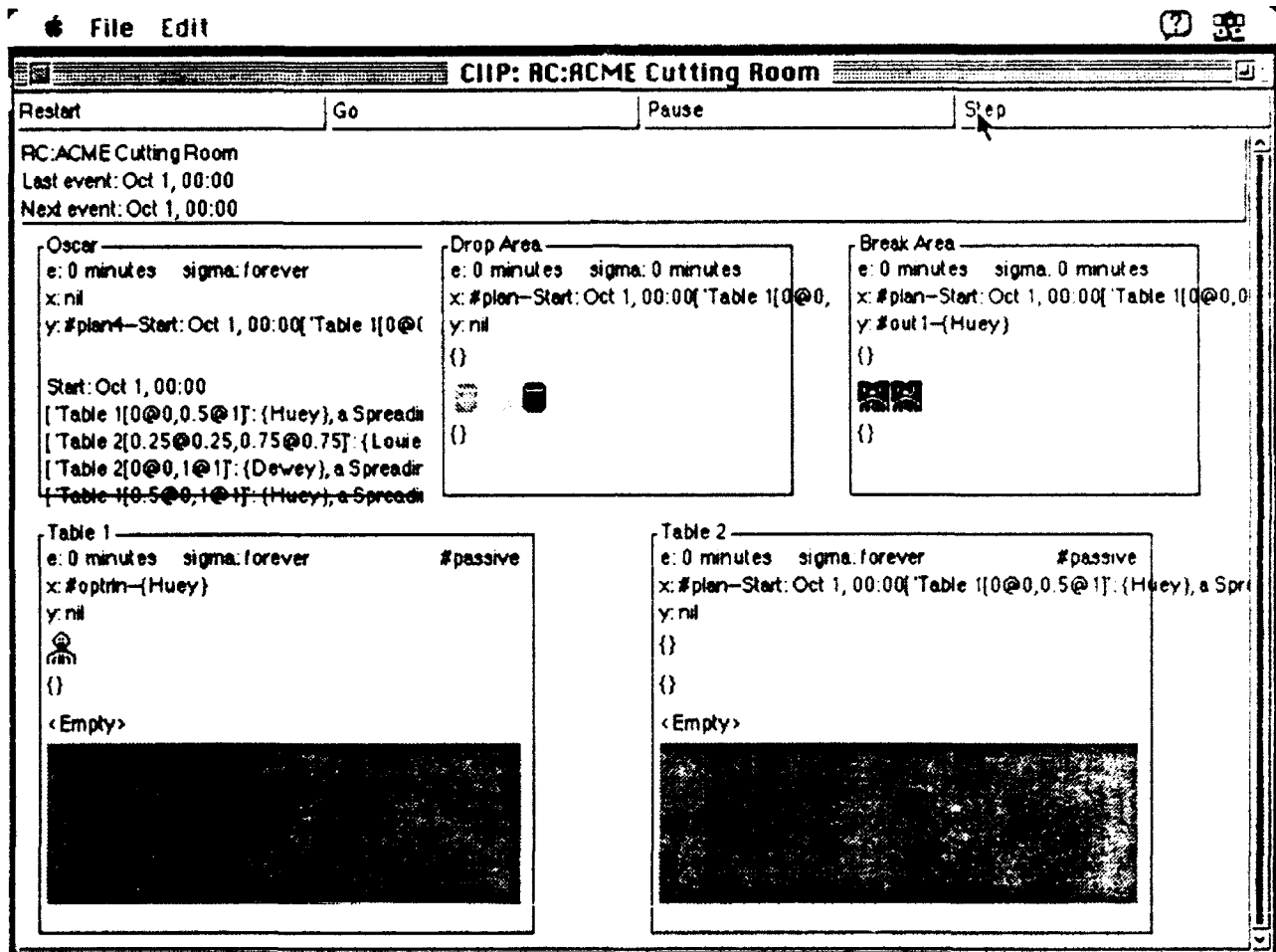


Figure 2.4: The screen for a simulation run after two steps.

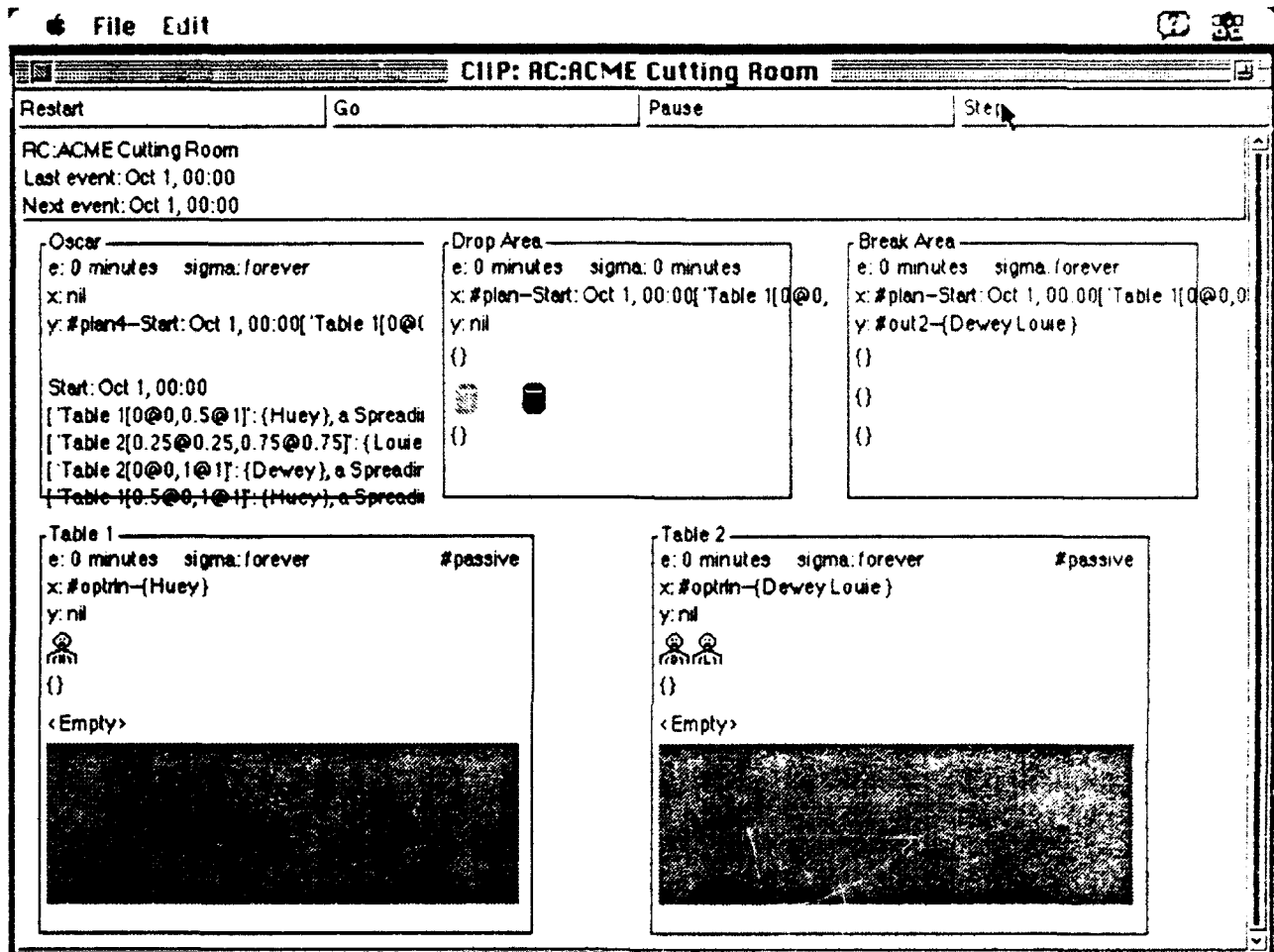


Figure 2.5: The screen for a simulation run after three steps.

assignment occurs. The icon shows him as being idle, presumably waiting for materials to arrive since the equipment itself is not in use. [Note that no time has actually passed as reflected in the root coordinator view.] The drop area is ready to dispatch the three rolls of fabric at the end of this step.

Figure 2.5 shows the window after the Step button has been pressed again. The operators have all been dispatched to the equipment at which they can next start working according to the plan.

Figure 2.6 shows the window after the Step button has been pressed four times. At this point, Huey can begin work on the next work assignment since the necessary materials have arrived, having been dispatched from the drop

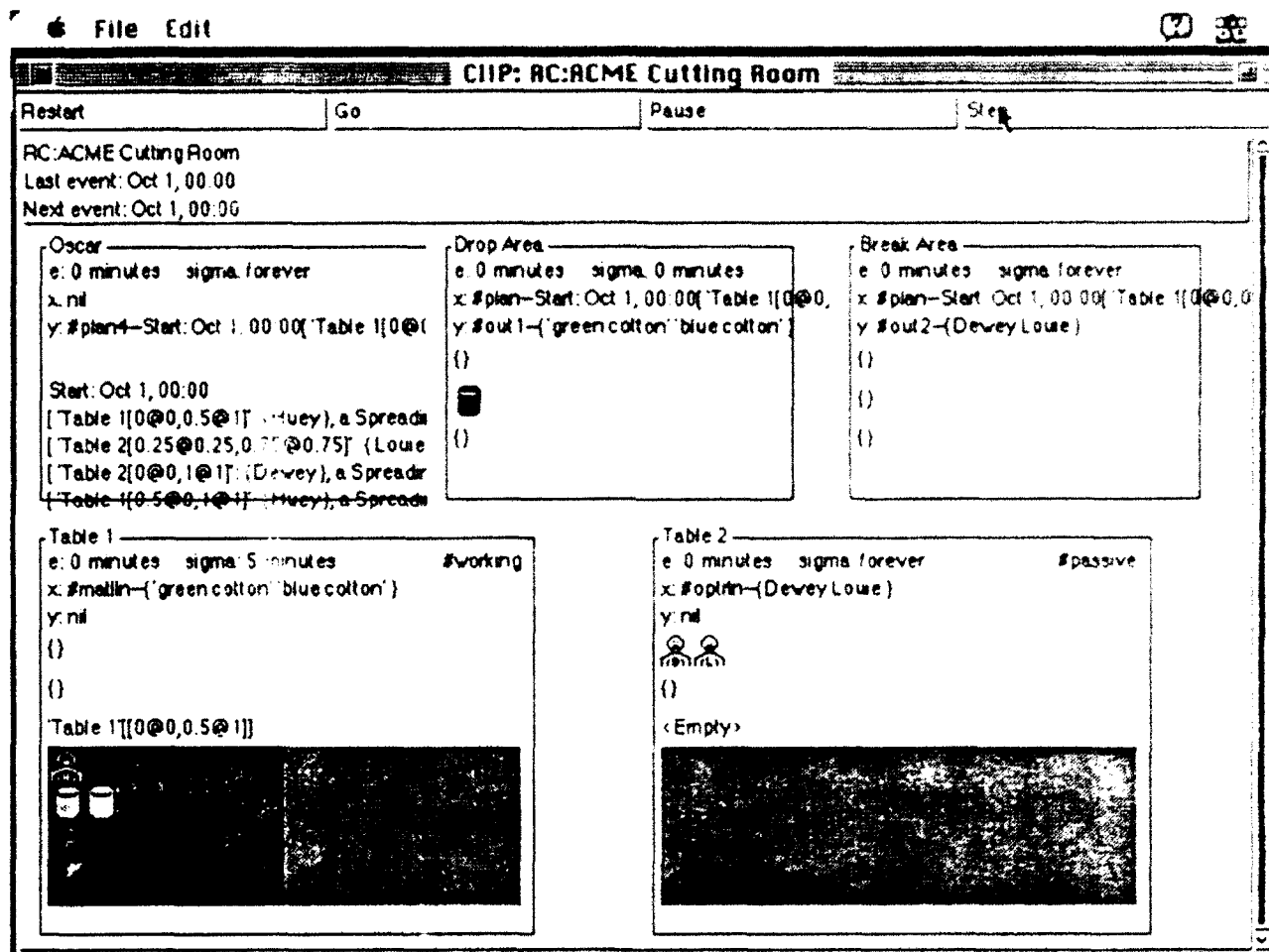


Figure 2.6: The screen for a simulation run after four steps.

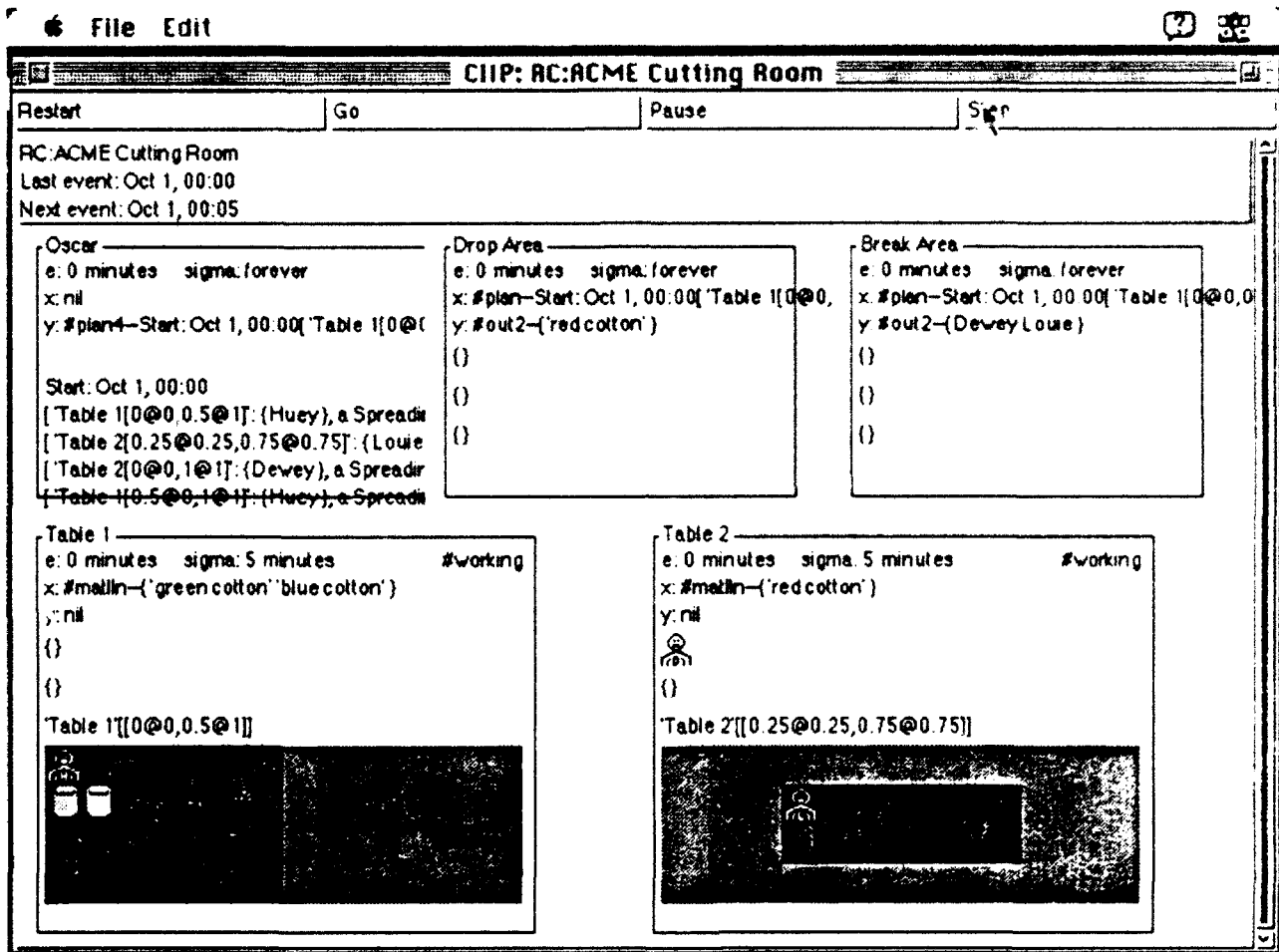


Figure 2.7: The screen for a simulation run after five steps.

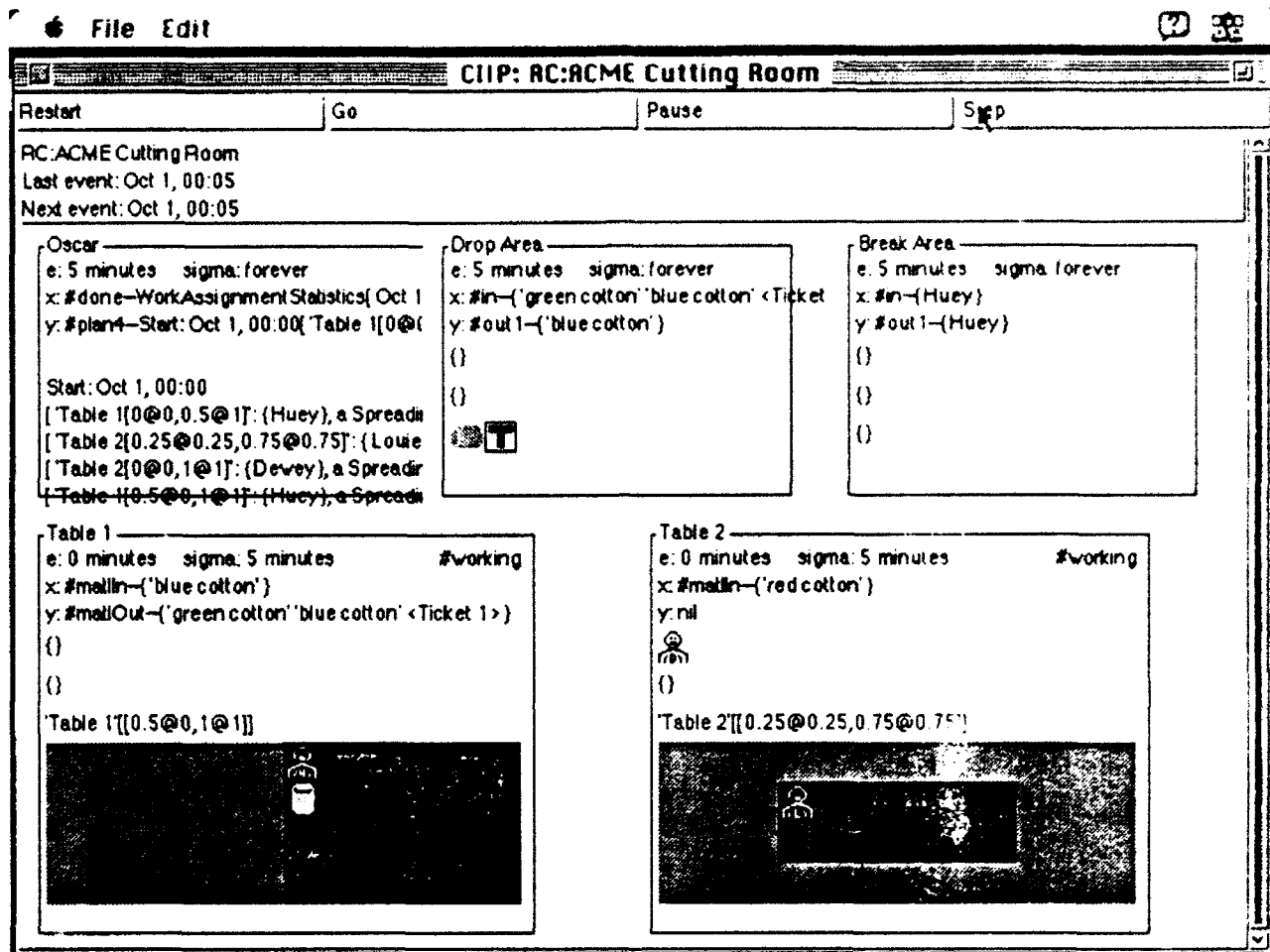


Figure 2.8: The screen for a simulation run after eight steps.

area. Similarly, Louie can begin working at Table 2 with the arrival of materials (see Figure 2.7).

Figure 2.8 shows the window after the simulation run has been stepped until the third task has started. Notice how Huey is now working at a different workstation at Table 1. Five minutes of simulated time have passed, the time needed for Huey to complete his first work assignment. Louie has completed his work assignment, but the simulation has not yet finished processing all state changes at the current simulation time. Before time advances, all state changes for Table 2 will have been performed.

Figure 2.9 shows the completed simulation. The tickets have been delivered

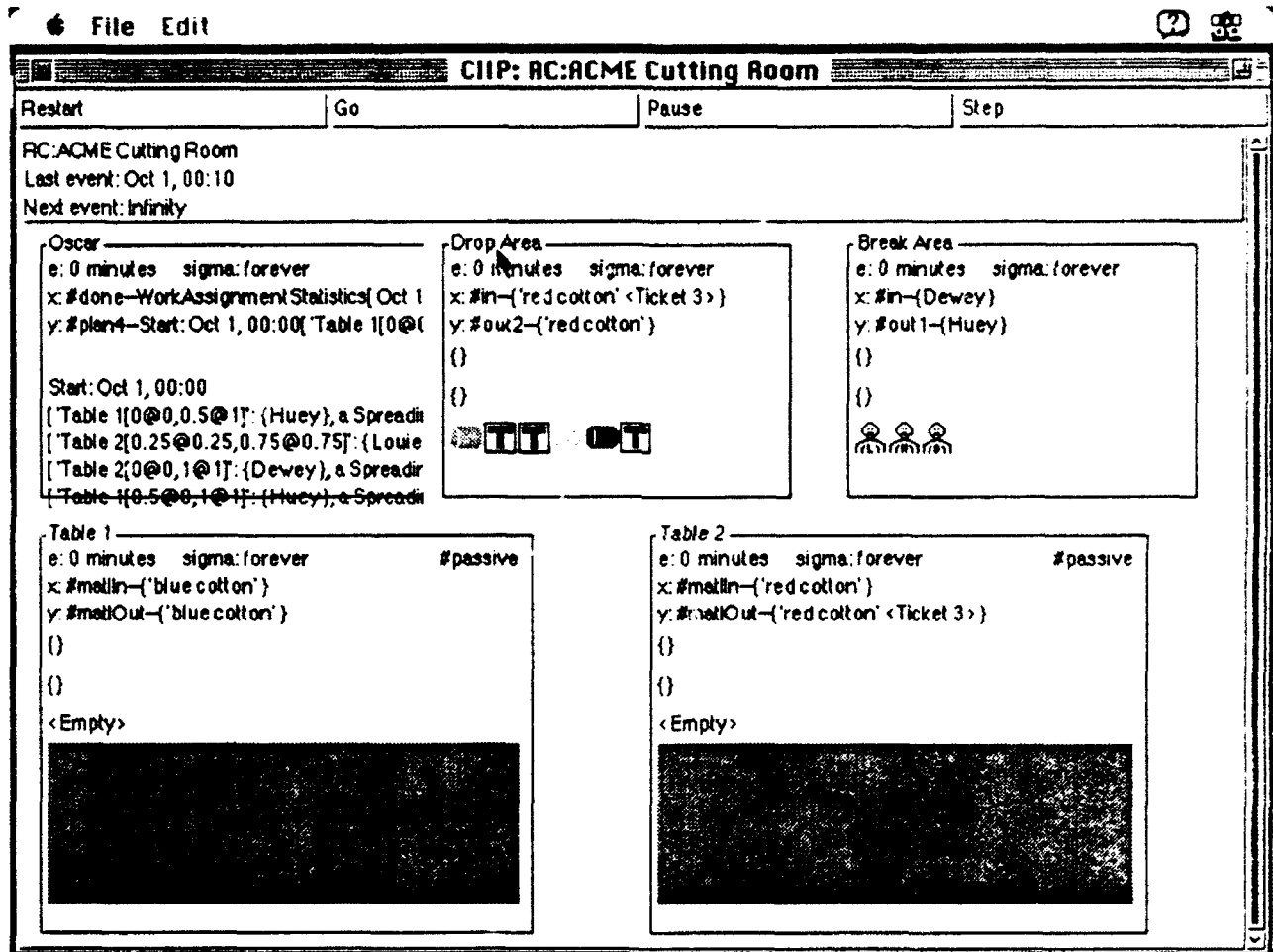


Figure 2.9: The screen for the end of a simulation run.

the system with a minimum amount of programming. Adding a new type of equipment or other resource can be effected by using inheritance. Adding new objects amounts to creating new instances of existing or custom classes.

For example, consider adding a new kind of task—say, one to both spread and cut. The basis for this new task is the class `Task` which encapsulates the knowledge of what it means to be “a task.” The new class, `SpreadAndCut`, is defined as a subclass of `Task` in a system or class browser (refer to the documentation for Smalltalk for details). The only programming required is the definition of the method that computes the duration of time required to complete the task given operator(s), materials, and a workstation.

All parts of the system can be customized using inheritance, including views and the default graphics used for various resources.

Appendix A

Installation

Installation of *CIIP* requires the availability of Objectworks\Smalltalk, Version 4.1 or higher. A disk containing the necessary files accompanies this manual.

To import the *CIIP* class definitions and support files into the Smalltalk environment, follow these steps:

1. If necessary, transfer the files on the installation diskette to a disk volume accessible from within the Smalltalk environment.
2. Start execution of the Smalltalk image on the host system.
3. Open a file browser by selecting *file list* from the *Utilities* menu in the Launcher window. Deselect the "Auto read" feature, type "*.ST" in the first field in the file list window, then hit the return key. A list of the files to be imported appears in the file list window. Highlight each file in turn and select *file in* from the menu obtained by holding down the (first) mouse button when the cursor is positioned over the bar at the top of the list of files. File-in messages will appear in the transcript window. Files may be processed in any order, but the messages in the transcript window will depend on the order used.
4. Once all files have been imported, open a system browser by selecting *system browser* from the *Browsers* menu in the Launcher window. The *CIIP* classes will appear in categories whose names are prefixed with "CIIP" or "DEVS".

Correct installation can be verified by running the test in the file *simple.tst*. Open a file editor on *simple.tst* by selecting *file editor* from the *Utilities* menu in the Launcher window. Type "simple.tst" into the dialog box prompting for the file name. Double click at either the start or end of the text in the window to highlight the text and then select "do it" from the menu obtained by holding down the (first) mouse button while the cursor is positioned over the bar at the

top of the text window. A simulation run window will appear. Press the Restart button with the mouse, and then the Go button. The results of the test run are written to the system transcript window. Close the simulation window when execution completes.

C ℓ IP System Architecture

John D. McGregor, Principal Investigator
and
David A. Sykes
Department of Computer Science
Clemson University
Clemson, SC 29634-1906

December 1992

Contents

1	Introduction	1
1.1	Purpose	1
1.2	Scope	3
1.3	Overview of the Document	3
2	Background	5
2.1	Object-Oriented Programming	5
2.1.1	Objects	5
2.1.2	Classes	6
2.1.3	Inheritance	9
2.1.4	Polymorphism and Dynamic Binding	11
2.1.5	Benefits	12
2.2	DEVS	14
2.2.1	Models	15
2.2.2	Processors	19
2.2.3	Simulation	22
3	System Design	35
3.1	DEVS Implementation in Smalltalk-80	36
3.1.1	DEVS Entities	37
3.1.2	Extensions	44
3.1.3	Support Classes	46
3.2	Application Classes	48
3.2.1	Plans and Tasks	49
3.2.2	Cutting Rooms	53
3.2.3	Equipment and Workstations	55
3.2.4	Resources	57
3.3	User Interface	58
3.3.1	Simulation Run Views	59
3.3.2	Models Views	60
3.3.3	Processors Views	61
3.3.4	Synchronization of Views and Simulation Runs	61

3.4	Debugging Support	62
3.4.1	Logging Messages	62
3.4.2	Debugging Views	64
3.5	Testing Support	64
A	DEVS Class Specifications	66
B	Application Class Specifications	90
C	Test Class Specifications	134

List of Figures

2.1	Polygon objects.	7
2.2	A class definition for quadrilaterals.	8
2.3	An example of the use of inheritance.	10
2.4	An inheritance hierarchy for some polygon classes.	11
2.5	A DEVS model for a simple processor.	16
2.6	A simple coupled model for a queuing processor.	18
2.7	Processor-model pairings for a queuing processor.	19
2.8	A simple hierarchical model of a job generator, transducer, and simple processor.	22
2.9	Pseudo-code for a job generator.	23
2.10	Pseudo-code for a transducer.	24
2.11	Composition tree and influence digraph for an experimental frame	25
2.12	Composition tree and influence digraph for model/frame pair. . .	25
3.1	The DEVS class hierarchy.	37
3.2	CZIP Application classes.	50
3.3	Cutting room model components and their connections.	54
3.4	An equipment model.	56
3.5	States of an equipment model.	56
3.6	User interface classes.	59

Chapter 1

Introduction

This document describes the results of our research into the application of object-oriented technologies to the development of a simulation of cutting room operations in an apparel plant. We gratefully acknowledge funding of this research by the Defense Logistics Agency through Clemson Apparel Research, Contract Number DLA900-87-D-0017, D.O. 0019. We also would like to thank Jantzen in Seneca, SC for their interest in the project that helped us get support from CAR, and Sundaresan Jayaramen at Georgia Institute of Technology for help in understanding cutting room operations and how they fit into the larger manufacturing process.

1.1 Purpose

The purpose of this research has been to investigate whether object-oriented technologies can be useful in improving the efficiency of cutting room operations. The cutting room is responsible for construction of the bundles of materials that are assembled into garments in the sewing room. The decision as to the order in which various cutting orders are processed and the way in which cutting room resources—operators, tables, cutting machines, and so on—are allocated to the cutting orders significantly affects the efficiency with which the cutting room operates. When this project started, a typical apparel plant would be cutting orders today that were not scheduled for sewing for another two or three weeks from today. Such long lead times tend to reduce the impact of problems or inefficiencies in the cutting room on other phases of production. In today's competitive environment, practices such as these are too expensive. Ideally, fabrics would arrive in the warehouse, almost immediately be dispatched to the cutting room for processing, and the resulting bundles delivered to the sewing room just when they are scheduled to be sewn.

With this "just in time" approach to manufacturing, the efficient operation

of the cutting room is more critical than it was previously. The impact of problems in the cutting room is even more significant as companies such as Jantzen move toward centralized cutting facilities—that is, cutting rooms that cut for two or more sewing rooms, some of which might be quite remote. For example, the cutting room at Jantzen in Seneca, SC cuts for the sewing room at the same plant as well as plants in the Caribbean.

Cutting room management is currently an art. Cutting orders must be assigned to the various machines and operators in such a way that costs are held to a minimum while bundles are delivered on time. For example, a schedule might call for cutting black wool slacks, white cotton shirts, and orange sweatpants on a given day. If the cutting room contains two tables with automatic spreaders and two operators are available, then the manager must decide which orders will be cut at each table and which operator will be assigned to each cut. If, for example, only one of the spreading machines can handle the heavy rolls of wool, then some of the decisions don't have to be made. Still, how the remaining resources are assigned to the work significantly affects production efficiency. The problem is compounded by the occasional need to process rush orders for cutting pieces for sample garments.

Many cutting room managers just have a "knack" for assigning resources well. He knows the operators, he knows the equipment, and he knows from experience about how long each cut will take. Unfortunately, if a cutting room manager leaves a plant, the problems of the cutting room can have a devastating effect on production in a whole plant. In fact, this research was first suggested to us as a result of a cutting room manager's leaving and throwing a whole plant into disarray until he had been replaced with someone who could get the cutting room running smoothly again.

Our approach in addressing this problem was to build a software tool to support a cutting room manager in making decisions with respect to the allocation of cutting room resources to cut orders. Our idea was to build a software system to simulate cutting room operations for a given set of cutting orders and work assignments based on those orders. A cutting room manager constructs a plan containing work assignments for the operators. A work assignment comprises a task and the equipment and materials to be used in performing that task. Given such a plan, the system simulates the operation of the cutting room, determining when each assignment is completed and when the plan is completed. The plan can be revised and new simulations run in order to determine the effect of these changes on the completion times.

We proposed to base the system on object technologies for two reasons.

1. We felt that the system would be most useful to a cutting room manager if it had a graphical interface such that an animation of the simulation could be displayed. Animation carries more impact than charts and tables. Our goal was to have a manager be able to watch the effects of a plan on the cutting room as operators move from workstation to workstation, as rolls

of fabric are processed into spreads, as spreads are cut into stacks, and as stacks are packaged into bundles.

Most modern graphical user interfaces (GUI) are based on object technology. Windows, widgets, icons, and geometric shapes are represented by objects.

2. Every cutting room is different. The number and capabilities of the operators, the number and types of equipment, and the layout of the equipment varies from cutting room to cutting room. This variety would generally mean that our simulation system must be customized for each cutting room. We saw objects as a good way of making the customization as easy as possible. Since objects encapsulate the behavior of the real-world entities they represent in software, then any object whose behavior corresponded to what the simulation expected could be "plugged into" the system. For example, the addition of another spreading table could be effected by creating an object to represent it, putting that spreading table object into the cutting room object, and just using it in assignments within a plan. The addition of a new type of cutting machine entails describing the behavior of the machine, requiring some programming. However, inheritance from the general class of cutting machines or even from a specific type of cutting machine that is "almost like" the new type can be used to define most of the behavior of the machine. Programming is needed only to describe any differences.¹

1.2 Scope

This document describes the design and implementation of *CLIP*, including information about DEVS, the simulation methodology used. The information in this document is complemented by information in the *CLIP User's Manual* and in the actual code for the system. Smalltalk-80 classes are not documented here, except in cases where changes were made to predefined library classes.

1.3 Overview of the Document

The remainder of this document is structured as follows:

- Chapter 2 provides information about object-oriented programming and DEVS, the methodology that we chose to use in building our simulation.
- Chapter 3 presents the software architecture of our system. The description comprises two main components:

¹We could hope for a day when manufacturers of cutting room equipment provide class descriptions of each of the products that they sell and those descriptions could be used directly by the simulation.

1. The implementation of DEVS

2. The implementation of cutting room components

- Appendix A provides class definitions for the DEVS component.
- Appendix B provides class definitions for the cutting room components.

A separate document, *CHIP User's Manual*, details the user interface and how to start up and run the system.

Chapter 2

Background

We present a brief description of object-oriented programming and of the DEVS methodology on which the discrete event simulation aspect of our design is based.

2.1 Object-Oriented Programming

In this section we will provide a brief description of object-oriented programming. Software professionals do not agree as to exactly what constitutes object-oriented software, but we adopt Wegner's[Weg87] definition, which requires that a language support these three concepts to be considered object-oriented:

- Objects
- Classes
- Inheritance

We will consider a software system to be *object-oriented* if it is designed and implemented using these three concepts. An object-oriented program is a software system whose components are objects. Computation is performed via the creation of new objects and the communications between them.

2.1.1 Objects

An *object* is the basic component of the object-oriented paradigm. Each object is characterized by its own set of attributes and by a set of operations that it can perform. The values of attributes may change as a result of the application of the operations and, in general, only through the operations. Operations are referred to as *methods* (or *member functions* in C++) and are applied via the process of *message passing* (or *messaging*). A message sent to an object specifies a

method name and a (possibly empty) list of arguments, each of which designates some object. A message received by an object causes code associated with the method named in the message to be executed with its formal parameters bound to corresponding values in the argument list. The processing of a message by the receiving object might result in a state change—that is, a change to one or more of the receiving object's attributes—and/or the sending of a message to itself or some other object. It is useful to think of message passing as being roughly equivalent to function calls in the procedural paradigm. However, the purpose of the method invoked as the result of a message is intended to modify the internal state of the object to which it is attached rather than to modify its arguments and return them. An object may even send itself a message. Some languages provide special terminology to allow the object to refer to itself—for example, *self* in Smalltalk—or let the object be the default if another object is not explicitly referenced.

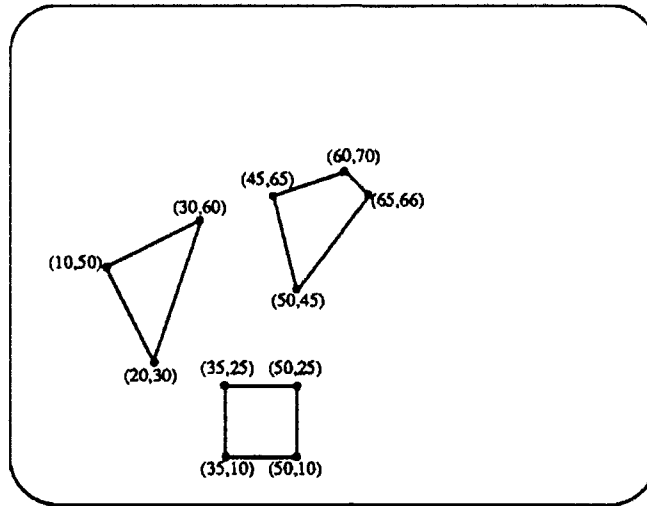
As an illustration, consider polygons drawn on a computer screen. Each polygon is an object defined by an ordered collection of vertices. The order specifies how the points are connected. The vertices define the state of a polygon object, both its shape and its location on the screen. Operations on a polygon include “draw” (display it on the screen), “move” (erase it from its current location and redraw it at some specified distance in the x and y directions), and “contains?” (a check whether some specified point is inside the polygon). Figure 2.1(a) shows three polygon objects on a computer screen and the points that define them. Figure 2.1(b) shows how these polygons are represented as objects.

Note that in describing polygon objects we have used other objects—namely, a screen and points. A screen is a physical object that for our needs here comprises an arrangement of picture elements (or *pixels*) that we can manipulate to draw shapes. A screen object provides methods to turn pixels on and off, to access the current state of a specified pixel (on or off), and to draw lines between any two points, where the screen maps points to pixels. A point represents a specific pixel according to some x and y coordinate system. A point object provides methods to access its x and y components and perhaps to compute its distance from another point.

There is no reason why objects must only represent physical objects. They may be instances of any sort of conceptual entity. Processes in an operating system, the level of illumination in a room, and the role of being a lawyer in a particular trial are all examples of objects.

2.1.2 Classes

A *class* is a set of objects that share a common conceptual basis. The definition of a class includes a set of data attributes plus the set of allowable operations on that data. A class definition can be viewed as a template that is used for the production of objects. All objects in a given class have matching attributes



(a) Three polygons on a computer screen.

triangle	quadrilateral ₁	quadrilateral ₂
(10,50) (30,60) (20,30)	(35,10) (50,10) (35,25) (50,25)	(45,65) (50,45) (65,66) (60,70)
draw move ($\Delta x, \Delta y$) contains?(aPoint)	draw move ($\Delta x, \Delta y$) contains?(aPoint)	draw move ($\Delta x, \Delta y$) contains?(aPoint)

(b) Three objects representing polygons.

Figure 2.1: Polygon objects.

Quadrilateral	
point ₁	point ₃
point ₂	point ₄
draw	
move(Δx , Δy)	
contains?(aPoint)	

Figure 2.2: A class definition for quadrilaterals.

and operations. Each object is an *instance* of some class, and the state of an object is contained in its *instance variables*.

For example, the two quadrilaterals in Figure 2.1—and every quadrilateral object—have the same properties, so we can define a class `Quadrilateral` shown in Figure 2.2 to specify these properties. Every object of class `Quadrilateral` has the same set of instance variables and methods defined by the class. In this sense, the class `Quadrilateral` provides a template for our representation of all four-sided polygon objects, specifying both the variables in each *instance* of a `Quadrilateral` and the set of methods that can be sent to any instance.

An instance creation operator must be provided in order to produce objects from a class definition. A number of approaches have been taken by object-oriented programming languages. C^{++} , for example, defines an explicit *new* operation that creates a new instance of a specified class, incorporating a *constructor* concept by which instances are initialized implicitly when an object of the class is created. Smalltalk uses class operators to create instances. The method *new* is inherited from the `Object` class and serves as the basis for creating instances of all other classes.

Programming languages take different approaches to instance destruction—that is, the deletion of objects when they are no longer useful so memory can be made available for other objects. C^{++} provides a *delete* operator that can explicitly free up the space used by an object, thereby relying on the programmer to manage objects in memory. C^{++} also allows each class to define a destructor method that is called implicitly when an object is destroyed. Smalltalk, on the other hand, does not provide a mechanism to destroy objects, but relies instead on garbage collection.

Most languages that support the object-oriented paradigm provide data abstraction mechanisms. The mechanism for class definition provides a means for designating the operations that users of the class will be able to access. This set of operations is termed the *class interface*. The remainder of the class definition provides data definitions and auxiliary function definitions that comprise the *class implementation*. This separation isolates the users of the class from

the effects of changes to the internals of a class.

The class interface is the set of operations that instances of the class can be requested to perform. The simplified public interface for the *Quadrilateral* class, Figure 2.2, shows the messages to which instances of the *Quadrilateral* class can respond. Sending the message *draw* to an instance of class *Quadrilateral* results in that instance executing its *draw* operator. The *draw* operator would be designed to draw a quadrilateral having shape and location determined by the point data inside the instance.

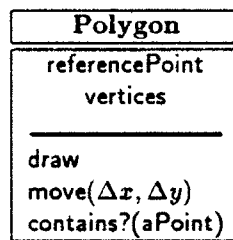
The class implementation often includes instances of other classes that provide services the new class requires. In the quadrilateral example, the four points that specify a quadrilateral are point objects defined within the quadrilateral object. These point objects are intended to be inaccessible from other objects. If we decide that it is important for other objects to be able to access these points, then we can add methods to the class interface to provide this access. For example, we might want to designate one point—perhaps the point closest to point (0,0) on the screen—as a reference point. We would define a method, say *referencePoint*, to provide that value rather than allowing other objects to compute it from the values of *point₁*, *point₂*, *point₃*, and *point₄*. A class implementation might also include some private methods—for use in implementing the class, but not intended for use by any other object. For example, we might want a private method that, when given a list of points, determines the point that is closest to (0,0).

A class is similar to—but also very different from—a record in Pascal or a structure in C in the sense that it is an aggregation of data values. Classes normally extend the usual semantics of records to provide varying levels of visibility—that is, some components of the record may not be accessible by every component that has visibility to the record type. Classes differ from records in that they include definitions of operators with the same status as the data values declared within the class. These are not equivalent to function pointers that are defined independently of the class and stored in the class instances.

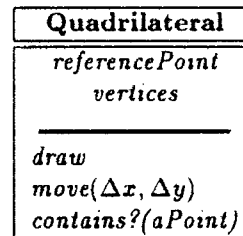
Virtually all object-oriented languages use objects as the representation of the instances that make up an application. Some languages, such as Smalltalk, also implement the classes themselves as objects. Since objects are the manipulatable entities in an object-oriented application, the implication is that languages that implement classes as objects allow the manipulation of classes by an application. This capability supports the flexibility needed in many application areas such as artificial intelligence.

2.1.3 Inheritance

Inheritance is a technique for using existing definitions as the basis for new definitions. The definition of the new class is a combination of the data and operation declarations from the existing class(es) and any declarations added by the new class. The new class reuses the existing definitions without any need



(a) a class Polygon.



(b) a subclass Quadrilateral
of class Polygon.

Figure 2.3: An example of the use of inheritance.

to modify the existing classes. It is less expensive to develop this way because a portion of the class has already been implemented and tested. The existing class is referred to as the *parent class*, the *base class*, or the *superclass*. The new class is correspondingly referred to as the *child class*, the *derived class*, or the *subclass*.

Consider the Quadrilateral class. If the class Polygon shown in Figure 2.3(a) existed when class Quadrilateral was defined, then the definition of Quadrilateral could have looked like Figure 2.3(b). The italicized items in Figure 2.3(b) have been defined in class Polygon and added to the definition of class Quadrilateral through the inheritance mechanism. Presumably these elements have already been tested as part of class Polygon and may not need to be retested as rigorously as newly written code.

Defining a new class using inheritance can be viewed as describing a new set of objects that is a subset of the objects described by the existing class. This new subset can be thought of as a *specialization* of the existing class. For example, the Quadrilateral class in Figure 2.3 is a specialization of the Polygon. A quadrilateral is a polygon restricted to four sides. We could further specialize a quadrilateral into a rectangle which is a quadrilateral having special properties. The interface for the Quadrilateral class might be identical to that of the Polygon class, and the interface for a Rectangle might be the same as that of the Quadrilateral.

The new class can also be viewed as having an interface that is an *expansion* of the interface of the existing class. For example, deriving a four-wheel drive vehicle class from an existing vehicle class would not only specialize the definition to a subset of vehicles, but probably also introduce new capabilities in the new class interface. Continuing with our example, we might wish to add more operations for a rectangle—for example, a method that would answer the largest ellipse that can be inscribed in that instance.

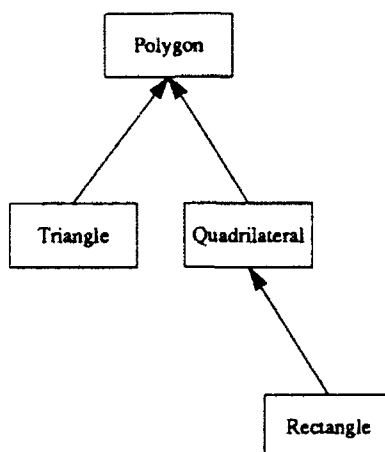


Figure 2.4: An inheritance hierarchy for some polygon classes.

How much of the existing definition is available to be added to the new definition varies from one language to another. Many languages give the implementor control over which attributes are actually inherited and how they are inherited. Typically, the system developer may choose to either inherit only the interface and not the representation or to inherit both the interface and the representation. In our approach to object-oriented design, we inherit *all* the attributes of the existing class in the new class definition, despite what facilities our implementation language might provide to hide inherited attributes. Our approach leads to the development of inheritance structures that are conceptually rational and understandable.

2.1.4 Polymorphism and Dynamic Binding

Objects, classes, and inheritance characterize the object-oriented paradigm, but other techniques are used in conjunction with them to provide additional power. Two of these are *polymorphism* and *dynamic binding*.

A language supports polymorphism if the same name or symbol can be used in different contexts. In an object-oriented programming language, more than one class may use the same method names, and hence the various instances of these classes can respond to the same messages, though each in its own way. The determination of which method to dispatch when a message is received by an object is determined through dynamic binding—that is, the method dispatched is determined by the class of the receiving object at run time. Consider, for example, the class hierarchy shown in Figure 2.4. By polymorphism, any instance of these classes can reply to a message `contains?(p)`, but the way in which the answer is computed depends on the kind of polygon. If the message is sent to

P, then the method dispatched to compute the answer depends on whether P is a triangle, a quadrilateral, or a rectangle. In this way, the action taken depends on the class of P at run time.

2.1.5 Benefits

We briefly examine some of the reasons for using object-oriented design and implementation techniques: Object-oriented programming

- *Promotes reusability.* Object-oriented techniques yield structures that are more readily reused than other design techniques. Reuse comes in many forms:
 1. There is reuse by using an instance. For example, an application might use many instances of class `Quadrilateral`.
 2. There is reuse by using an instance in a definition. For example, the `Quadrilateral` uses instances of class `Point`.
 3. There is reuse by evolution. For example, class `Quadrilateral` is used to define class `Rectangle`.

The support for data abstraction in object-oriented methods promotes these types of reuse. Designers view object-oriented techniques from different perspectives. Those currently using languages such as C will see the additional potential for reuse provided by classes. Those using languages such as Ada or Modula-2 will see packages and modules as providing the first two types of reuse. What they will not have seen is the third type of reuse.

- *Facilitates maintenance.* The information-hiding supported by most object-oriented programming languages facilitates maintenance. The interface of a class defines the set of operations on the data of an instance of that class. If a change is made to the representation of data defined in the class, then those operations defined in the class that interact with the changed data need to be modified. There is no need for users to modify their references to instances of the class unless the signature of one of the operations has been changed. The impact of this and many other maintenance activities is localized.

For example, consider an implementation of the class `Rectangle` in which we wish to use two points to define it instead of four points. [Note that if we require a rectangle to be aligned with the x and y axes, then we can define it in terms of its upper-left vertex and its lower-right vertex. Many microcomputer systems provide facilities to draw rectangles in terms of these two points.] The methods `draw`, `move`, `contains?`, and `referencePoint` would have to be reimplemented to determine the other two points if they

are needed. These changes would be localized to the code of the methods of the Rectangle class.

- *Exploits commonality.* Object-oriented techniques exploit commonality in two ways. First, there is commonality across applications. Software development firms tend to develop applications that address a common domain such as communications or graphics. By developing units that are easily reused, object-oriented design exploits the commonality within a company's applications. Second, there is commonality across system components. For example, a graphics system defines numerous shapes such as lines, triangles, circles, and rectangles. These components are different shapes, but they have many common attributes. Object-oriented design develops a structure that factors out these common elements into a class, which can then be the basis for defining each of the individual shape classes.
- *Reduces complexity.* This is where the advertisement starts because we have little more than anecdotal evidence to support this claim. A recent paper by Rosson and Gold[RG89] does provide some evidence in support of the claim. Existing procedural techniques require that the designer have a solution in mind before beginning the design process. This requires the designer to be an expert problem solver because the design technique only supports the computerization of a solution rather than the problem-solving process. This further implies that the designer cannot begin the system design until a complete problem solution is known. With today's complex systems, this is often a severe limitation.

Object-oriented techniques begin the development process in the problem domain rather than in the solution domain. This relieves the designer from developing a complete solution before beginning any design work. Thus, the designer is able to handle more complex problems because of the support provided by the design techniques.

We have used these techniques because we wanted an effective solution. We proposed to use object-oriented techniques for the design and implementation of C/TP for a number of reasons:

- The use of objects in design leads to an architecture that is characterized by discrete structures that communicate with each other. The encapsulation of functionality means that otherwise lengthy and complex flows of control are decomposed into shorter, simpler segments hidden within the individual objects. Consider the simulation component of C/TP. The functionality needed to simulate the passage of simulated time, to post the time of next event for a given object, and even to display an object must be provided for each simulation object. In different scenarios, users may have the power to create an almost unlimited number of cutting room

resources to use in a simulation run. Having each instance of a resource contain the functionality that it needs results in an architecture that distributes responsibility. Such a system easily handles multiple occurrences of complex resource entities.

- The use of inheritance to evolve new definitions from existing ones supports the solution of complex problems. Areas in which solutions evolve over time rather than being obvious at the outset require many changes as an application is developed. Developing subclasses from existing classes allows the designer to modify the behavior of classes incrementally as new information about a solution becomes available. These changes can be made without disturbing existing software components, which depend on the classes that are to be modified. We want to be able to add new equipment to a cutting room as well as to be able to add new kinds of equipment.
- The encapsulation of representation characteristic of objects supports the prototyping of systems. Once the module interfaces are developed, the underlying representations of classes can be modified without impacting the modules that use the services of the changing module. Quickly developing the behavior of a simulation object can be followed by detailed implementation that provides specialized responses. Strictly speaking, this is a characteristic of data abstraction that is an integral part of object-orientation. Our goal has been to develop a prototype that can be refined later for a specific installation. We have been less concerned about finding actual values for things such as operator efficiency and spread rates for equipment, and more concerned about identifying how all the objects need to interface in order to produce a meaningful simulation run.

2.2 DEVS

In this section we describe how DEVS works. [For a more formal treatment and further details, see Zeigler's book.]

The DEVS methodology is based on the definition of models and on the interconnection of these models to construct larger models which may, in turn, be connected to other models. This layered construction provides a hierarchy of models. Each model is modular in the sense that it can be used in a variety of simulations.

Two main kinds of entities exist in DEVS-Scheme: models and processors. Models contain information and operations about the domain of interest. Processors carry out the simulation of DEVS models.

In this section we describe models, processors, and simulation in DEVS.

2.2.1 Models

Zeigler describes a DEVS *model* as an object containing the following information [Zei90, pp. 48–49]:

- a set of input ports through which external events are received
- a set of output ports through which external events are sent
- the set of state variables and parameters: two state variables are usually present—*phase* and *sigma* (in the absence of external events the system stays in the current *phase* for the time given by *sigma*)
- the time advance function which controls the timing of internal transitions—when the *sigma* state variable is present, this function just returns the value of *sigma*.
- the internal transition function which specifies to which next state the system will transit after the time given by the time advance function has elapsed
- the external transition function which specifies how the system changes state when an input is received—the effect is to place the system in a new *phase* and *sigma* thus scheduling it for a next internal transition; the next state is computed on the basis of the present state, the input port and value of the external event, and the time that has elapsed in the current state.
- the output function which generates an external output just before an internal transition takes place.

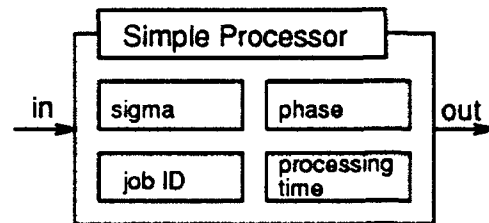
DEVS models comprise two categories: atomic models and coupled model.

Atomic Models

Atomic models are those models at the lowest level of the hierarchy. The behavior of such a model is determined by the input even types it recognizes, its state variables and parameters, its time advance function, its internal transition function, its external transition function, and its output function. We characterize an atomic model by means of a simple example.

Figure 2.5 shows a graphical representation of a simple processor. The processor being modeled is provided with jobs, each requiring a fixed amount of processing time. This processor does not queue up jobs as they arrive. If a job arrives while the processor is busy with a job, then the arriving job is ignored.

The model behaves in accordance with the description provided in the right part of the figure. When the model receives an input x on port *in* and the processor is idle (state *passive*), then the processor starts to work on the job



State variables:

sigma = ∞
 phase = *passive*
 jobID = *None*

Parameters:

processingTime = 5

External transition function:

if phase is *passive* then
 jobID $\leftarrow$ *input job ID*
 hold in phase *busy* for processingTime
 else - phase is *busy*
 continue

Internal transition function:

case phase is
 busy: *passive*
 passive: (Does not arise)

Output function:

send jobID to port out

Figure 2.5: A DEVS model for a simple processor.

whose ID is specified in *x*. The processing time for each job is fixed at five time units (parameter *processingTime*). After processing time has passed, the identification of the completed job is emitted on output port *out* as determined by the output function. If a job arrives on the input port while the processor is already busy (as signified by phase *busy*), the arriving job is ignored as determined by the external transition function's handling of an input when the processor is in the *busy* phase.

An atomic model can have any number of input and output ports. The external input function must specify an action for an arrival on each of the input ports. Usually, the processing associated with an arrival on any given port is determined by the current state of the model—as determined by *phase* and possibly other state variables—and/or by the object arriving on the port.

With respect to the management of state and time, four actions are commonly taken:

1. *hold-in* phase *for* duration. The model stays in the phase indicated for the duration of time indicated. Thus, the next internal transition occurs at the current time plus the duration indicated. However, the arrival of an external input might preempt that next internal transition.
2. *passivate in* phase. The model enters the phase indicated for an infinite amount of time, equivalent to *hold-in* phase *for* ∞ . Thus, no next internal transition is scheduled to occur and the model will react only to external inputs.
3. *passivate*. The model enters a passive phase for an infinite amount of time, equivalent to *passivate in* passive.
4. *continue*. The model continues in the same phase, reducing the amount of time to the next internal transition by the amount of (simulated) time passed in this state so far.

It is important to realize that an atomic model has no conception of a global simulation time. Time relates only to the state transitions that a model must undergo. In the simple processor example, the model is either busy or passive and oblivious to the current simulation time. The only times of significance are durations—the elapsed times at which state transitions occur.

Coupled Models

A coupled model comprises a set of components—each an atomic or coupled model—with specified interconnections. A coupled model contains the following information:

- the set of component models
- the set of input ports through which external events are received

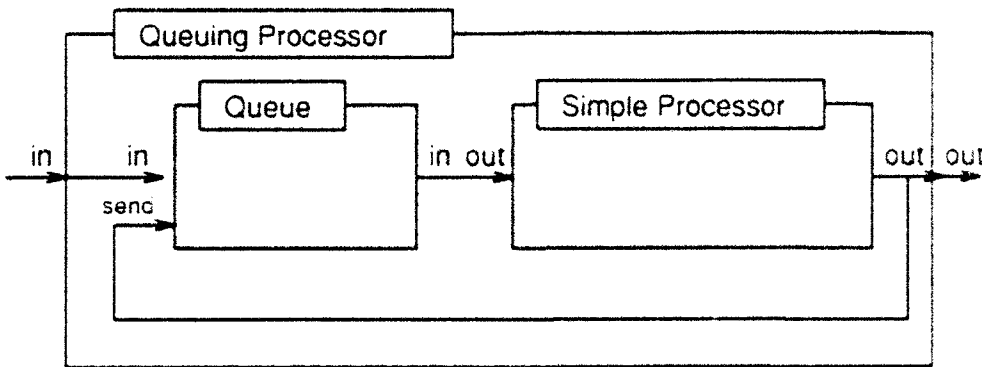


Figure 2.6: A simple coupled model for a queuing processor. The queue is used to hold jobs waiting for the simple processor.

- the set of output ports through which external events are sent
- for each component, its *influencees*—the other components of the model that affect it
- the coupling specification:
 - the external input coupling that specifies how input ports of the coupled model are connected to input ports of the component models
 - the external output coupling that specifies how output ports of the component models are connected to the output ports of the coupled model
 - the internal coupling that specifies how the output ports of components are connected to the input ports of components

Note that an output port of a component model can be connected to any number of the input ports of component models as well as to any number of output ports of the coupled model. Similarly, an input port of the coupled model can be connected to any number of input ports of component models.

- the *select* function that specifies the order in which two or more components that are ready for a state transition at the same time will undergo the transition.

Consider the coupled model shown in figure 2.6. This model is for what we'll call a "queuing processor" because it queues jobs until they can be processed. It comprises two models: one simple processor model and one queue model. A

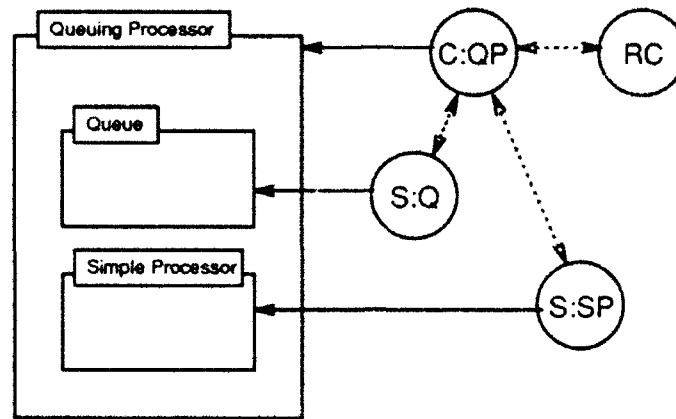


Figure 2.7: Processor-model pairings for a queuing processor.

queuing processor has one input port—*in*— and one output port—*out*. Input jobs are routed to the queue component model which buffers jobs arriving on its *in* port and sends jobs one at a time to the processor in response to an arrival of anything on its *send* port. The output of the processor is routed both to the output of the queuing processor and to the *send* input port of the queue.

Note that the queue model could be either atomic or coupled. The behavior of either type of model is indistinguishable, illustrating the modularity of DEVS.

2.2.2 Processors

Within a simulation run, each DEVS model is controlled by a processor that monitors the passage of simulated time and activates functions in the model as appropriate. There is a one-to-one mapping of processors to models. The model attached to a processor is referred to as the processor's *devsComponent*. Each atomic model is controlled by a *simulator*. Each coupled model is controlled by a *coordinator*. A special processor, a *root coordinator*, manages the overall simulation and is linked to the coordinator of the outermost coupled model. Processors are linked in a tree that reflects model hierarchy. Leaf nodes are simulators while internal nodes are coordinators except for at the root. The configuration for the processor example is shown in figure 2.7. Processor objects are shown as circles, models as rectangles.

Processors send and receive four kinds of messages to effect simulation. A processor takes action on its attached model based on the type and content of the messages it receives. Simulation begins when the root coordinator sends the first message to indicate the start of simulated time. The message is routed by processors through the hierarchy to a model with an appropriate state change. (Models are prioritized so that if more than one has a state change imminent,

then the next one to change is selected deterministically.) A state change can result in outputs that affect other models. Messages are routed by processors as appropriate based on output and input port connections between models. Eventually, once all state changes at the current time have settled out, the root coordinator receives a message indicating that simulation time can advance and a cycle of messages is begun anew. Simulation terminates after some preset duration or when no more transitions are scheduled for any of the models in the simulation.

The four message types used in DEVS are:

- ** —indicates that the next internal event is to be carried out within the scope of the processor receiving the message.
- x* —signifies the arrival of an external input to a processor's model and bears the global model time of that event. The input value and port of arrival are indicated in the message content.
- y* —carries the output of a model in its content and the global model time at which output occurred.
- done* —indicates that a state transition has been carried out and bears the global time of the next event.

A message comprises three main components:

- source** —designating the originator of the message.
- time** —a simulation time stamp or a duration, depending on the use.
- content** —comprising a *port* designation and a *value*, determined by the model output function. This component is not meaningful for *done* and *** messages.

Simulators

Simulators handle atomic models in a simulation. A simulator *S:M* for atomic model *M* tracks the global time at which the last event occurred in *M* and the global time at which the next event will occur, based on the duration of time provided by *M*'s time advance function.

A simulator receives *** and *x* messages and sends *y* and *done* messages in reply. A simulator *S:M* having devComponent *M* acts as follows in response to *x* and *** messages:

- Upon receipt of an *x* message, *S:M* activates the external transition function of *M* and then respond by sending a *done* message to its parent processor. The *done* message indicates that the transition has been performed and carries with it the simulation time of *M*'s next internal transition. This time is computed by adding the value of *M*'s time advance function to the time carried in the *x* message.

- Upon receipt of a ** message*, S:M activates M's internal transition function and responds by sending a *y message* to its parent, followed by a *done message*. The *y message* content contains the output port and value computed by M's output function. The *done message* indicates that the transition has been carried out and indicates the simulation time of M's next internal transition.

Coordinators

Coordinators handle coupled models in a simulation. A coordinator C:M for coupled model M tracks the global time at which the last event occurred in M and the global time at which the next event will occur. A coordinator also maintains a list of the processors for the component models of its *devsComponent*, sorted according to increasing time until the next transition. The processor at the head of this list is the processor's *imminent child*. If two or more models have matching times, then a selection function is used to determine the order. Ordinarily this function consults a prioritized list of the models.

A simulator receives and sends all four kinds of messages:

- Upon receipt of an *x message*, C:M transmits this message to each of M's component models that have input ports connected to the port indicated in the message.
- Upon receipt of a ** message*, C:M transmits a ** message* to its imminent child.
- When a coordinator receives a *y message* from its imminent child, it checks the coupling in the model to see whether the output port in the message is connected to an output port of the coupled model. If so, the message is transmitted to its parent. Next, if the output port indicated in the message is connected to an input port of any other models within the coupled model, then the output value is transmitted to each child in an *x message* directed to the appropriate port.
- After a coordinator receives a *done* for each of the *x messages* it has sent to children and the *y messages* it has sent to parents, then it determines a new imminent child from among its children and sends the time for the imminent child's next event in a *done message* to its parent.

Root Coordinators

A root coordinator controls a simulation run. Unlike the other two kinds of processors, a root coordinator has no model attached, but a coordinator attached

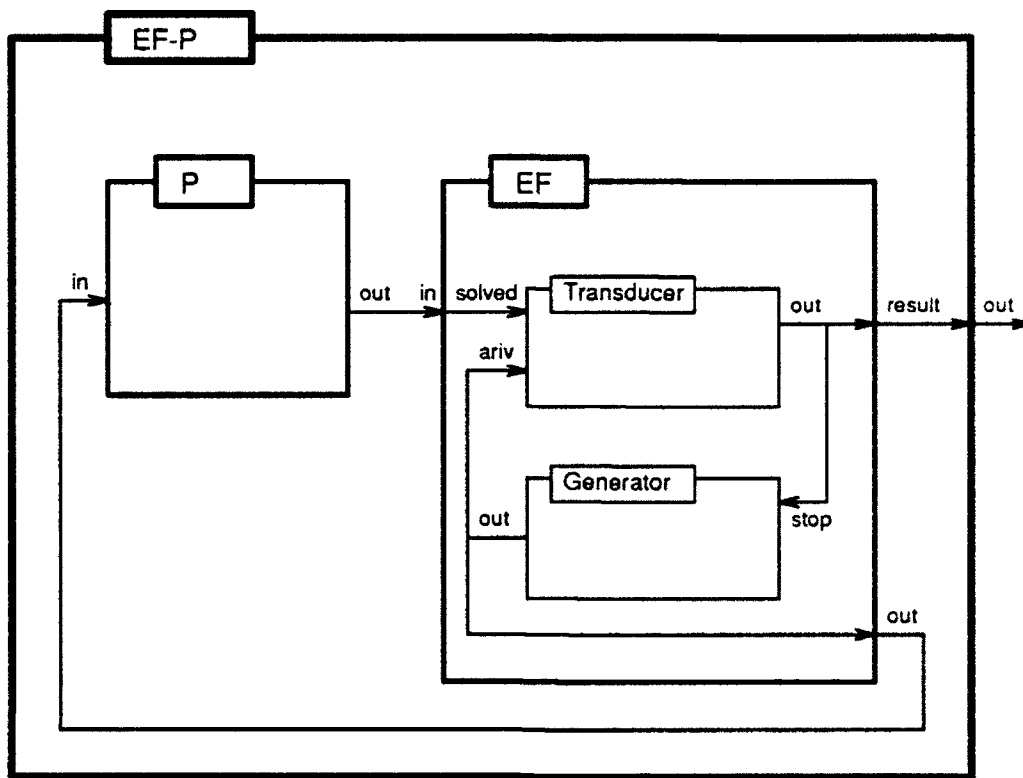


Figure 2.8: A simple hierarchical model of a job generator, transducer, and simple processor.

instead.¹ A root coordinator maintains the current global simulation time and starts the simulation by sending a *** message containing the current simulation time to the attached coordinator. When the coordinator responds with a *done* message indicating the time of the next event, the root coordinator updates the simulation clock and starts the next cycle with another *** message. The coordinator might send a *y* message to a root coordinator. This message contains the output of the model. A root coordinator just ignores such messages by default.

2.2.3 Simulation

We illustrate the operation of DEVS on a simple model taken from [Zei90] and shown in Figure 2.8. At the top level, this model comprises a processor model

¹Zeigler describes only a connection to a coordinator. There seems to be no reason why a simulator cannot be attached, though perhaps atomic models are just not interesting by themselves.

```

ATOMIC MODEL: GENR
state variables: sigma = infinity
                phase = active
parameters: interarrival-time = 10
external transition function:
    case input-port
        stop: passive
        else: error
internal transition function:
    case phase
        active: hold-in active interarrival-time
        passive: (does not arise)
output function:
    case phase
        active: send a random job name to port out
        passive: (does not arise)

```

Figure 2.9: Pseudo-code for a job generator.

and an experimental frame model. The latter is a digraph model comprising two other models, a job generator and a transducer. The job generator creates a job every ten seconds until signaled to stop. The transducer collects statistics about jobs—average turnaround time and throughput—by monitoring the jobs generated and processed. The transducer model includes a clock among its state variables to track global simulation time. Descriptions of the models are given in Figures 2.9 through 2.12. For a more detailed description these models, see Chapter 5 of [Zei90].

The sequence of messages to effect simulation is given below. This sequence is quite lengthy even though a significant part of it has been deleted. We present this level of detail in order to describe the operation of DEVS. This level is not provided in [Zei90], and we worked for a long time to collect the information necessary to determine how messages are generated and processed. While the sequence below is difficult to follow because of its length, it does serve as a reference for determining the correct operation of a simulation.

The levels of indentation in the sequence roughly match the levels in the hierarchy. One can easily see how messages propagate down the levels, back up, down again, and finally back up. Actions shown in *italics* correspond to actions on an atomic model attached to the simulator.²

²This sequence was generated by *C2IP* during a run. We have included a provision for producing LaTeX commands which can be formatted to decipher the operation of the system

ATOMIC MODEL: TRANSD

state variables: sigma = observation-interval

phase = active

arrived-list = ()

solved-list = ()

clock = ()

total-ta = ()

parameters: interarrival-time = 10

external transition function:

advance local clock to agree with global clock;

case input-port

ariv: append job to arrived-list

solved: find job arrival time;

total-ta = clock - arrival-time;

put the job to solved-list

continue

internal transition function:

case phase

active: passive - - *end of observation interval*

output function:

case phase

active: average-turnaround-time =

total-ta / solved-job-number;

thruput = solved-job-number / clock

else: no output

Figure 2.10: Pseudo-code for a transducer.

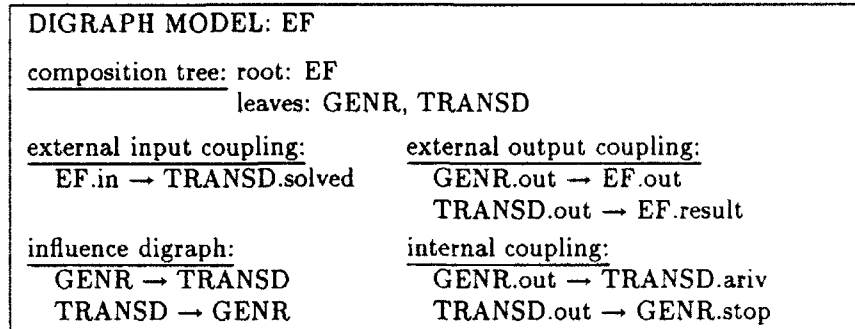


Figure 2.11: Composition tree and influence digraph for an experimental frame.

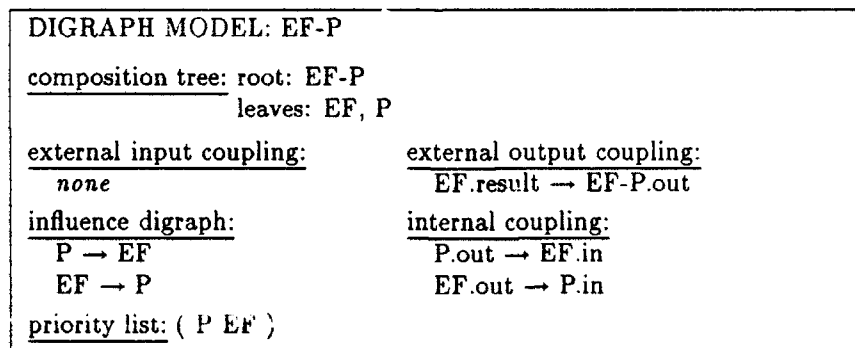


Figure 2.12: Composition tree and influence digraph for model/frame pair.

RC EF-P S:P EF S:GENR S:TRANSD

```

restartAt: Dec 1, 00:00
restarting P at: Dec 1, 00:00
  restartAt: Dec 1, 00:00
  send: Done Infinity
recv: Done Infinity
restarting EF at: Dec 1, 00:00
  restartAt: Dec 1, 00:00
  restarting GENR at: Dec 1, 00:00
    restartAt: Dec 1, 00:00
    send: Done Dec 1, 00:00
  recv: Done Dec 1, 00:00
  restarting TRANSD at: Dec 1, 00:00
    restartAt: Dec 1, 00:00
    send: Done Dec 1, 01:40
  recv: Done Dec 1, 01:40
  send: Done Dec 1, 00:00
recv: Done Dec 1, 00:00
send: Done Dec 1, 00:00

```

Time: Dec 1, 00:00

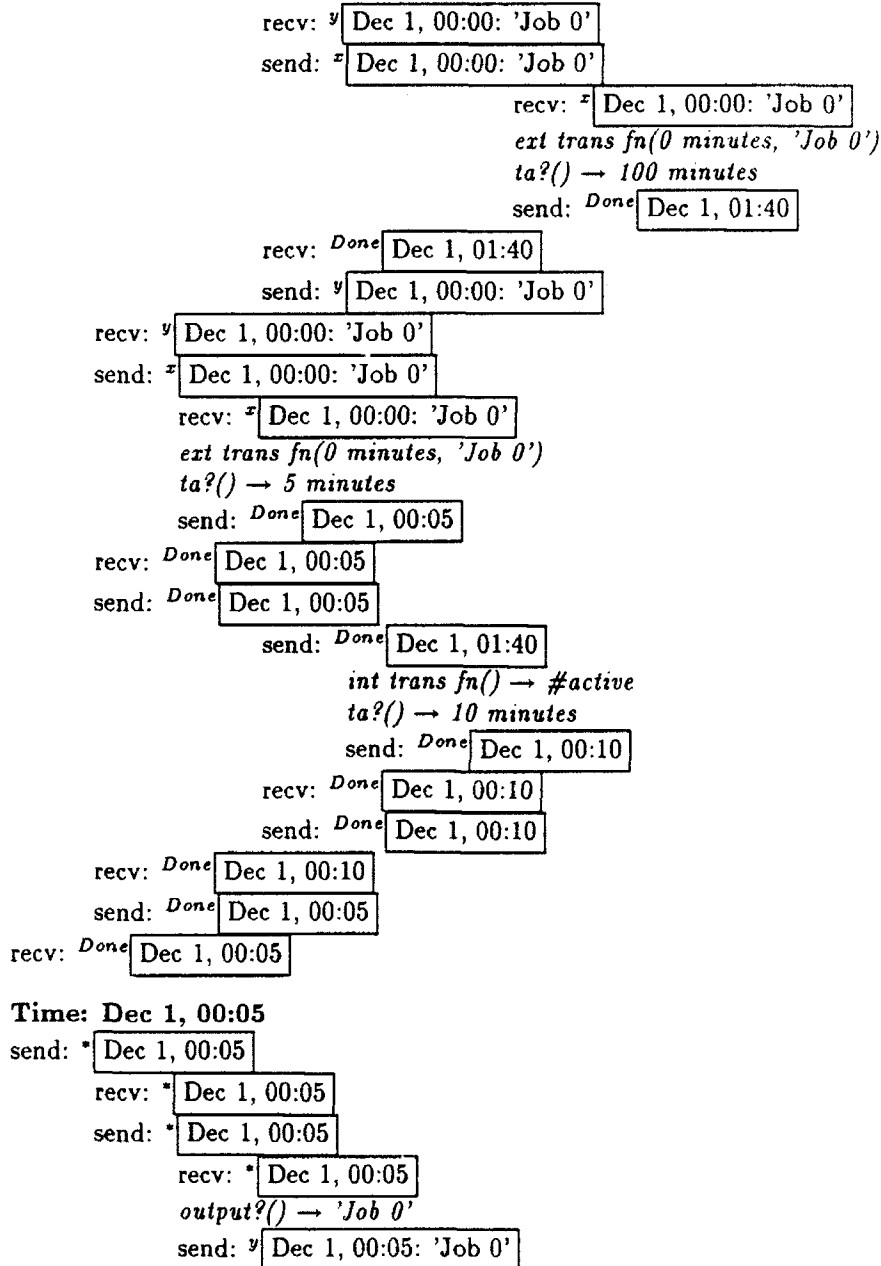
```

send: * Dec 1, 00:00
recv: * Dec 1, 00:00
send: * Dec 1, 00:00
recv: * Dec 1, 00:00
send: * Dec 1, 00:00
recv: * Dec 1, 00:00
output?() → 'Job 0'
send: * Dec 1, 00:00: 'Job 0'

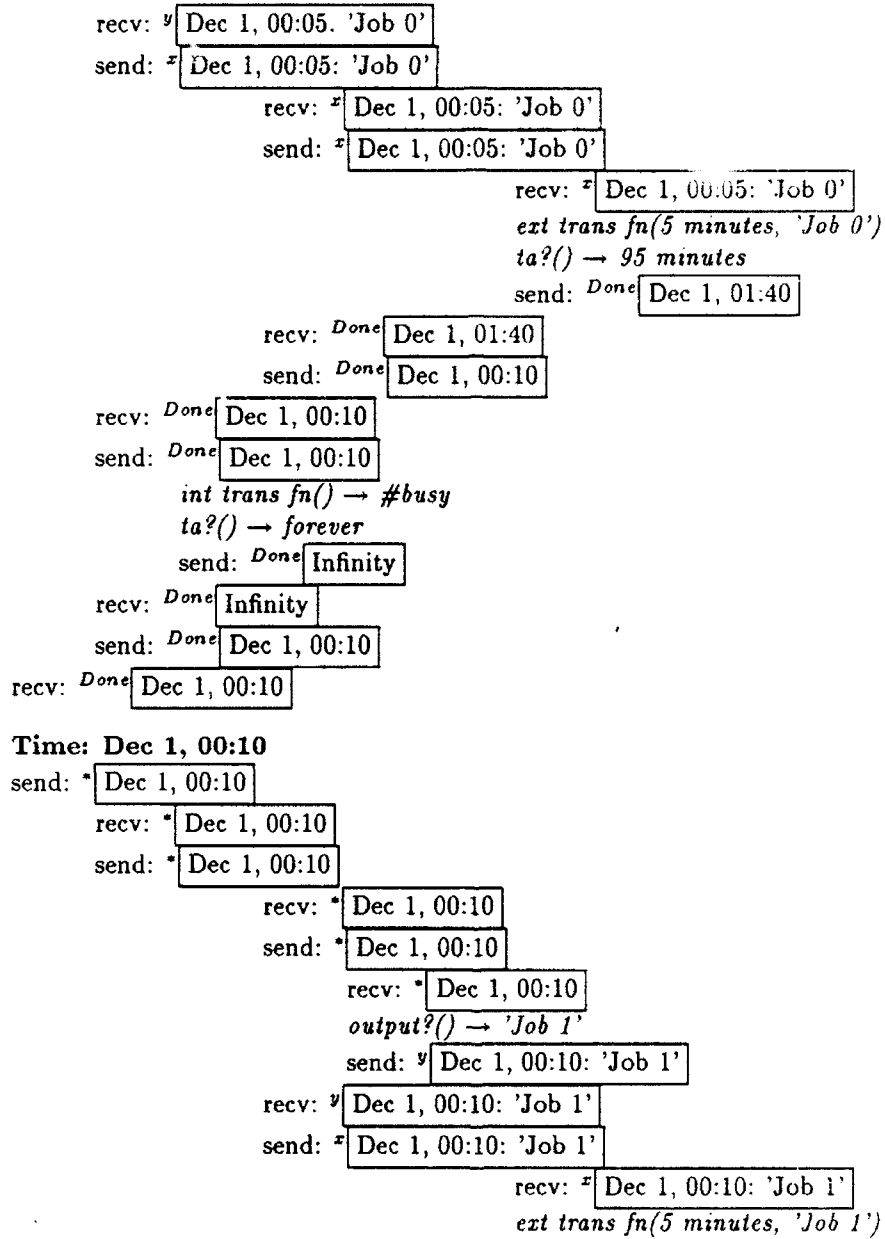
```

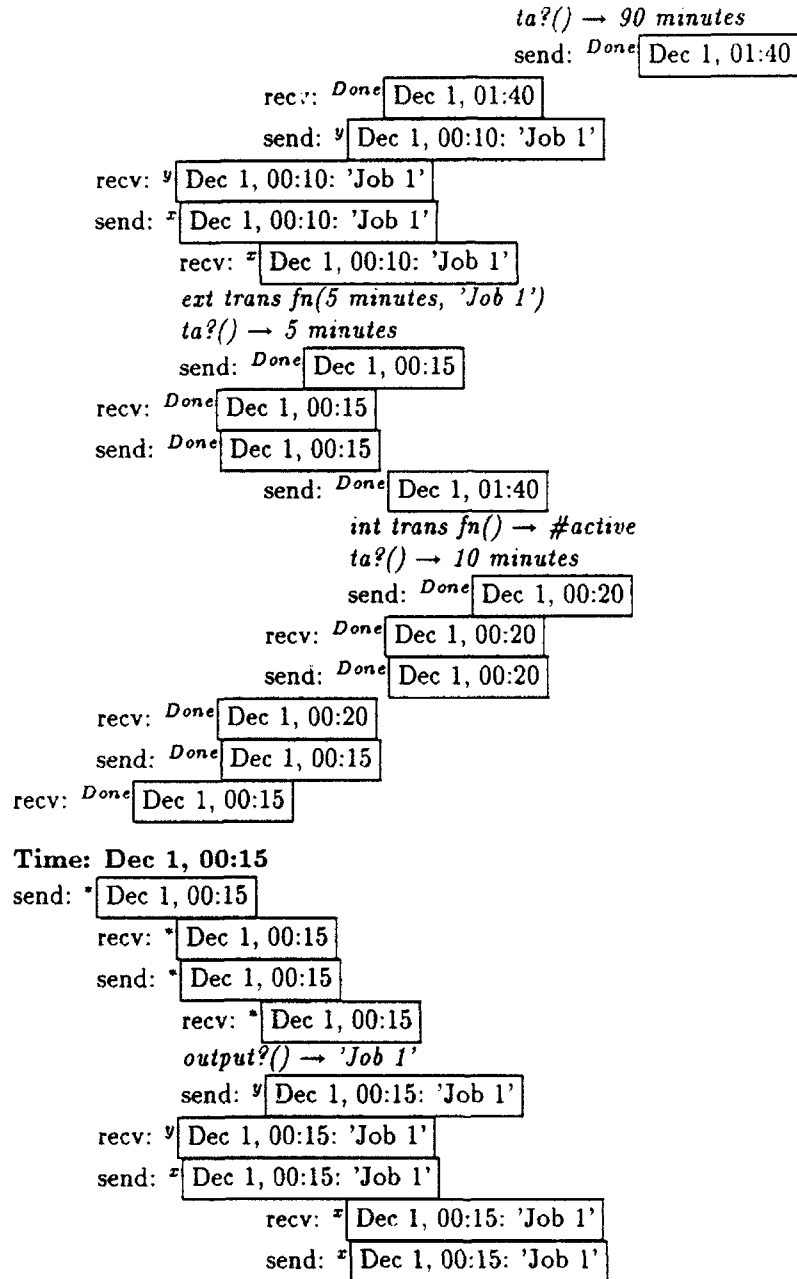
at the lowest level. Obviously, the amount of messaging that occurs in a simulation run is very large and being able to format that output meaningfully is of considerable help in debugging. See Section 3.4 for details on how to produce this output.

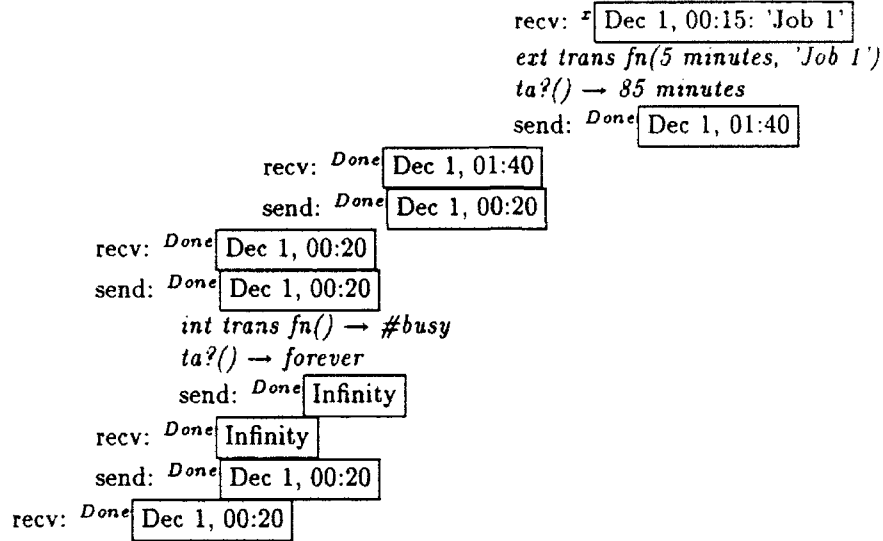
RC EF-P S:P EF S:GENR S:TRANSD



RC EF-P S:P EF S:GENR S:TRANSD



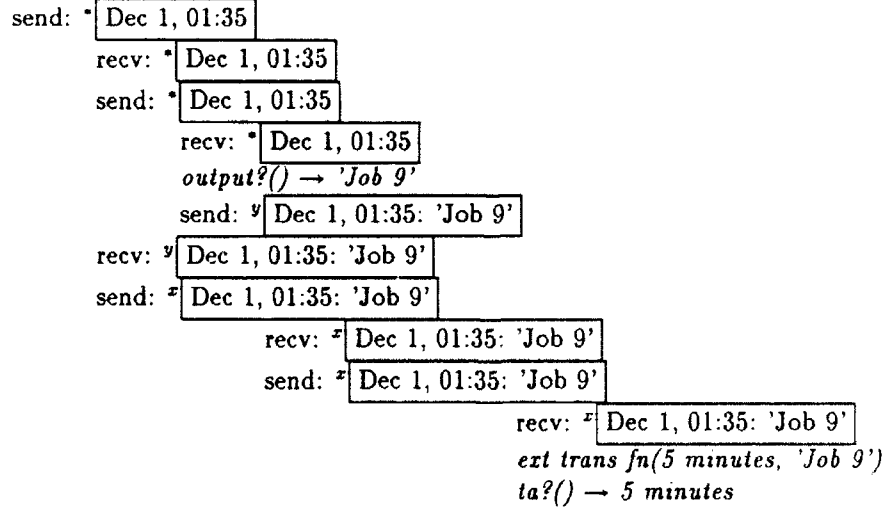


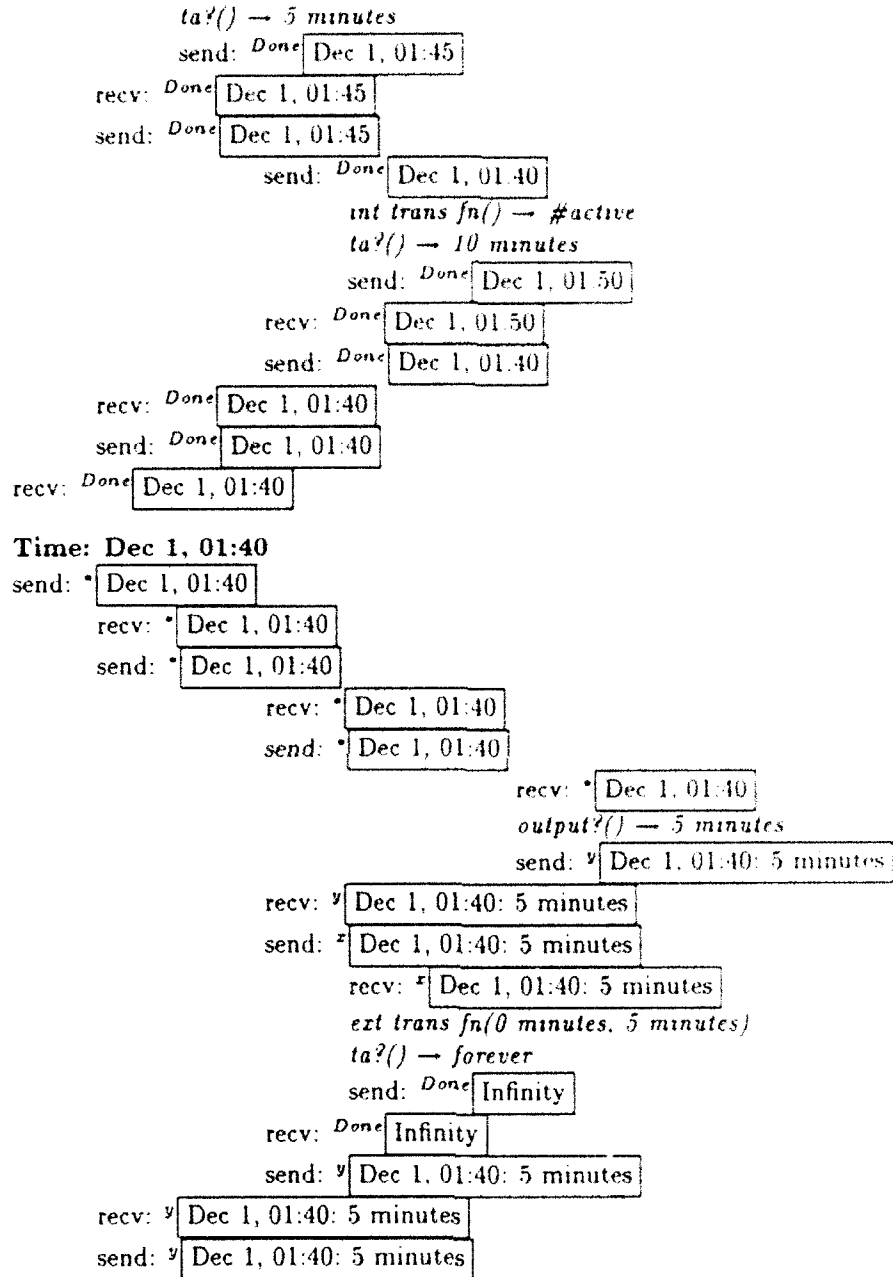


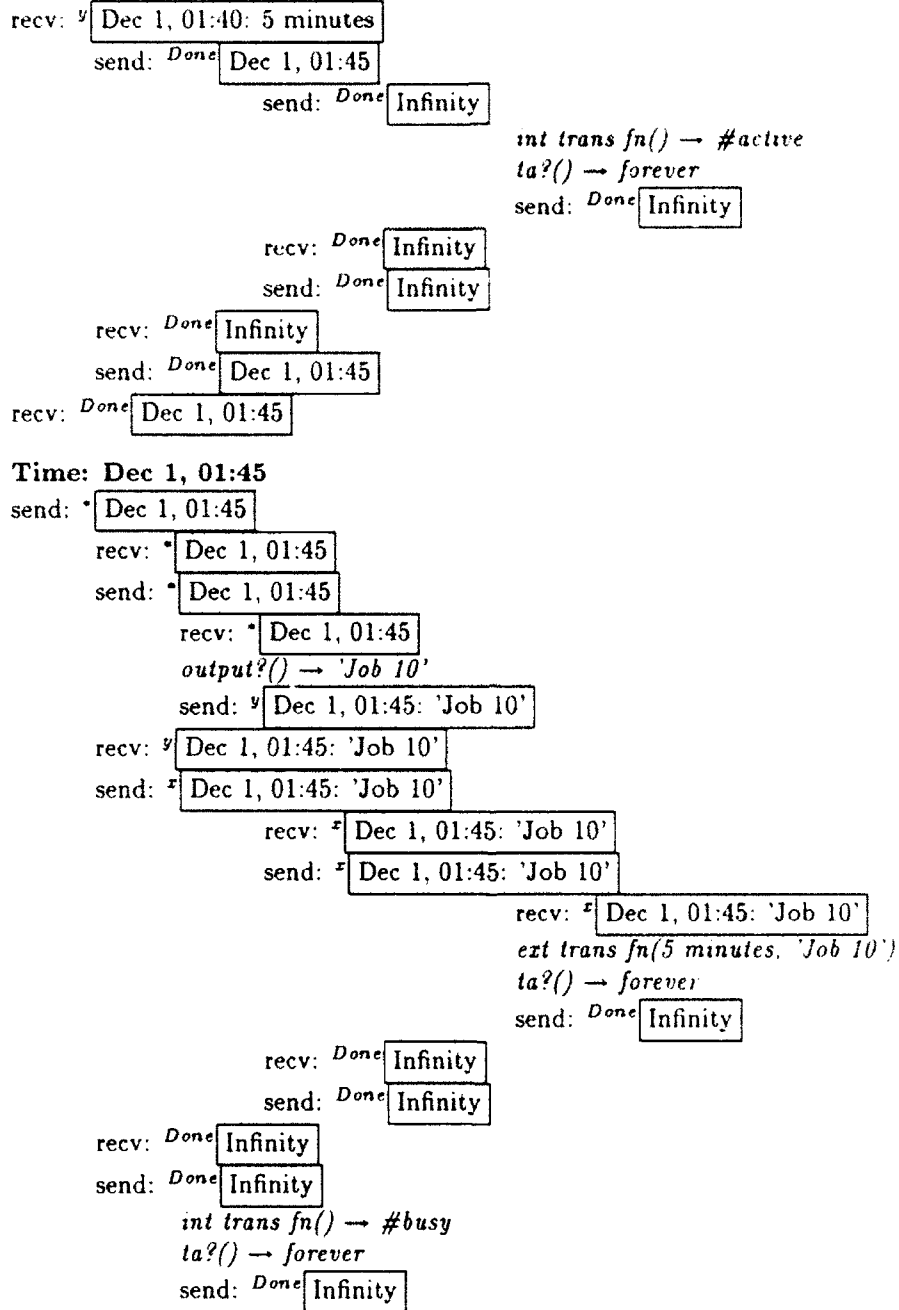
Time: Dec 1, 00:20

⋮
A similar pattern of messages intervenes
 ⋮

Time: Dec 1, 01:35







recv: ± Dec 1, 00:00: 'Job 0'

Chapter 3

System Design

We chose to use DEVS [Zei90] as the methodology for constructing the simulation components of our system. The DEVS methodology is modular and hierarchical, supporting the development of independent models that are linked. An implementation in Scheme, an object-oriented programming language based on Lisp, is described in [Zei90] and was to be the basis for our implementation.

DEVS-Scheme was also attractive because it was developed with running on multicomputers in mind[Zei90, p. 62]:

... The implementation in DEVS-Scheme has the characteristics of a "virtual multiprocessor" in that each of the processor objects could in principle be assigned to a different physical computer. This renders modelling in DEVS-Scheme a natural basis for implementing discrete event models on multi-computer architectures.

We were interested in being able to run our cutting room simulations in a multiprocessor environment because we expected them to take quite a while to complete when run on a desktop computer. Part of our project was to examine ways to distribute the simulation. Research on distributing DEVS-Scheme would simplify that task for us.¹

The system comprises three main categories of classes:

- DEVS classes
- application classes
- user interface classes

¹Because of our significant underestimation of the programming effort required for our simulation, we did not get a chance to investigate distribution of simulator components. We note that execution time on a Macintosh (M68040 processor) running Objectworks\Smalltalk (Version 4.1), the machine and programming language environment used for development, is acceptable for at least the small test simulations we have run.

A description of the main classes in each of these categories is provided in the next three sections. The remaining sections in this chapter provide descriptions of support for debugging and testing.

3.1 DEVS Implementation in Smalltalk-80

Our implementation of DEVS in Smalltalk-80 parallels Zeigler's implementation in Scheme, but differs in some aspects arising primarily from a different perspective on the application of object-oriented technologies. Zeigler's implementation was influenced by the view of objects presented by Scheme. His approach is to define a class, then alter the contents of that class as appropriate for a particular model. For example, the simple processor model we discussed earlier would be an instance of class `AtomicModel` whose list of state variables is augmented by `jobID` and `processingTime` to represent the processor's state. Our approach uses subclassing: a class `SimpleProcessor` is a subclass of a class `AtomicModels`, inheriting the state variables and methods required to be an atomic model, then adding in new instance variables and methods to simulate the state and behaviors of a simple processor. Our approach is cleaner from a programming view, but a little further removed, perhaps, from the DEVS formalism. For example, time advance is no longer a function (object), but is a method provided by a model object.

The implementation that we describe has evolved from an initial implementation that was faithful to the Scheme implementation. As we began to figure out how a DEVS simulation worked, we began to refine our implementation so that it became easier to create models. Our goal is to make the addition of new classes of objects that represent cutting room resources as straightforward as possible. As work progressed, we revised Zeigler's design to suit our needs. Over the term of the project, we reimplemented significant portions of the classes we describe at least two times. Our implementation could probably use one more good review to flush out the remnants of the original design. We didn't want to take the time away from using these classes to implement simulations, so we never gave ourselves an opportunity for another review.

We have restricted our implementation to include only those classes that we need for *CUIP* simulations—namely, the only coupled models we use are digraph models. We have used the same class hierarchy and have defined the same set of support classes. However, we have extended the Scheme implementation with respect to how we handle time and with respect to how we handle instance variables. We have also added a special kind of digraph model that supports dynamic change to the composition tree. In this section we describe our implementation of DEVS in Smalltalk-80. Class specifications for all the classes described here may be found in Appendix A.

Detailed descriptions and specifications of DEVS and application classes described in the remainder of this chapter are provided in Appendices A and B.

```

Entities ('name' 'parent')
  Models ('processor' 'inport' 'outport')
    AtomicModel ('x' 'y' 'sigma' 'phase' 'e')
    CoupledModels ('receivers' 'influencees' 'priorityList')
      DigraphModels ('compositionTree' 'influenceDigraph' 'selectFn')
      DynamicDigraphModels ()
  Processors ('devsComponent' 'childProcessors' 'timeOfLastEvent' 'timeOfNextEvent')
    Coordinator ('starChild' 'waitList' 'tNChildren')
      DynamicCoordinator ()
    RootCoordinator ('clock' 'child' 'startTime' 'timeLimit')
    Simulators ()

```

Figure 3.1: The DEVS class hierarchy. (Instance variables are shown in parentheses.)

3.1.1 DEVS Entities

The hierarchical structure of classes for DEVS entities is shown in figure 3.1. Class **Entities** is common to all DEVS classes². This class corresponds to Zeigler's class of the same name and is as an abstract class, holding the common attributes of its subclasses. The common attributes are the *name* of the entity and the *parent* of the entity. Zeigler does not include the parent in this class, but duplicates it in subclasses. We have elevated this common attribute to the **Entities** class. Zeigler includes a list of instances in his **Entities** class, but Smalltalk maintains the list of instances for us so we don't include it. Below **Entities** in the hierarchy are **Models** and **Processors** corresponding to DEVS models and processors, respectively.

Models

The abstract class of models comprises two main subclasses: **AtomicModels** and **CoupledModels**. Common attributes of these subclasses are:

- *processor*—the processor attached to the model during a simulation run.
- *inport*—a list of the input ports for the model.
- *outport*—a list of the output ports for the model.

²We have not shown it here, but **Entities** is a subclass of Smalltalk's **Model** class, allowing us to make any DEVS model a component of the Model/View/Controller framework. We will discuss this more fully in regard to the design of the user interface.

Zeigler includes attribute *cell position* in this class, but that attribute is associated with kernel models, a class of model that we don't support³.

We require each model subclass to provide a *restart* method that, when dispatched, initializes the model to some base state and answers the duration to the next internal transition. Zeigler seems to use a method *initialize* for this purpose, but we reserve the method *initialize* for the more conventional application of initializing an instance upon creation. Use of a separate *restart* operation allows us to construct a hierarchy of models once and restart all its components any number of times for a variety of simulation runs. By convention, any subclass of *Models* that overrides *restart* should include "super restart" as part of the method's definition.

Atomic Models Atomic models comprise the leaves of the hierarchical construction of models in the DEVS methodology. Recall that an atomic model is characterized by state variables (*sigma* and *phase*), an internal transition function, an external transition function, an output function, and a time advance function. In our implementation, state variables are represented by instance variables and the functions are represented by methods. Every instance of *AtomicModel* has instance variables *phase* and *sigma* and four methods:

<i>intTransFn</i>	<i>implements the internal transition function</i>
<i>extTransFn</i> : aTime externalInput: aContent	<i>implements the external transition function</i>
<i>outputFn</i>	<i>implements the output function</i>
<i>timeAdvanceFn</i>	<i>implements the time advance function</i>

We have omitted the state parameter from all of the functions because in our design, each model maintains its own private state. Nor do the transition functions return a state. Instead, these functions modify the state of the model on which they are applied. Zeigler's design represents each of these functions as a lambda expression (Lisp function) separate from an instance and therefore each requires a state parameter to establish a context in which the function is evaluated. His design arises from the way in which Scheme associates methods with instances. Smalltalk lets us use a much simpler design.

All functions except for the time advance function must be implemented by subclasses of *AtomicModel*. The time advance function defined in class *AtomicModels* answers the current value of *sigma*. In most cases, this behavior is the one needed, so this function definition is seldom overridden. An *AtomicModel* instance has three state variables that are maintained internally:

³In DEVS, a coupled model can be either a digraph model or a kernel model (in which all its component models are of the same class). In C/IP, we have needed only digraph models, so *DigraphModels* is the only subclass of *CoupledModels* that appears in the hierarchy.

- e* the elapsed time in the current state
- x* the external input causing the most recent event
- y* most recently generated content for output

Access methods exist to read and write each of these. Variables *e* and *x* are set when the external transition function is dispatched. Variable *y* is set when the output function is dispatched.

The useful actions taken by atomic models are implemented as macros in DEVS-Scheme and as methods in class `AtomicModel`:

Action	Method
<i>hold-in</i> phase for duration	holdIn: aPhase forTime: aDuration
<i>passivate in</i> phase	passivateIn: aPhase
<i>passivate</i>	passivate
<i>continue</i>	continue

Two macros supporting the generation of output are also implemented as methods in the class:

Action	Method
<i>send</i> value to port	send: aValue toPort: aPort
<i>noOutput</i>	noOutput

An instance of `AtomicModel` is created using the class method `new: aString` or using class method `makePair: aString`, where `aString` designates the model's name. The latter method automatically attaches a simulator to the model. The name given the simulator is the name of the model prefixed by "S:".

Coupled Models Coupled models comprise the internal nodes of the hierarchical construction of models in the DEVS methodology. These models supply the mechanism for building complex models from simpler models. Class `CoupledModels` serves to establish a common protocol for all kinds of coupled models. The information in an instance is:

children	<i>the set of component models.</i>
receivers	<i>associates an input port of this model with the children who are connected to it.</i>
influencees	<i>connections between children's output ports and input ports. Each component is a coupling.</i>
priorityList	<i>used to break ties resulting from two children having the same time to next event. The first child in this list has highest priority. The select function (p. 56) uses this list.</i>

The first three instance variables appear in the DEVS-Scheme implementation. Our implementation includes a `priorityList` that orders the children. The default select function definition uses this list to determine the order in which transition functions are invoked on children models that are ready at the same simulation time—the earlier a model appears in the list, the higher its priority. A subclass can override this behavior by defining its own `selectFn`.

Method `getChildren` answers the children of the receiver—the collection of models in the receiver's `children` instance variable. The following methods are specified by class `coupledModels` to be provided by subclasses:

getReceivers	<i>answers a list of children that will receive an external event input to the receiver.</i>
getInfluencees	<i>answers the list of children to which the output of the imminent child is an input.</i>
translate:model:	<i>provides port-to-port translation—that is, provides the input port to a model that is connected to a specified output port of another model.</i>

While DEVS defines a variety of coupled models, our implementation focuses on only the most general kind: digraph models. Digraph models contain a heterogeneous mixture of children and/or non-regular couplings between children (see [Zei90, Chapter 5]).

In *C&IP*, every coupled model is an instance of a subclass of class `DigraphModels`. Class `DigraphModels` contains the fundamental instance variables and behaviors of digraph models. Recall that a coupled model is characterized by its children models and the couplings of ports between the children, and between the model itself and its children as specified by: a composition tree, an influence digraph, and a select function. In our implementation, the composition tree and the influence digraph of a coupled model are represented by instance variables `compositionTree` and `influenceDigraph`, respectively, and the select function is represented by a method, `selectFn`.

Processors

The class `Processors` has three subclasses corresponding to the three kinds of processors in DEVS: `Simulator`, `Coordinator`, and `RootCoordinator`. The attributes common to these subclasses are placed in `Processors`:

- *devsComponent*—the model attached to this processor instance during a simulation run.
- *timeOfLastEvent*—the simulation time at which the last event in the DEVS component occurred.
- *timeOfNextEvent*—the simulation time at which the next event in the DEVS component is scheduled to occur.

Each `Processors` instance responds to a `restartAt: aSimulationTime` message that initializes the receiving processor and its DEVS component (via a `restart` message). The definition for this method is a subclass responsibility. The method must return a *done* message whose time supplies the simulation times at which the processor's DEVS component is scheduled for its next internal transition.

Our implementation of message passing is different from that used in DEVS and DEVS-Scheme with respect to the handling of *done* messages. Instead of a processor explicitly sending a *done* message to its parent processor, a processor implicitly sends the message by returning it as the reply to an *x* or a *** message. This convention we have adopted has the cost of slowing down a concurrent implementation because of the synchronization involved with waiting for a return message, but has the benefit of simplifying our implementation, especially since we could dispense with managing a *wait list* in coordinators. Our implementation is slightly more efficient, and simpler, than that used in DEVS-Scheme.

Simulators Class `Simulators` instances respond to the receipt of *x* and *done* messages, replying in either case with a *done* message. Messages are received via messages with selectors `xMessage:` and `starMessage:`, respectively:

- `xMessage: anXMessage`—verifies that the message time lies between the time of the last event and the time of the next event. If so, then the receiver sends its DEVS component the elapsed time, forwards the content of the *x*-message to the model, applies the model's external transition function, updates the local time of last event and time of next event, and returns a *done* message to report the time of the next event.
- `starMessage: aStarMessage`—checks that the time in the message and the time to next event agree. If so, then the processor gets an output value from its DEVS component by invoking its output function and sends the

value in a *y* message to its parent coordinator, has the model update its state via its internal transition function, updates the local time of last event and time of next event based on the model's time advance function, and returns a *done* message to report the time of the next event.

In addition, an instance responds to a *restartAt:* message:

- *restartAt:* *aSimulationTime*—sets the time of last event to the time indicated in the message and sends a *restart* message to its DEVS component. Based on the duration returned by the model, the processor sets the time to next event and replies with a *done* message carrying that time.

This message must be sent before a simulation can be run.

Coordinators A class *Coordinator* instance handles the processing associated with the coupled model paired with it as its DEVS component. Since a coupled model contains other models, each connected to a *Processors* instance, a coordinator has child processors determined in the obvious way. It is these children that a coordinator manages—that is, simulation is effected by a coordinator managing its children more than a coupled model managing its children in response to transitions.

Each *Coordinator* instance maintains a list of children in an instance variable, *tNChildren*. [The name was invented by Zeigler.] This list contains *done* messages rather than processors, each such message containing a simulation time and a message originator, in this case the originator being one of the coordinator's children. The messages in *tNChildren* are sorted in increasing order of the times they contain. Only one message for each child can appear in the list.

Each coordinator also maintains a variable naming its imminent child, *starChild*. The imminent child is set each time a *** is received and processed. This child is a model—one of the children of the coordinator's DEVS component.

A coordinator responds to *x*, ***, and *y* messages:

- *xMessage:* *anXMessage*—verifies that the message time lies between the time of the last event and the time of the next event. If so, then the receiver determines which of its children have an input port connected to the port on which the external input is arriving. To each of these children, the coordinator sends an *x* message and places the resulting *done* message into its local *tNChildren* that contains a list of *done* messages, sorted by time. Thus, the imminent child appears at the front of this list. The time of the last event is updated to the time appearing in the *x* message, and the time of the next event is set to the time indicated in the *done* message appearing at the front of *tNChildren*.
- *starMessage:* *aStarMessage*—sets *starChild* to the imminent child from *tNChildren* and removes that child from the list. [Note that *tNChildren* is

sorted by simulation time. If two or more *done* messages at the front of this list have the same time, then the select function of the DEVS component is used to determine which child is imminent.] Next, the * message is forwarded to the imminent child's processor and the resulting *done* message is inserted into *tNChildren*. A wave of *y* and *x* messages ensue, then the coordinator updates its time of last event to the time in the * message and sets the time of the next event based on the time in the message at the front of *tNChildren*.

- **yMessage: aYMessage**—determines the other models within the DEVS component that are influenced by the output produced by the imminent child and routes the content of the *y* message to each in an *x* message to its processor. The list *tNChildren* is updated to include the *done* message produced by each such send of an *x* message. Next, the coordinator determines whether the output should be routed to the parent in a *y* message—that is, whether the output generated by the imminent child is also an output of the coupled model. If so, then the output value is wrapped in a *y* message and sent to the parent. [The returned *done* message is ignored.] Finally, a *done* giving the time in the message at the front of *tNChildren* is constructed and returned to the sender of the *y* message.

In addition, a coordinator responds to a **restartAt: message**:

- **restartAt: aSimulationTime**—sets the time of last event to the time indicated in the message and sends a **restartAt: message** to each of the processors for the component models of its DEVS component. Each **restartAt: message** is responded to with a *done* message and these are collected in *tNChildren*. Based on the simulation time in the first entry in this list, the processor sets the time to next event and replies with a *done* message carrying that time.

This message must be sent before a simulation can be run.

Root Coordinators A **RootCoordinator** instance manages a simulation run of a model by sending a sequence of * messages to a coordinator to which it is attached. The coordinator is the root of a model hierarchy.

A simulation is constructed by attaching a coordinator to an instance of **Root Coordinator**. The root coordinator must first be sent one of three messages to reset the simulation, all of which send a **restartAt: message** to the attached coordinator:

- **restart**—equivalent to **self restartAt: (self startTime)**, where *startTime* is assumed to have been set to a simulation time with accessor *startTime*.
- **restartAt: aSimulationTime**—equivalent to **self restartAt: aSimulationTime andRunFor: Duration infinite**.

- **restartAt: aSimulationTime andRunFor: aDuration**—establishes a start time and an end time for a simulation run—that is, the simulation run terminates when the simulated time reaches the specified start time plus the specified duration.

A simulation run actually starts when a root coordinator receives a *simulate* message.

Since a root coordinator acts as a parent processor for a coordinator, it can receive *y* messages corresponding to the output of the model paired with the coordinator. In our implementation, *y* messages are simply ignored.

A root coordinator also contains support for a window interface. This support—in the form of processes and semaphores—is discussed in detail in connection with the user interface (see Chapter 3.3).

3.1.2 Extensions

In our design, we have extended DEVS to include coupled models whose components vary over time. While this may violate the formalism of DEVS, we found it useful for modeling the workstations at a piece of equipment. We define a workstation to be a portion of a piece of equipment. For example, a spreading table might be used as a single workstation for producing a long spread, and then used subsequently as two workstations for working on two spreads simultaneously. Without the ability to vary the components of an equipment model, we would either have to include a large number of components—for example, one for each potential workstation—or we would have to implement a discrete event simulation algorithm to model the various workstations in use at a given time.

We have restricted our extension to digraph models, using the term *dynamic* to describe the behavior. The implementation of a dynamic digraph model requires the cooperation of an attached coordinator since the two objects work together to implement the behavior of any digraph model. Consequently, the conditions under which a coupled model can change its components are restricted to those that maintain the integrity of the coordinator and all other processors in the hierarchically structured simulation whose states are based on the state of this coordinator. We use a subclass, *DynamicCoordinator*, to effect changes in a way that is invisible to other processors.

Recall that a coordinator maintains a sorted list *tNChildren* that provides information about the times at which each component model is due for a transition. One entry in this list identifies the imminent child, and the time of next event for this imminent child is reported by a coordinator as the time of next event for the coupled model. Thus, any change within the model must be done between the activation of the processor with an *x* or a *** message and the subsequent emission of a *done* message by the coordinator.

It is interesting to note that a coordinator does not send *x* messages to a

coupled model, but rather sends such messages directly to its component models' processors based on information provided by the coupled model in messages such as `getInfluences` and `getReceivers`. Consequently, one of the component models must take responsibility for restructuring the components of a coupled model and, in fact, this model must be an atomic model.

An atomic model might want to change a digraph model's component structure at any transition point, corresponding to the arrival of a `*` or an `x` message to the coordinator for the dynamic coupled model. We restrict the decision to change to the imminent child, an atomic model. Consider two cases:

1. The arrival of a `*` message signals the occurrence of an internal transition and is forwarded to the processor corresponding to the imminent child in the form of a `*` message. Since the imminent child is an atomic model, then its simulator will invoke its output function and pass the output to the simulator's parent, then invoke the model's internal transition function, and then invoke the time advance function. Based on this time as provided in a final *done* message provided to the coordinator, the coordinator determines the schedule for this atomic model's internal transition.

If the atomic model wishes to restructure the digraph model, then it must do so when it is the imminent child, and it must take responsibility for updating the state of the coordinator. It must preserve itself as the imminent child and is allowed only to change the internal and external couplings, including removing and adding components in the process.

2. The arrival of an `x` message signals the occurrence of an external transition. The message is distributed to all component models connected to the input port on which the message arrives. The processing required is the same as described above for a `*` message.

Thus, an atomic model can restructure the dynamic coupled model containing it, but can do so only as an imminent child and by preserving itself as the imminent child. Consequently, a dynamic model will typically contain an atomic model that acts as a manager for the model—essentially a model that monitors the activities of the other components and activates itself by becoming the imminent child, taking appropriate action, and returning to a passive, monitoring state.

This design violates the modularity of DEVS because certain atomic models are aware of their status as a component in a coupled model. A better design would, perhaps, make what we have described as a manager a part of the coupled model itself and somehow coordinate more closely with the coordinator. Our approach is based on the way in which coordinators interact directly with the processors of component models of a coupled model.

Independent of the best way to include these capabilities, it is clear that DEVS as defined was not sufficiently powerful in providing the functionality we needed to handle multiple, dynamically varying workstations within equipment

models. Our design of dynamic entities addresses the problem by extending DEVS.

3.1.3 Support Classes

A number of classes have been defined to support the DEVS entities and simulation. These classes are presented in this section.

DEVS Messages

The four types of messages used to coordinate processing during a simulation run are structured as follows:

```
DEVSMessage ('source' 'time')
  DEVSIOMessage ('content')
    XMessage ()
    YMessage ()
  DoneMessage ()
  StarMessage ()
```

This organization's structure reflects the fact that *done* and *** messages carry no content. All messages carry a time stamp and the processor that originated the message. The time stamp is an instance of class `SimulationTime` (see section 3.1.3). The model is an instance of class `Processors`.

A message content is represented by an instance of class `Content`. A content has two attributes: a port and a value. The value is generated by a model and can be any object. A port is an instance of class `Port` which designates a model and a port name. The name of a port is always represented as a Smalltalk symbol—for example, `#in`. A port can be constructed by sending a comma message to a model—for example, `(M , p)` creates an instance designating port `p` of model `M`.

We note that at some points during a simulation, a processor might be required to send a *y* message even though no output is available. Like Zeigler, we represent a null content explicitly—that is, as a `Content` instance that has null port and value attributes.

Model Components

The structure of a digraph model is represented by an instance of `CompositionTree` that specifies the coupling among the component models. An instance represents a specific composition tree as defined in [Zei90, pp. 29ff, p. 56]. Protocol provided for an instance supports the construction of the tree and the querying of the structure and various couplings. Note that Zeigler is not clear about where coupling information is stored. We have selected to store it in the composition tree. An instance has five components.

root	<i>the root of the composition tree.</i>
leaves	<i>the leaves of the composition tree.</i>
extInpCoup	<i>the external input coupling which connects the input ports of the coupled model to one or more of the input ports of the components—this directs inputs received by the coupled model to designated component models.</i>
extOutCoup	<i>the external output coupling which connects output ports of components to output ports of the coupled model—thus when an output is generated by a component it may be sent to a designated output port of the coupled model and thus be transmitted externally.</i>
intCoup	<i>the internal coupling which connects output ports of components to input ports of other components—when an input is generated by a component it may be sent to the input ports of designated components (in addition to being sent to an output port of the coupled model).</i>

A coupling is represented as an instance of class *Coupling* which is simply an ordered pair of ports designated as the *from* port and the *to* port.

Simulation Time

In his description of DEVS Scheme, Zeigler is vague with respect to simulation time. In all of his examples, time is represented as an integer value that starts at zero and progresses toward (positive) infinity, which is represented by the Lisp atom 'inf. This representation of time was not useful for us: we needed to keep track of date and time of day because, for example, an operator's work schedule is a significant factor in determining when a task is completed or when an event in a model might occur—for example, a shift change. As a result, we decided to treat simulation time more formally and model it in a class. As such, we have modeled two aspects of time: a simulation time and a duration.

A simulation time is a specific date and time—for example, 1:00 A.M. on 1 January 1993. All times are tracked to the minute. We decided to ignore seconds since it is doubtful that any meaningful results would come out of tracking seconds in a simulation. A special simulation time is *Infinity*, indicating a time very far in the future.

We distinguish a simulation time as just described from a *duration* of time. A duration is a length of time—for example, five minutes or ten days. All models

maintain *sigma* as a duration. An infinite duration is sometimes denoted as *forever*.

Some arithmetic operations are defined on simulation times and durations:

SimulationTime - SimulationTime	→	Duration
SimulationTime + Duration	→	SimulationTime
SimulationTime - Duration	→	SimulationTime
SimulationTime + Duration	→	SimulationTime
SimulationTime - Duration	→	SimulationTime
Duration + Duration	→	Duration
Duration - Duration	→	Duration
Duration + SimulationTime	→	SimulationTime

If any of the operands are infinity or forever, then the result is infinity or forever, as appropriate. Relational operators for comparing two simulation times or two durations are also defined. Comparing infinity to infinity, or forever to forever, is undefined.

Our use of these classes was a convenience in coding up our models of cutting room resources. We augmented the predefined Smalltalk class **Number** with a set of methods to facilitate the creation of durations. The messages `minute(s)`, `day(s)`, `week(s)`, and `month(s)` sent to a number answer a duration.

The classes associated with simulation time and duration are: **SimulationTime**, **Infinity**, **Duration**, and **InfiniteDuration**. Class descriptions may be found in the appendix.

Miscellaneous Classes

Just as we found it convenient to define classes to model simulation time, we defined classes to represent lengths—_inches, feet, and yards. Class **Length** embodies this concept. As in the case of **Duration**, we have added methods to class **Number** to facilitate the creation of lengths—for example, 3 yards produces an instance of length. Arithmetic is also defined for lengths—adding and subtracting lengths is supported as is multiplying and dividing lengths by a number. We do not support the multiplication or division of two lengths. [We didn't want to get involved in tracking units—for example, recognizing that the product of two lengths is an area—because it was not necessary for the simulations we were trying to construct.] The class description for **Length** may be found in the appendix.

3.2 Application Classes

Application classes comprise those subclasses of **Models** representing the cutting room and its resources. The main kinds of models in the cutting room application fall into four main categories:

- Plans and tasks
- Cutting room
- Equipment and workstations
- Resources

We describe each of the main classes.

The classes in the hierarchy relating to application objects is highlighted in Figure 3.2. Note that the class **Entities** is a subclass of **Model**⁴. This allows any DEVS entity to be used as a model in connection with views as described in Section 3.3.

3.2.1 Plans and Tasks

The simulation is controlled by a plan that details the tasks to be performed and the operators and materials assigned to perform them. The purpose of a simulation run is to model the result of the plan in a particular cutting room.

A *plan* is a sequence of work assignments. Each work assignment comprises a task, operators to perform the task, and a workstation on which the task is to be performed. Operators can be specified to perform a task individually or together with one or more others. Additionally, several alternative assignments are possible so that, for example, one of several operators might perform the task, depending on availability⁵. A task is a description of a job to be performed, including one or more materials to be used for the task. For example, a task might be to spread thirty 5-yard layers of white cotton fabric, taking the fabric from two specific rolls. A work assignment would associate that task in the plan with an operator and a workstation. A workstation is simply a portion of a piece of equipment—for example, the spreading task might be performed on the left half of a spreading table.

The work assignments in a plan are to be performed in the order in which they appear. Associated with each work assignment is a status of completion—*unstarted*, *in progress*, *suspended*, and *completed*. Initially all work assignments are marked *unstarted*. Given a plan P , a set of operators O , a set of materials M , and a set of equipment E , do the following until all work assignments are completed:

⁴It is unfortunate that we use two classes with very similar names, **Model** and **Models**. The former is defined by the Model/View/Controller framework. The latter is the corresponding DEVS entity. It is interesting that Zeigler uses the same name (with an "s") for this class. The reader is urged to note the distinction between the two names. In general, we will use the term *model* to refer to DEVS entities unless otherwise noted.

⁵Our current simulation does not take advantage of this facility because it complicates the routing of operators to workstations.

Object

Model

Entities

Models

AtomicModel

Dispatcher

Equipment

CuttingMachine

AutomaticCutter

BITECutter

LaserCutter

ManualCutter

SpreadingMachine

AutomaticSpreader

Table

CuttingTable

SpreadingTable

Planner

CoupledModels

DigraphModels

DynamicDigraphModels

Room

CuttingRoom

Warehouse

Processors

Coordinator

DynamicCoordinator

RootCoordinator

Simulators

Figure 3.2: *CIP* application classes (boldfaced).

For each operator o in $\mathcal{O}$ that is available at the current time, associate o with the next work assignment that is not completed and that entails o .

For each material m in $\mathcal{M}$, associate it with the next work assignment that is not completed and that entails the use of m .

For each work assignment w that is unstarted or suspended, with which sufficient materials and operators are associated, and for which the required workstation is available, mark w as being in progress.

If the operator must leave the workstation before the task is completed, then mark the task as suspended and return the operator to $\mathcal{O}$.

If a task is completed, then mark the work assignment for it as completed and return the operator to $\mathcal{O}$ and any unconsumed materials to $\mathcal{M}$.

Under this discipline, each operator extracts the tasks on which he is scheduled to work and does them in order, realizing that some tasks might be completed while he is off-duty or on break. Similarly, materials are moved around the cutting room from equipment to equipment, depending on where that material is next to be used. The appearance of various resources—operators and materials and equipment—in the plan place a partial ordering on work assignments. This algorithm preserves that ordering, except that in the case in which alternative operators o_1 and o_2 are specified for a given work assignment w , and o_i starts work on the task, then o_j ($j \neq i$) can start work on a task appearing subsequently in the plan, later to return to complete w .

Some tasks require the output of a prior task as materials—for example, cutting tasks generally depend upon the completion of spreading tasks. We represent the result of tasks by *tickets* that are defined in work assignments. A ticket may appear as a required resource in a task. The relation between ticket definition and ticket use also creates a partial ordering on tasks that must be followed in carrying out a plan.

As an example, consider a portion of a plan involving three operators, three rolls of fabric, and two spreading tables:

	Task	Workstation	Operators	Ticket
1.	Spread 4 layers of fabric A, then 4 layers of fabric C.	 Table 1	Huey	
2.	Spread 10 layers of fabric B.	 Table 2	Dewey	
3.	Spread 6 layers of fabric A.	 Table 2	Louie	
4.	Cut  .	 Table 1	Dewey	

The equipment column shows a workstation as a shaded portion of the whole. We designate and represent a workstation internally just that way. A ticket is not required if the result is not of interest, though in this case we need to process the result of each task.

Our current design includes four tasks, each of these is represented by a subclass of the abstract class Task.

- *Spreading*—expressed as, "Spread { $roll_1, roll_2, \dots, roll_n$ } following template T ," where the template specifies how the spread is to be constructed and the fabric rolls satisfy the requirements for fabric specified in the template. The product of this task is a spread.
- *Cutting*—expressed in terms, of "Cut *spread*," where the spread is an instance of Spread or a ticket that represents a spread product of some task. The product of this task is a set of stacks.
- *Moving*—expressed in terms of, "Move *object* to *workstation*," where the object is any material (including a ticket) and workstation designates a workstation, usually one at the same equipment. This task represents an operation such as sliding a spread from one end of a table to the other or positioning a cutter at the end of one spreading table to the end of another.
- *Bundling*—expressed as, "Bundle *set of stacks*," where the set of stacks is an instance of StackSet or a ticket that represents a set of stacks produced as a result of some task.

A task must be able to compute the time needed to complete it by the operators at the workstation. A task also tracks its completion status as a fraction of the whole. This fraction—represented as a percent—is determined by the operator and the materials that contributed to the level of completion so far. Thus, a task that is half-completed by one operator in an hour will require two hours to complete by another operator who works at half the efficiency at the task as the first operator.

3.2.2 Cutting Rooms

A cutting room model is a digraph model comprising a model for each piece of equipment and three special models:

- A *Break Area* in which each operator resource resides when it is off-duty—that is, on break or not at work. However, an operator might be idle, but by a piece of equipment instead of in the break area because an on-duty operator waits by the equipment containing the workstation at which his next task is to start. (The algorithm we use was described in Section 3.2.1.) All operators are in the break area when a simulation run starts.
- A *Drop Area* in which each material resource resides when it is not required for a task. All resources are in the drop area when a simulation run starts.
- A *Planner* that holds the plan controlling the simulation. For historical reasons, this component is called “Oscar” and roughly corresponds to the cutting room manager. This component disseminates the plan for performing the various cutting room tasks to other simulation components and collects simulation results.

This organization is illustrated in Figure 3.3.

`CuttingRoom` is a subclass of `Room`, which is an abstract digraph model class. A `CuttingRoom` instance contains equipment specified at the time of instance creation and the three special models described above. Oscar is an instance of a class `Planner` that has an output port `#plani` connecting to the `#plan` port of the break room, the drop area, and each equipment model (see Section 3.2.3). The plan is distributed to all other component models of a cutting room via these connections. A planner has an output port `#result` through which statistics about the plan are produced when all work assignments in the plan are completed. The plan held by a planner is sent over each `#plani`, $i = 1, 2, \dots, n$, where n is the number of other models in the cutting room (equipment, drop area, and break area), when a plan is received on its input port `#start`. Statistics about the plan are computed based on inputs on its `#done` input port. These inputs are instances of class `WorkAssignmentStatistics` that provide information about the start and end times of the task it includes.

The drop area and the break area in a cutting room are instances of class `Dispatcher`. A dispatcher is an atomic model that manages a pool of resources.

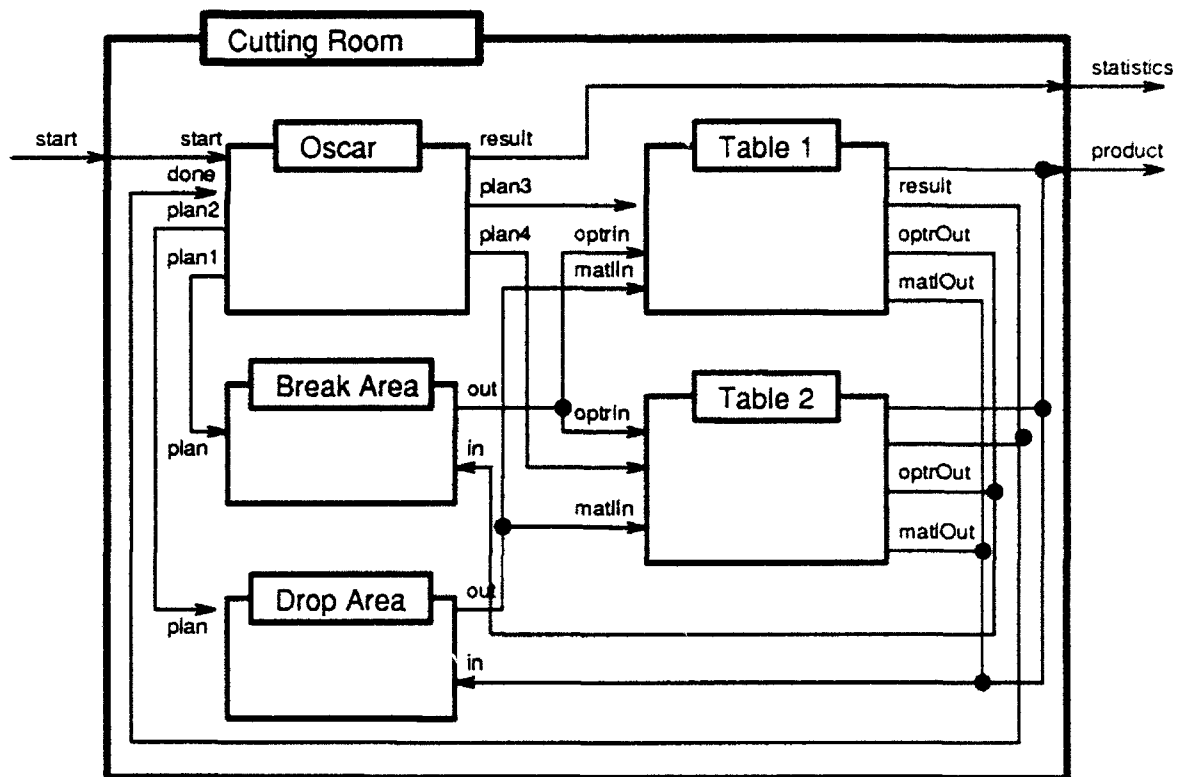


Figure 3.3: Cutting room model components and their connections.

dispatching a resource to another model in accordance with some plan. A dispatcher has two input ports, `#plan` and `#in`, and n output ports: `#out1`, `#out2`, ..., `#outn`. Port `#plan` is an input port on which a plan is to be received that drives the routing of resources to and from the dispatcher. Resources arrive on port `#in` and are dispatched to other models on an `#outi` port.

Each output port is assumed to be routed to one model. The association between model and output port is stored in a dictionary (`mappingDictionary`) and is established at instance creation using the models provided. For example, if the models are m_1 and m_2 , then an association between `#out1` and m_1 and between `#out2` and m_2 is made. When a resource arrives on `#in`, and the plan reflects that the resource should be routed next to m_1 , then the resource is output on `#out1` based on the dictionary association of m_1 and `#out1`. A `CuttingRoom` instance ensures that connections are correct.

Every resource in a dispatcher must be able to answer the message `whereNext: aPlan`, providing the next work assignment in which the receiver is to be used. An answer of `nil` designates that the resource has no more uses within the plan. No resource is dispatched until it is available for use—for example, an operator who is off-duty.

At the start of a simulation, all operator resources are placed in the break area and all material resources are placed in the drop area. Each dispatcher instance sends resources to the equipment model containing the workstation at which that resource is first used, based on its reply to `whereNext: aPlan`. When a model no longer wants a resource, it routes the resource back to the appropriate dispatcher.

3.2.3 Equipment and Workstations

Instances of class `Equipment`—or more properly, instances of its various subclasses—model pieces of machinery. An equipment model has three input ports:

- `#plan` on which a plan arrives before any other inputs are accepted
- `#optrIn` on which operators arrive
- `#matIn` on which materials arrive

and four output ports:

- `#done` on which statistics about a completed work assignment is emitted
- `#product` on which a completed product is emitted
- `#optrOut` on which operators are emitted
- `#matOut` on which materials are emitted

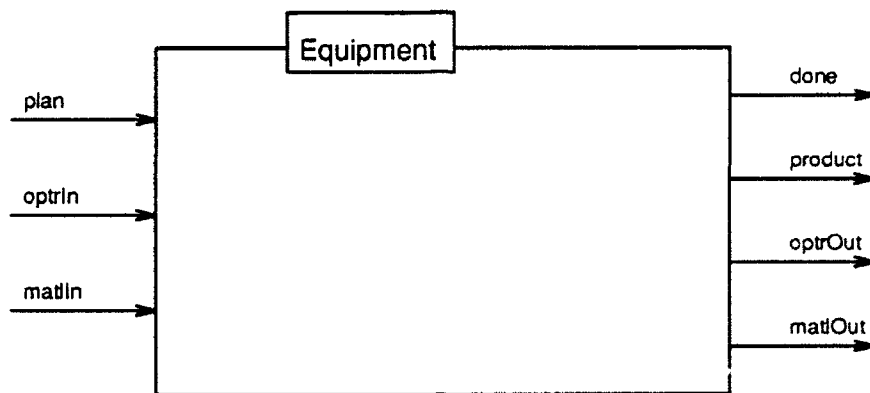


Figure 3.4: An equipment model.

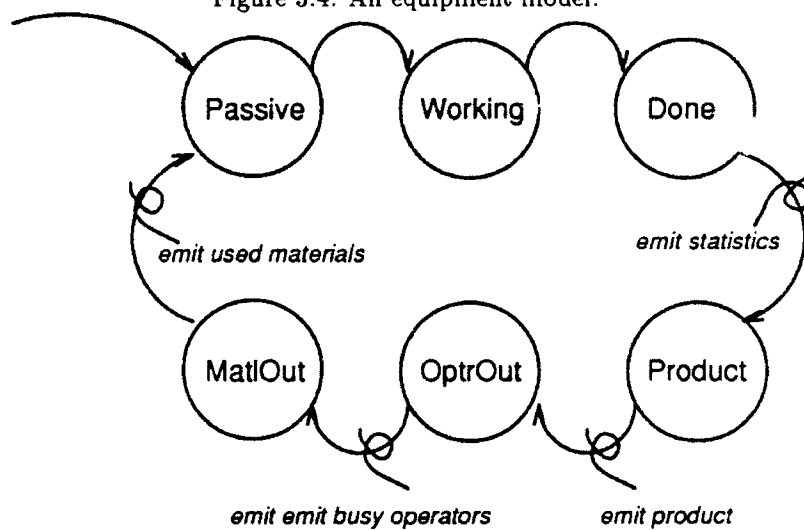


Figure 3.5: States of an equipment model.

Equipment has behavior such that if it has a work assignment in the plan that is still not completed, and sufficient resources and operators are available at the equipment to perform the work assignment, and the workstation for the assignment is available, then work on the task is begun. Upon completion of a work assignment, statistics are emitted on port `#done`, the product is emitted on `#product`, the collection of operators who performed the task are emitted on `#optrOut`, and the collection of materials—what's left of them—is emitted on `#matlOut`. It is sometimes the case that an operator at a workstation, or an operator in the wait area, goes off-duty. In that case, the operator is emitted on `#optrOut` and any task on which he was working is either suspended or is resumed by another on-duty operator in the wait area.

When a resource arrives at equipment, it is put in the wait area and then a check is made to see if an uncompleted work assignment in the local plan—those work assignments for this equipment—can be started based on the presence of all required resources in the wait area and the availability of the workstation specified⁶. When a work assignment is started, eligible operators in the wait area not selected for the task are emitted on port `#optrOut` (where they are routed to the dispatcher).

Every piece of equipment comprises a set of workstations. For this class, only one workstation can be active at a time. (See subclasses for equipment that can support multiple active workstations.)

Each piece of equipment has a wait area at which idle operators and materials wait for the start of a task specified in a work assignment. The wait area is represented by instance variables `waitingOperators` and `waitingMaterials`.

Equipment has the phases and transitions shown in Figure 3.5.

3.2.4 Resources

Resources comprise the operators and materials to be allocated to a set of work assignments. Class `Resource` is an abstract class that contains `Operator` and `Material` as subclasses. `Resource` establishes common state and protocols to determine:

- *job*—the work assignment in which this resource is participating (if any)
- *equipment*—the equipment at which this resource is stationed (if any)
- *status*—the current status of this instance: `#busy` or `#idle`
- *image*—a pixmap that provides a picture of this resource in its current state.

The images are stored in a dictionary and retrieved using the current status as a key.

⁶The current implementation insists that work assignments be started in their order in the plan. However, this is not necessary as long as the partial ordering is preserved. This was just an implementation convenience

Operators

Operators are resources that model the behavior of workstation operators. An operator is not a DEVS entity, but an object that is passed between models and affects their states. Operators are unusual objects in this system in the sense that they are autonomous, acting based on the circumstances in which they find themselves and changing state independently of where they are based on simulation time—for example, quitting work for the day when a shift ends.

We have addressed this autonomy by making operators passive components and having all models that base their states on operators—equipment models and dispatcher models—be aware of operator state. For example, a dispatcher model will consult with each operator object it wishes to dispatch, asking whether it is available at the current simulation time and dispatching it if it is, or basing an internal transition on the time at which it becomes available if it is not. In short, models that use operator resources consult the operators as part of their transition functions.

As a passive object, each operator must be able to respond to two messages:

- `availableAfter: aSimulationTime` to which the receiver answers the duration until it is available after the time indicated
- `availableAt: aSimulationTime` to which the receiver answers whether or not it is available at the time indicated

Thus, an operator instance need only know its work schedule to respond to these messages.

Operator instances have other operations as well, such as being able to answer its efficiency at a given workstation and the various tasks it is asked to perform.

Materials

Materials are resources that are consumed or produced in a cutting room—for example rolls of fabric, markers, spreads, stacks, and so on. The class `Material` serves as an abstract class and all materials are instances of its subclasses. Unlike operators, materials are passive, manipulated by the models.

Certain equipment resources must be considered materials—namely, hand-held equipment such as rotary shears. These items do not function in the same way as other equipment. For one thing, they do not support workstations. For another, they can be moved around just as materials.

3.3 User Interface

The user interface classes provide support for windows that display a simulation run in progress and buttons for controlling a simulation run. The interface is

```

Object
  VisualComponent
    VisualPart
      DependentPart
        View
          CuttingRoomSimulationView
          DEVSRUNView
          DEVSVIEW
          ModelsView
            AtomicModelView
            DispatcherView
            EquipmentView
            PlannerView
            SimpleProcessorModelView
            TransducerModelsView
          ProcessorsView
            CoordinatorView
            RootCoordinatorView
            SimulatorsView
          SimulationRunView

```

Figure 3.6: User interface classes.

based on the Model/View/Controller framework. The simulation model is a subclass of Model. The simulation window contains instances of classes View and Controller.

The user interface classes comprise views of some of the DEVS and application classes. All of the user interface classes are subclasses of the class View defined by Model/View/Controller. The hierarchy is shown in Figure 3.6.

3.3.1 Simulation Run Views

The window placed on a screen to show a simulation run is an instance of SimulationRunView. An instance is created with the message openOn: which takes an instance of a root coordinator as its parameter⁷.

A simulation run view creates a window on the screen containing a view for each atomic model in the tree connected to the root coordinator. The window also displays a pane for the root coordinator that displays the simulation time and other simulation run information.

⁷In the current design, a scenario is a root coordinator, but a scenario class should have been defined!

A simulation run view provides four buttons for controlling a simulation.

Restart sends a *reset* message to the root coordinator.

Go sends a message *go* to the root coordinator.

Pause sends a message *pause* to the root coordinator.

Step sends a message *step* to the root coordinator.

The meanings of these buttons are described in the User's Manual. These buttons can be used at any time, though the Restart button generally can be used only once at the beginning of a simulation run unless all of the models and processors involved have stored their start states to permit resetting. In addition, all consumed materials must be re-created, so restarting is a nontrivial operation, in general. We have included the button in our design primarily to help us with debugging.

The placement of the various views in the window is hard-coded for our prototype. An interface should be defined for more explicit layout of the various views, perhaps with an interactive component. This capability entails more programming than we could address within time constraints. Ideally, any model's view could be placed anywhere within the window. Similarly, coupled models could have their component models arranged arbitrarily within the model's boundaries. The Smalltalk class library supports such operations, but we were unable to take advantage of this because it took us a long time just to figure out how to get the rudimentary set of views installed in a window⁸.

The various views installed in the window display bitmaps intended to portray the status of the model. Consequently, every DEVS entity must be able to provide an instance of *Pixmap* (or one of its subclasses) in response to a *state* message.

3.3.2 Models Views

Class *ModelView* provides an abstract class from which all other model view classes inherit. This class provides a pixmap comprising a rectangle labeled with the name of the model provided upon instance creation. The size of the rectangle is determined by the messages *defaultWidth* and *defaultHeight*. A subclass can override these methods to change the size of the pixmap.

Each model view has a rectangle in which subclasses can place additional information. This rectangle is accessed by subclasses using the message *insetRectangle*. The origin and corner of the inset rectangle can be used to determine where subclasses can put additional state information.

A subclass of *ModelView* is defined for atomic models and for digraph models.

⁸In fact, we were unable to eliminate a bug in our implementation with respect to the buttons. For some reason, the window must be scrolled to the top before any button press will take effect.

Atomic Model Views

Instances of the class `AtomicModelView` display the current values of ϵ , σ , and x for the model.

3.3.3 Processors Views

3.3.4 Synchronization of Views and Simulation Runs

A root coordinator and a view must synchronize their operations. When receiving a `go` message, a root coordinator spawns a process to perform a simulation run. The view controls this process by sending messages to the root coordinator. These messages are:

- `reset`—restart the simulation
- `pause`—interrupt the simulation on the next iteration of the loop that advances global time
- `step`—run an interrupted simulation one time step
- `resume`—continue running an interrupted simulation

These messages control the actions taken by the simulation run process and are translated by the root coordinator into actions on its current *mode* that reflects one of five states for a run:

- `idle`—not yet initialized, or run completed
- `ready`—initialized, but not yet started
- `running`—run in progress
- `suspended`—run suspended
- `stepping`—running a single step

The mode is set using accessors `mode` and `mode:` which implement critical regions using a semaphore `modeSemaphore` to ensure exclusive access. An additional semaphore, `controlSemaphore`, is used to suspend the simulation run process when the mode is not *running*.

This design using a separate process and semaphores was necessary to get the interactive behavior that we wanted in the Objectworks environment. The Model/View/Controller framework is designed based on a transaction-oriented model that cycles between user-initiated stimulus and model response. Our system requires that the system be able to operate—that is, run a simulation—freely on its own, providing the opportunity for user intervention.

3.4 Debugging Support

The system contains code to support debugging in both the form of logging message traffic between processors and models, and in the form of an interactive user interface.

3.4.1 Logging Messages

Messages sent and received by the entities in the system can be logged by sending the name of an open write stream to the class `Entities` (or any of its subclasses) via the message `logStream`. The output is text written to the stream, one message per line. For example, the following sequence will log messages to the file `foo.log`:

```
Processors logStream: 'foo.log' asFilename writeStream.
```

The file can be closed by sending `nil` or another stream object as the parameter:

```
Processors logStream: nil.
```

Logging output is also available in `LaTeX` format by sending `true` as the parameter to a `teXFormat` message:

```
Processors teXFormat: true.
```

The format can be cancelled by sending `false` as the parameter. The output makes use of a number of macros that can be defined for convenience. These macros fall into two general categories:

1. *General DEVS components.* These are the macros that pertain to messages and their contents—messages sent and received, ports, times, and contents

```
\logMessage{text}  
\sendMessage{target entity}{content}  
\logTime{time}  
\receivedMessage{content}  
\outputEh{value}  
\timeAdvanceEh{duration to next transition}  
\extTransFn{elapsed time}{input value}  
\intTransFn{new phase}  
\port{model name}{port symbol}  
\Content{port}{value}  
\DEVSMMessage{type}{originator}{time}  
\DEVSIOMessage{type}{originator}{time}{value}
```

2. *Simulation-specific components.* These are macros whose names are taken from the names of the models and processors in a simulation. The macro name corresponds to the name of an entity—for example, a model named *M* is represented by the macro `\M` and its simulator *S M* by the macro `\SM`. (Any non-alphabetic character is removed from the name to satisfy L^AT_EX naming conventions.) Each of these macros has one argument, the operation being logged by the component associated with the name—for example, `\SM{\receivedMessage{...}}`.

The macros below were used to produce the output in Section 2.2.3.

```

\newenvironment{CLIP}{
\begin{tabbing}
xxxxxx\=xxxxxx\=xxxxxx\=xxxxxx\=xxxxxxxxxx\=xxxxxx \k111
}{
\end{tabbing}
}

\newcommand{\RC}[1]{#1 \\\}
\newcommand{\CEFP}[1]{\> #1 \\\}
\newcommand{\SP}[1]{\> \> #1 \\\}
\newcommand{\CEF}[1]{\> \> \> #1 \\\}
\newcommand{\SGENR}[1]{\> \> \> \> #1 \\\}
\newcommand{\STRANS D}[1]{\> \> \> \> \> #1 \\\}

\newcommand{\PROCESSOR}[1]{\> \> {\em #1} \\\}
\newcommand{\GENR}[1]{\> \> \> \> {\em #1} \\\}
\newcommand{\TRANS D}[1]{\> \> \> \> \> {\em #1} \\\}

\newcommand{\logMessage}[1]{#1}
\newcommand{\sendMessage}[2]{send: #2}
\newcommand{\logTime}[1]{\rule{0em}{4ex}{\bf Time: #1}}
\newcommand{\receivedMessage}[1]{recv: #1}
\newcommand{\outputEh}[1]{output?() $\rightarrow$ #1}
\newcommand{\timeAdvanceEh}[1]{ta?() $\rightarrow$ #1}
\newcommand{\extTransFn}[2]{ext trans fn(#1, #2)}
\newcommand{\intTransFn}[1]{int trans fn() $\rightarrow$ #1}
\newcommand{\port}[2]{#1.#2}
\newcommand{\Content}[2]{#2}
\newcommand{\DEVSMMessage}[3]{${\sim}$#1}$\fbox{#3}$}
\newcommand{\DEVSIOMessage}[4]{${\sim}$#1}$\fbox{#3: #4}$}

```

The output of the simulation run is formatted in a CLIP environment. Note that the simulation contains a processor model *P* and we had to rename the macro

\P to \PROCESSOR to avoid the predefined macro for the paragraph symbol (§). Alternatively, we could have just renewed command \P.

3.4.2 Debugging Views

In addition to logging messages to a file, we have implemented views for interactively monitoring the activity of DEVS entities. These views are subclasses of `View` and one class exists for each kind of DEVS processor:

```
Object
  VisualComponent
    VisualPart
      DependentPart
        View

        ProcessorsView
          CoordinatorView
            RootCoordinatorView
              SimulatorsView
```

Each of these views display the instance variables of their corresponding DEVS entities—for example, a `SimulatorsView` instance displays sigma, elapsed time in the current state, the most recently received x message, and the most recently sent y for its model.

These views are installed within other views and each is attached to the appropriate model upon creation via the `new: aModel` message sent to the class. For convenience, each DEVS class responds to the message `view` with the class appropriate for itself—for example, an atomic model responds with the `SimulatorsView` class.

These views are most useful for debugging and are currently a part of class `SimulationRunView` instances because we have been debugging. At some point, these views should be removed. It is more appropriate to have separate windows providing application and debugging views of a simulation run. That design occurred to us late and we were unable to make the changes required (straightforward and simple as they are) within our time constraints.

3.5 Testing Support

Test cases are provided in some class comments, generally in a section labeled *Testing*. Some classes require a substantial amount of setup—for example, complex networks of objects—so test cases are not provided for them.

The experimental frame example used by Zeigler to describe the various aspects of DEVS has been the basis for our testing of the DEVS components of our system. This model hierarchy is embodied in the classes `SimpleProcessor`.

Generators, and Transducers (see Appendix C). We did not implement classes to explicitly model experimental frames (EF) or experimental frames connected to simple processors. Instead, these digraph models are constructed explicitly in a code sequence found in file `simpleTest`. This code parallels the construction used by Zeigler.

Class `DEVSRUNView` provides a window for a run of this model hierarchy. The window is laid out with a column of views on processors down the left side of the window and a column of views of models down the right. The processor and model views appearing in the same row are paired. This view allows one to watch the message traffic between processors as well as the state changes to the atomic models.

To run this test, open the file `simpleTest` in a file editor window, select all the text, and select *do it*. [The text of this file is given in Appendix C] The program opens a window containing a view of the various components of the model and execution begins after the selection of *Restart* and then *Go*. The statistics collected by the transducer are printed to the transcript window. These results should match the following:

```
*****
The arrived list:
  'Job 5'-->50 minutes
  'Job 0'-->0 minutes
  'Job 2'-->20 minutes
  'Job 4'-->40 minutes
  'Job 1'-->10 minutes
  'Job 3'-->30 minutes
  'Job 6'-->60 minutes
  'Job 7'-->70 minutes
  'Job 8'-->80 minutes
  'Job 9'-->90 minutes
  'Job 10'-->100 minutes
The solved list:
  'Job 0'
  'Job 1'
  'Job 2'
  'Job 3'
  'Job 4'
  'Job 5'
  'Job 6'
  'Job 7'
  'Job 8'
  'Job 9'
Avg. turnaround time: 5 minutes
Throughput: (1/10)
```

Appendix A

DEVS Class Specifications

Specifications for DEVS classes are provided in this appendix. These classes are in the DEVS-Simulation Category in the image `st80-CLIP`.

A class specification follows the format:

Superclass subclass:

Class

A comment about the purpose and usage of the class.

instanceVariable

inheritedInstanceVariable

instance method category	<u>newInstanceMethod</u>
	<u><i>inheritedInstanceMethod</i></u>
	<u><i>overriddenInstanceMethod</i></u>

class method category	<u>newClassMethod</u>
	<u><i>inheritedClassMethod</i></u>
	<u><i>overriddenClassMethod</i></u>

The classes in the DEVS component of *CIP* are structured as:

- Entities [68]
 - Models [80]
 - AtomicModel [88]
 - CoupledModels [82]
 - DigraphModels [84]
 - DynamicDigraphModels [86]
- Processors [69]
 - Coordinator [73]
 - DynamicCoordinator [75]
 - RootCoordinator [77]
 - Simulators [71]

The number in brackets following each class name is the page number on which the specification starts.

Model subclass:

Entities

Entities are the root class in DEVS. This is an abstract class that comprises Models and Processors.

Instance variables:

<i>name</i>	<i><String></i>	<i>—the name of this entity</i>
<i>parent</i>	<i><Entities></i>	<i>—the parent of this entity.</i>

Class variables:

<i>LogStream</i>	<i><WriteStream></i>	<i>—stream to which logging is to occur, or nil. Used by some subclasses</i>
<i>TeXFormat</i>	<i><Boolean></i>	<i>—log output in a TeX format</i>

name

parent

accessing

compositePart

displaying

name
state

initialize-release

visualComponent
initialize

printing

printOn:
printString

private-accessing

name:
parent
parent:

private-logging

teXName
isLogging

instance creation

lst
new

logging access

new:
isLogging
logStream:
teXFormat:

Entities subclass:

Processors

Abstract class for a DEVS processor (see Zeigler Sec 3.5.3). This class provides protocols for linking models in a hierarchy and tracing a simulation run. Tracing is enabled if traceStream has a value other than nil.

Note that we use 'restart' rather than 'initialize' for model initialization. We use 'initialize' in the conventional way to initialize an instance; 'restart' is used to reset an instance for another simulation run.

My subclasses are:

*RootCoordinator
Coordinator
Simulators*

Instance variables:

<i>devsComponent</i>	<i><Models></i>	<i>—my model</i>
<i>childProcessors</i>	<i><Set></i>	<i>—my direct Processors class descendents</i>
<i>timeOfLastEvent</i>	<i><SimulationTime></i>	<i>—the time of the last event</i>
<i>timeOfNextEvent</i>	<i><SimulationTime></i>	<i>—the time of the next event</i>
<i>lastSentMessage</i>	<i><String></i>	<i>—the last message to be sent</i>
<i>lastReceivedMessage</i>	<i><String></i>	<i>—the last message to be received</i>

name

parent

devsComponent

childProcessors

timeOfLastEvent

timeOfNextEvent

lastReceivedMessage

lastSentMessage

accessing

childProcessors

lastReceivedMessage

lastSentMessage

processorTree

processorTreeLeaves

timeOfLastEvent

timeOfNextEvent

DEVS	devsComponent devsComponent: linkToParent: restartAt: state visualComponent initialize printOn: printString child: lastReceivedMessage: lastSentMessage: parent: processors timeOfLastEvent: timeOfNextEvent: log: logReceipt: logSend:to: logTime: view
displaying	
initialize-release	
printing	
private-accessing	
private-logging	
views	
instance creation	lst new new: isLogging logStream: teXFormat:
logging access	

Processors subclass:

Simulators

A simulator controls an atomic model (see Zeigler section 3.5.3 and section 4.5).

Instance variables:

None.

name

parent

devsComponent

childProcessors

timeOfLastEvent

timeOfNextEvent

lastReceivedMessage

lastSentMessage

accessing

DEVS

displaying

initialize-release

printing

private-accessing

private-logging

protected-accessing

views

processorTreeLeaves

restartAt:

starMessage:

xMessage:

state

visualComponent

initialize

printOn:

printString

child:

lastReceivedMessage:

lastSentMessage:

parent:

processors

timeOfLastEvent:

timeOfNextEvent:

log:

logReceipt:

logSend:to:

logTime:

child:

processors

view

instance creation

!st

new

logging access

new

isLogging

logStream:

teXFormat:

Processors subclass:

Coordinator

A coordinator is a processor for a coupled model.

Instance variables:

starChild	<Processor>	—the imminent child—the next one to get a message
waitList	<OrderedCollection>	—processors waiting for messages (not used: we are doing sequential execution so only starChild would ever be in this list)
tNChildren	<SortedCollection>	—a list of my children ordered by increasing time of the next event. Actually, a list of done messages.

Note: When a coupled model is created, then its children's processors are linked as a tree, so there's no need to override method 'linkToParent' in this class.

name
parent
devsComponent
childProcessors
timeOfLastEvent
timeOfNextEvent
lastReceivedMessage
lastSentMessage
starChild
waitList
tNChildren

accessing	processors starChild tNChildren <u>restartAt:</u>
DEVS	starMessage: xMessage: yMessage:
displaying	state visualComponent
initialize-release	initialize

printing	<i>printOn:</i>
	<i>printString</i>
private	<i>nextGuy</i>
	<i>starChild:</i>
private-accessing	<i>child:</i>
	<i>lastReceivedMessage:</i>
	<i>lastSentMessage:</i>
	<i>parent:</i>
	<i>processors</i>
	<i>timeOfLastEvent:</i>
	<i>timeOfNextEvent:</i>
private-logging	<i>log:</i>
	<i>logReceipt:</i>
	<i>logSend.to:</i>
	<i>logTime:</i>
views	<u><i>view</i></u>

instance creation	<i>lst</i>
	<i>new</i>
	<i>new:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>teXFormat:</i>

Coordinator subclass:

DynamicCoordinator

A dynamic coordinator is a coordinator that supports simulation of a coupled model that changes dynamically—that is, that changes its component models as simulation progresses.

A dynamic coordinator adds operations to add and remove sub-models. The methods in this class update the processor only. It is the sender's responsibility to update the model correspondingly.

An added component cannot become the imminent child and a removed component cannot have been the imminent child. An easy way to meet these restrictions is for a dynamic digraph model to contain a persistent model that performs updates during an internal or external transition.

A dynamic coordinator is our own invention. Zeigler defines no such object.

Note that this implementation is incomplete.

name

parent

devsComponent

childProcessors

timeOfLastEvent

timeOfNextEvent

lastReceivedMessage

lastSentMessage

starChild

waitList

tNChildren

access-dynamic

accessing

DEVS

displaying

initialize-release

printing

addChildModel:

removeChildModel:

processors

starChild

tNChildren

restartAt:

starMessage:

xMessage:

yMessage:

state

visualComponent

initialize

printOn:

printString

private	<i>nextGuy</i>
private-accessing	<i>starChild:</i> <i>child:</i> <i>lastReceivedMessage:</i> <i>lastSentMessage:</i> <i>parent:</i> <i>processors</i> <i>timeOfLastEvent:</i> <i>timeOfNextEvent:</i>
private-logging	<i>log:</i> <i>logReceipt:</i> <i>logSend:to:</i> <i>logTime:</i>
views	<i>view</i>
instance creation	<i>lst</i> <i>new</i>
logging access	<i>new:</i> <i>isLogging</i> <i>logStream:</i> <i>texFormat:</i>

Processors subclass:

RootCoordinator

A root coordinator controls a simulation. It serves as the root of a hierarchy of DEVS models and contains one child, either a coordinator (coupled model) or a simulator (atomic model).

Method 'linkToParent:' is used instead of 'initialize' described by Zeigler p. 66.

Instance variables:

clock	<SimulationTime>	—the current global clock
child	<Processors>	—the single child processor-coordinator or simulator. We use this for convenience, rather than 'childProcessors first'
timeLimit	<SimulationTime>	—the time at which a simulation run is to stop
mode	<Symbol>	—the current mode:
#idle	— not yet run or completed	
#running	— run in progress	
#suspended	— run suspended	
#stepping	— running a single step	
processes	<Set>	—the set of processes created. Each run requires the creation of a process. This process is terminated when an instance is released, usually when the window is closed via method 'changeRequest'.
modeSemaphore	<Semaphore>	—Used for mutually exclusive access to mode
controlSemaphore	<Semaphore>	—Used to control each iteration of a run

name
parent
devsComponent
childProcessors
timeOfLastEvent
timeOfNextEvent
lastReceivedMessage
lastSentMessage
 clock
 child
 startTime
 timeLimit
 mode
 traceText
 runView
 processes
 modeSemaphore
 controlSemaphore

accessing

child
 clock
 clock:
 startTime
 startTime:
 timeLimit:
 changeRequest
linkToParent:
 restart
restartAt:
 restartAt:andRunFor:

changing
DEVS

simulate
 yMessage:
state
initialize
 release
 go
 pause
 reset
 resume
 run

displaying
initialize-release

operating

step
 printOn:
 printString

printing

private-accessing	child: lastReceivedMessage: lastSentMessage: parent: processors timeOfLastEvent: timeOfNextEvent:
private-logging	log: logReceipt: logSend:to: logTime:
private-synchronizing	mode
views	mode: <u>view</u>

instance creation	lst new new:
logging access	isLogging logStream: teXFormat:

Entities subclass:

Models

Models is the root class for a DEVS model. [Note that 'Model' is already a class in the library.]

Class variables:

None.

Instance variables:

<i>processor</i>	<i><Processors></i>	<i>—The processor associated with this model instance.</i>
<i>inport</i>	<i><Collection></i>	<i>—The input port designations.</i>
<i>outport</i>	<i><Collection></i>	<i>—The output port designations.</i>
<i>pixmap</i>	<i><Pixmap></i>	<i>—A pixmap that represents the current state.</i>

In addition, sigma is the time left in the current phase.

Each subclass should initialize the input and output port designations for each instance. This class requires protocols for 'initialize' and 'restart'. The distinction is:

<i>initialize</i>	<i>Initialize those components of the instance that are fixed at creation, normally the input and output port names and perhaps defaults for graphical components. Note: a method 'initPorts' is required for initializing port names.</i>
<i>restart</i>	<i>Initialize those components of the instance that define the state of the model at the start of a simulation.</i>

name

parent

processor

inport

outport

pixmap

accessing

inport

outport

processor

processor:

restart

DEVS

displaying

state

initialize-release	extent: <u>initialize</u>
port creation	initPorts
printing	,
private	<i>printOn:</i> <i>printString</i> baseVisualComponent
private-accessing	inport: outport: <i>name:</i> <i>parent</i> <i>parent:</i> <i>teXName</i> <i>isLogging</i>
private-logging views	view
instance creation	<u>new</u>
logging access	<i>isLogging</i> <i>logStream:</i> <i>teXFormat:</i>

Models subclass:

CoupledModels

A coupled model is an abstract class that embodies hierarchical model composition (see Zeigler pp. 59ff).

Instance variables:

<i>children</i>	<i><Collection></i>	<i>—the list of component models</i>
<i>receivers</i>	<i><Collection></i>	<i>—associates an input port of my model with my children who are connected to it. This is maintained by my subclasses.</i>
<i>influences</i>	<i><Collection></i>	<i>—connections between component output ports and input ports. Each component is a coupling.</i>
<i>priorityList</i>	<i><OrderedCollection></i>	<i>—used to break ties resulting from two children having the same time to next event. The first child in this list has highest priority. The select function (p. 56) uses this list.</i>

name
parent
processor
inport
outport
pixmap
receivers
influences
priorityList

accessing	<i>getChildren</i> <i>getInfluences:</i> <i>getReceivers:</i> <i>influences:</i> <i>priorityList:</i> <i>receivers:</i> <i>select:</i> <i>translate</i> <i>restart</i>
------------------	--

DEVS

displaying	<i>state</i>
initialize-release	<i>extent:</i>
	<i>initialize</i>
	<i>initPorts</i>
port creation	<i>,</i>
printing	<i>printOn:</i>
	<i>printString</i>
private	<i>base VisualComponent</i>
	<i>inport:</i>
	<i>outport:</i>
private-accessing	<i>name:</i>
	<i>parent</i>
	<i>parent:</i>
private-logging	<i>texName</i>
views	<i>isLogging</i>
	<i>view</i>
instance creation	<i>makePair:</i>
.logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>texFormat:</i>

CoupledModels subclass:

DigraphModels

Digraph models are coupled models containing a heterogeneous mixture of children and/or non-regular couplings between children (see Zeigler, chapter 5).

Instance variables:

<i>compositionTree</i>	<i><CompositionTree></i>	<i>—defines the component models</i>
<i>influenceDigraph</i>	<i><DirectedGraph></i>	<i>—defines which children influence others, i.e. whose outputs affect whose inputs.</i>
<i>selectFn</i>	<i><BlockContext></i>	<i>—a block having one parameter, an ordered collection from which it answers the next imminent child to be selected. If this variable is nil, then priorityList is used.</i>

name

parent

processor

inport

outport

pixmap

receivers

influencees

priorityList

compositionTree

influenceDigraph

selectFn

accessing

getChildren

getInfluencees:

getReceivers:

influencees:

priorityList:

receivers:

select:

translate

DEVS

	addCouple:port:connectedTo:port:
	buildCompositionTree:
	compositionTree
	connect:to:
	getChildren
	getInfluencees:
	getReceivers:
	select:
	selectFn:
	setExtInpCoup:connect:to:
	setExtOutCoup:connect:to:
	setIntCoup:portName:to:portName:
	specifyChildren:
	translate:model:
displaying	state
initialize-release	extent:
	initialize
	initPorts
port creation	,
printing	printOn:
	printString
private	base VisualComponent
	inport:
	outport:
private-accessing	name:
	parent
	parent:
	texName
private-logging	isLogging
views	view
instance creation	new:
logging access	isLogging
	logStream:
	texFormat:

DigraphModels subclass

DynamicDigraphModels

A dynamic digraph model is a dynamic model in which the component models can be created and destroyed during a simulation run. The dynamics require close cooperation with the coordinator attached to this instance, which must be an instance of DynamicCoordinator.

Note that this class has not been implemented

name

parent

processor

inport

outport

pixmap

receivers

influencees

priorityList

compositionTree

influenceDigraph

selectFn

access-dynamic

addChildModel:

addChildModel:withExternalInput-

Couplings:withExternalOutput-

Couplings:withInternalCouplings:

getChildren

getInfluencees:

getReceivers:

influencees:

priorityList:

receivers:

select:

translate

accessing

DEVS

	<i>addCouple:port:connectedTo:port:</i>
	<i>buildCompositionTree:</i>
	<i>compositionTree</i>
	<i>connect:to:</i>
	<i>getChildren</i>
	<i>getInfluencees:</i>
	<i>getReceivers:</i>
	<i>select:</i>
	<i>selectFn:</i>
	<i>setExtInpCouple:connect:to:</i>
	<i>setExtOutCouple:connect:to:</i>
	<i>setIntCouple:portName:to:portName:</i>
	<i>specifyChildren:</i>
	<i>translate:model:</i>
displaying	<i>state</i>
initialize-release	<i>extent:</i>
	<i>initialize</i>
	<i>initPorts</i>
port creation	<i>,</i>
printing	<i>printOn:</i>
	<i>printString</i>
private	<i>baseVisualComponent</i>
	<i>inport:</i>
	<i>outport:</i>
private-accessing	<i>name:</i>
	<i>parent</i>
	<i>parent:</i>
	<i>texName</i>
private-logging	<i>isLogging</i>
views	<i>view</i>
instance creation	<i>new:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>texFormat:</i>

Models subclass:

AtomicModel

An instance of class AtomicModel represents a specific atomic model. Protocol provided for the object corresponds to the DEVS protocol for atomic models. This implementation varies from Zeigler's in that we use instance variables to hold that state rather than a single variable s.

Instance Variables:

<i>x</i>	<i><Content></i>	<i>—external input causing this event</i>
<i>y</i>	<i><Content></i>	<i>—most recently generated content for output</i>
<i>sigma</i>	<i><Duration></i>	<i>—time remaining (in minutes) to the next internal event</i>
<i>phase</i>	<i><Symbol></i>	<i>—the current phase (state)</i>
<i>e</i>	<i><Duration></i>	<i>—elapsed time in the current phase</i>

name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e

accessing

e
e:
phase
phase:
sigma
sigma:
x
x:
y
y:

DEVS	extTransFn:externalInput: extTransition intTransFn intTransition outputEh outputFn <u>restart</u> timeAdvanceEh timeAdvanceFn status <i>extent:</i> <i>initialize</i> <i>initPorts</i> continue holdIn:forTime: inject:value: inject:value:elapsedTime: noOutput passivate passivateIn: send:toPort: , <i>printOn:</i> <i>printString</i> <i>base VisualComponent</i> <i>inport:</i> <i>outport:</i> name: <i>parent</i> <i>parent:</i> <i>texName</i> logExtTransFn logIntTransition: logOutputEh: logTimeAdvanceEh: <u>view</u>
displaying	
initialize-release	
macros	
port creation	
printing	
private	
private-accessing	
private-logging	
views	
instance creation	makePair:
logging access	new: <i>isLogging</i> <i>logStream:</i> <i>texFormat:</i>

Appendix B

Application Class Specifications

Specifications for application classes are provided in this appendix. These classes are in the **CLIP** Category in the image **st80-CLIP**.

The classes in the application component of **CLIP** are subclasses of DEVS classes and are shown in **boldface**:

- Entities [68]
 - Models [80]
 - AtomicModel [88]
 - Dispatcher [103]
 - Equipment [108]
 - CuttingMachine [114]
 - AutomaticCutter [116]
 - BITECutter [120]
 - LaserCutter [118]
 - ManualCutter [122]
 - SpreadingMachine [130]
 - AutomaticSpreader [132]
 - Table [124]
 - CuttingTable [128]
 - SpreadingTable [126]
 - Planner [99]
 - CoupledModels [82]
 - DigraphModels [84]
 - DynamicDigraphModels [86]
 - Room [92]
 - CuttingRoom [96]
 - Warehouse [94]
- Processors [69]
 - Coordinator [73]
 - DynamicCoordinator [75]
 - RootCoordinator [77]
 - Simulators [71]

The format corresponds to that described in Appendix A.

DigraphModels subclass:

Room

A Room is a component of a plant.

Instance Variables:

equipment

layout

—the equipment in this room.

—the layout of the room—a description of the room's size and shape and the location of each piece of equipment in the room.

name

parent

processor

inport

outport

pixmap

receivers

influencees

priorityList

compositionTree

influenceDigraph

selectFn

accessing

getChildren

getInfluencees:

getReceivers:

influencees:

priorityList:

receivers:

select:

translate

DEVS

	<i>addCouple:port:connectedTo:port:</i>
	<i>buildCompositionTree:</i>
	<i>compositionTree</i>
	<i>connect:to:</i>
	<i>getChildren</i>
	<i>getInfluences:</i>
	<i>getReceivers:</i>
	<i>select:</i>
	<i>selectFn:</i>
	<i>setExtInpCouple:connect:to:</i>
	<i>setExtOutCouple:connect:to:</i>
	<i>setIntCouple:portName:to:portName:</i>
	<i>specifyChildren:</i>
	<i>translate:model:</i>
displaying	<i>state</i>
initialize-release	<i>extent:</i>
	<i>initialize</i>
	<i>initPorts</i>
port creation	<i>,</i>
printing	<i>printOn:</i>
	<i>printString</i>
private	<i>base VisualComponent</i>
	<i>inport:</i>
	<i>outport:</i>
private-accessing	<i>name:</i>
	<i>parent</i>
	<i>parent:</i>
private-logging	<i>texName</i>
views	<i>isLogging</i>
	<i>view</i>
instance creation	<i>new:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>texFormat:</i>

Room subclass:

Warehouse

A Warehouse is that portion of a plant that is responsible for receiving materials, maintaining stored items, and delivering materials to organizations both inside and outside the plant.

name

parent

processor

inport

outport

pixmap

receivers

influencees

priorityList

compositionTree

influenceDigraph

selectFn

accessing

getChildren

getInfluencees:

getReceivers:

influencees:

priorityList:

receivers:

select:

translate

DEVS

	<i>addCouple:port:connectedTo:port:</i>
	<i>buildCompositionTree:</i>
	<i>compositionTree</i>
	<i>connect:to:</i>
	<i>getChildren</i>
	<i>getInfluences:</i>
	<i>getReceivers:</i>
	<i>select:</i>
	<i>selectFn:</i>
	<i>setExtInpCoup:connect:to:</i>
	<i>setExtOutCoup:connect:to:</i>
	<i>setIntCoup:portName:to:portName:</i>
	<i>specifyChildren:</i>
	<i>translate:model:</i>
displaying	<i>state</i>
initialize-release	<i>extent:</i>
	<i>initialize</i>
	<i>initPorts</i>
port creation	<i>,</i>
printing	<i>printOn:</i>
	<i>printString</i>
private	<i>baseVisualComponent</i>
	<i>inport:</i>
	<i>outport:</i>
private-accessing	<i>name:</i>
	<i>parent</i>
	<i>parent:</i>
	<i>texName</i>
private-logging	<i>isLogging</i>
views	<i>view</i>
instance creation	<i>new:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>texFormat:</i>

Room subclass:

CuttingRoom

A CuttingRoom is a facility comprising the resources needed to cut fabric: equipment, operators, and materials. An instance is a digraph model that comprises a planner, a dispatcher for operators, a dispatcher for materials, and equipment such as spreaders and cutters.

The relationships between the various pieces of equipment must be specified explicitly. The relationships between the planner and dispatchers are defined automatically at instance creation.

Instance Variables:

<i>oscar</i>	<i>< Planner ></i>	<i>—A planner that oversees execution of a plan. The plan designates how resources are routed to equipment for various tasks. The planner component is named after Oscar Estes, the cutting room manager at Jantzen in Seneca, SC, who helped us with this project.</i>
<i>breakArea</i>	<i>< Dispatcher ></i>	<i>—A dispatcher of the operators in the cutting room. An operator is always routed to the model for the equipment to which he is next assigned. When an operator is off-duty, then he is routed back to the dispatcher.</i>
<i>dropArea</i>	<i>< Dispatcher ></i>	<i>—A dispatcher of the materials in the cutting room. A material is always routed to the model for the equipment to which it is next used.</i>
<i>equipments</i>	<i>< Set ></i>	<i>—The equipment available for use in the cutting room.</i>

Testing:

"instance creation test.."

| cuttingRoom e1 e2 equipment huey dewey louie operators materials |

e1 := Equipment makePair: 'E1'.

e2 := Equipment makePair: 'E2'.

equipment := OrderedCollection with: e1 with: e2.

huey := Operator new name: 'Huey'.
dewey := Operator new name: 'Dewey'.
louie := Operator new name: 'Louie'.
operators := Set with: *huey* with: *dewey* with: *louie*.
materials := Set with: (Material new).
cuttingRoom := CuttingRoom makePair: 'Test CR' containing: equipment
operators: *operators* *materials*: *materials*.
cuttingRoom

name
parent
processor
inport
outport
pixmap
receivers
influencees
priorityList
compositionTree
influenceDigraph
selectFn
oscar
breakArea
dropArea
equipments
plan

<i>accessing</i>	<i>breakArea</i> <i>dropArea</i> <i>equipment</i> <i>oscar</i> <i>plan:</i> <i>restart</i> <i>state</i> <i>containing:operators:materials:</i> <u><i>initPorts</i></u> <i>,</i> <i>printOn:</i> <i>printString</i> <i>base VisualComponent</i> <i>inport:</i> <i>outport:</i>
<i>DEVS</i>	
<i>displaying</i>	
<i>initialize-release</i>	
<i>port creation</i>	
<i>printing</i>	
<i>private</i>	

private-accessing

name:

parent

parent:

teXName

isLogging

private-logging

views

view

instance creation

makePair:containing:operators:materials:

logging access

isLogging

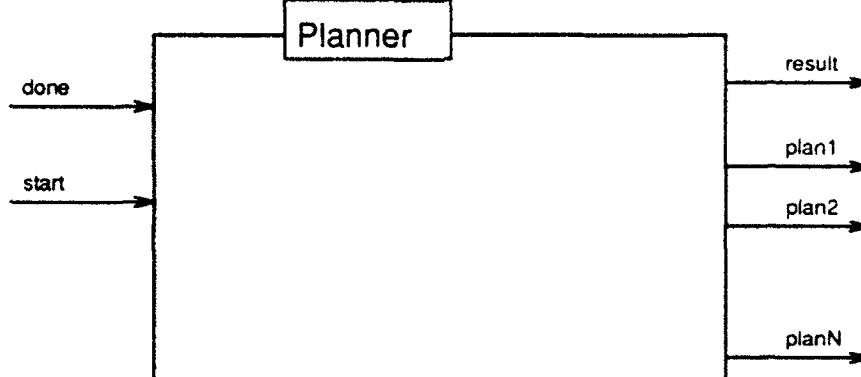
logStream:

teXFormat:

AtomicModel subclass:

Planner

A planner is a model that distributes a plan to other models. The plan is specified before simulation begins. A planner has N output ports, labeled #plan1, #plan2, ..., planN that can be used to connect to other models. The value of N is specified on instantiation and may not be changed.



A planner also tracks the completion of work assignments in the plan. A work assignment arriving on the #done port is logged as completed as of the time of its arrival. When all work assignments are completed, then the plan and statistics are emitted on the #result port.

A planner has $(N+2)$ states, numbered 0 through $(N+1)$. State 0 indicates that no plan has yet been received. State I ($1 \leq I \leq N$) indicates the plan is to be output on port #planI. State $(N+1)$ is a tracking state in which information about completed tasks arriving on the #done input port is recorded.

The output ports are represented by an array collection so that the names can be matched with states—the first element is #plan1, the second #plan2, and so on.

state variables:

sigma = infinity -- nothing happens until a plan arrives (on #start)
phase = 0 -- signifies that a plan has not yet arrived
plan = nil

external transition function:

```
case input-port
  start:      store plan
              hold in 1 for 0
  done:       record task completion
              continue
else
  error
```

internal transition function:

```
case phase
  1:          2
  2:          3
  ...
  N:          #tracking
```

output function:

```
if ( phase #tracking )
  send plan to port #plan(phase)
```

<i>plan</i>	<Plan>
<i>n</i>	<Integer>
<i>results</i>	<Statistics>
<i>port</i>	<Symbol>

Instance variables:

--the plan to be dispersed
--the number of output ports, excluding #result
--the statistics associated with the plan
--the name of the port to which the plan should be sent, nil if none.

Testing:

Planner makePair: 'Test Planner' outportCount: 3

"The following is a stand-alone test of this model (see pp. 82ff)."

"Inspect p after each of the statements below to perform the test."

| p plan wa1 wa2 wa3 |

p := Planner new: 'Planner'.

p outportCount: 3.

plan := Plan new.

wa1 := WorkAssignment new.

wa2 := WorkAssignment new.

wa3 := WorkAssignment new.

plan add: wa1; add: wa2; add: wa3.

p plan: plan.

p restart. self halt.

p intTransition.

p outputEh.

"(#plan1, plan)"

p intTransition.

"state: (sigma = 0, phase = 2, plan = plan

x = (#start, plan), y = (#plan1, plan)"

p outputEh.

"(#plan2, plan)"

p intTransition.

p outputEh.

"(#plan3, plan)"

p intTransition.

p inject: #done value: wa1 elapsedTime: 1.

p inject: #done value: wa2 elapsedTime: 2.

p inject: #done value: wa3 elapsedTime: 3.

name

parent

processor

inport

outport

pixmap

x

y

sigma

phase

e

plan

n

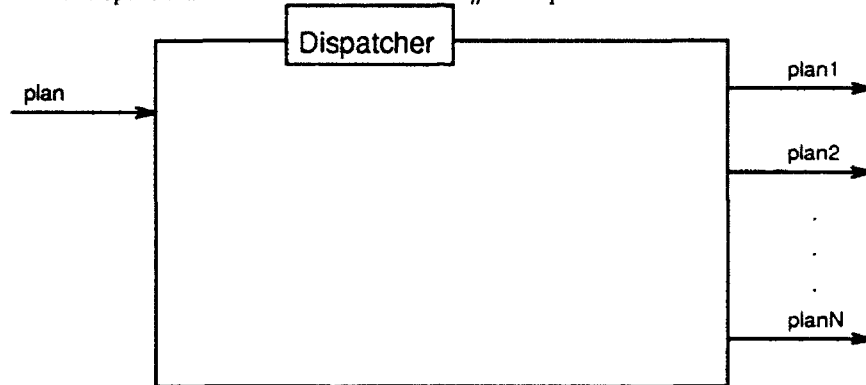
results

accessing	outportNumber:
	plan
	plan:
DEVS	statistics
	<u>extTransFn:externalInput:</u>
	<u>intTransFn</u>
	<u>outputFn</u>
displaying	<u>restart</u>
initialize-release	<u>status</u>
	<u>initPorts</u>
macros	outportCount:
	continue
	holdIn:forTime:
	inject:value:
	inject:value:elapsedTime:
	noOutput
	passivate
	passivateIn:
	send:toPort:
port creation	,
printing	printOn:
	printString
private	base VisualComponent
	inport:
	outport:
private-accessing	name:
	parent
	parent:
	teXName
private-logging	logExtTransFn
	logIntTransition:
	logOutputEh:
	logTimeAdvanceEh:
views	<u>view</u>
instance creation	makePair:outportCount:
logging access	isLogging
	logStream:
	teXFormat:

AtomicModel subclass:

Dispatcher

A Dispatcher is a model component that manages a pool of resources, dispatching a resource to another model in accordance with some plan. A dispatcher has 2 input ports, #plan and #in, and N output ports: #out1, #out2, ..., #outN. Port #plan is an input port on which a plan is to be received that drives the routing of resources to and from the dispatcher. Resources arrive on port #in and are dispatched to other models on an #out port.*



Each output port is assumed to be routed to one model. The association between model and output port is stored in a dictionary (mappingDictionary) and is established at instance creation using the models provided. For example, if the models are (m1 m2), then an association between #out1 and m1 and between #out2 and m2 is made. When a resource arrives on #in, and the plan reflects that the resource should be routed next to m1, then the resource is output on #out1 based on the dictionary association of m1 and #out1. Care must be taken to preserve correct routing in the model—that is, that connections are correct.

Every resource must be able to answer the message 'whereNext: aPlan' with the next work assignment the resource is called for. An answer of nil designates that the resource has no more uses within the plan. In determining the next model, a resource may safely assume that first work assignment for it in the plan is the next to be performed.

Instance variables:

<i>plan</i>	<i><Plan></i>	—The plan against which resources are dispatched.
<i>mappingDictionary</i>	<i><Dictionary></i>	—A dictionary whose keys are workstations and whose values are output port names. This is used to route resources to a workstation as given in the plan.
<i>resources</i>	<i><Set></i>	—A set of the resources that have not yet been dispatched, either because they have no work designated to be done or because they are not available at the current time.
<i>readyResources</i>	<i><RoutingTable></i>	—The resources that are ready for dispatching, keyed by the model that is the destination.
<i>doneResources</i>	<i><ResourceCollection></i>	Resources not needed any further in the plan

Testing:

"The following is a stand-alone test of this model (see pp. 82ff)."

"Inspect dispatcher after each of the statements below to perform the test."

```
| models resources plan wa1 wa2 wa3 o1 o2 o3 e1 e2 dispatcher |
o1 := Operator new name: 'o1'; yourself.
o2 := Operator new name: 'o2'; yourself.
o3 := Operator new name: 'o3'; yourself.
e1 := Equipment new name: 'e1'; yourself.
e2 := Equipment new name: 'e2'; yourself.
plan := Plan new.
plan startTime: (SimulationTime date: (Date today) time: (Time now)).
wa1 := WorkAssignment workstation: (Workstation at: e1) operators: (Set
with: o1) task: nil.
wa2 := WorkAssignment workstation: (Workstation at: e2) operators: (Set
with: o2) task: nil.
wa3 := WorkAssignment workstation: (Workstation at: e1) operators: (Set
with: o3) task: nil.
plan add: wa1; add: wa2; add: wa3.
models := OrderedCollection with: e1 with: e2.
resources := OrderedCollection with: o1 with: o2 with: o3.
dispatcher := Dispatcher makePair: 'Big D' forModels: models forRe-
sources: resources.
dispatcher initialize.
dispatcher inject: #plan value: plan elapsedTime: 0.
dispatcher intTransition.
dispatcher outputEh.
dispatcher intTransition.
dispatcher outputEh.
dispatcher
dispatcher inject: #in value: plan elapsedTime: 0.
```

name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e
plan
mappingDictionary
resources
readyResources
doneResources
clock

accessing

DEVS

displaying
initialize-release

macros

operating

port creation
printing
private

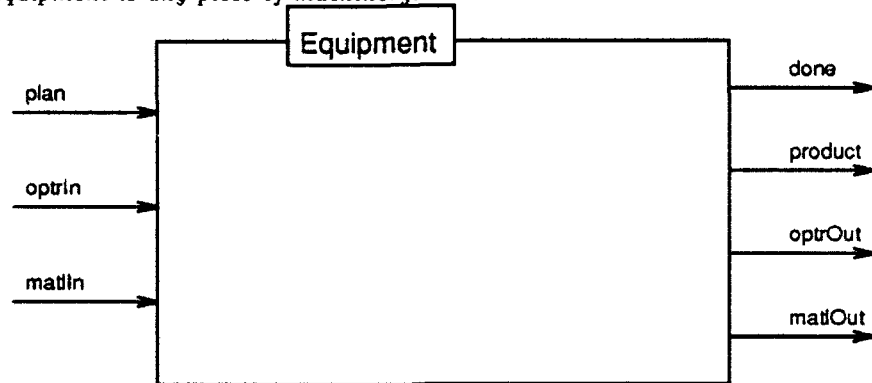
doneResources
outputNumber:
portFor:
readyResources
resources
extTransFn:externalInput:
intTransFn
outputFn
restart
status
forModels:forResources:
initPorts
continue
holdIn:forTime:
inject:value:
inject:value:elapsedTime:
noOutput
passivate
passivateIn:
send:toPort:
checkResources
wakeup
,
printOn:
base VisualComponent
inport:
outport:

private-accessing	name:
	parent
	parent:
private-logging	texName
	logExtTransFn
	logIntTransition:
	logOutputEh:
	logTimeAdvanceEh:
views	<u>view</u>
instance creation	forModels:forResources:
	makePair:forModels:forResources:
logging access	isLogging
	logStream:
	texFormat:

AtomicModel subclass:

Equipment

Equipment is any piece of machinery.



Equipment has behavior such that if it has a work assignment in the plan that is still "to do", and sufficient resources and operators are available to perform the work assignment, and the workstation for the assignment is available, then work on the task is begun. Upon completion of a work assignment, statistics are emitted on the #done port, the product is emitted on the #product port, the operator(s) who performed the task are emitted on the #optrOut port, and the collection of materials (what's left of them) is emitted on the #matlOut port.

Every piece of equipment comprises a set of workstations. For this class, only one workstation can be active at a time. (See subclasses for equipment that can support multiple active workstations.)

Each piece of equipment has a wait area at which idle operators and materials wait for the start of a task specified in a work assignment. The wait area is represented by instance variables 'waitingOperators' and 'waitingMaterials'.

When a resource arrives at equipment, it is put in the wait area and then a check is made to see if a work assignment can be started based on the presence of all required resources in the wait area and the availability of the workstation specified. The current implementation insists that work assignments be started in their order in the plan. When a work assignment is started, eligible operators not selected for the task are emitted on the #optrOut port. [A work assignment may specify a task to be performed by any one of a set of operators.]

Equipment has the following phases and transitions:

#passive =>

```
#working =>  
#done => [emit statistics]  
#product => [emit product ]  
#optrOut => [emit busyOperators ]  
#matlOut => [emit busyMaterials ]  
(back to #passive or #working)
```

Instance Variables:

<i>location</i>	—the location of the equipment on the shop floor.
<i>name</i>	—the descriptive name for this equipment.
<i>number</i>	—the identification number assigned to this equipment.
<i>status</i>	—an indication of whether the equipment is operable or available for work or not.
<i>manufacturer</i>	—the equipment manufacturer.
<i>serialNumber</i>	—the serial number for this piece of equipment.
(Simulation-related entities:)	
<i>clock</i>	—the (global) simulation time
<i>plan</i>	—the current plan under simulation (a shallow copy of the original)
<i>toDo</i>	—the portion of the plan for this piece of equipment that is not yet started.
<i>startClock</i>	—the value of clock when work on the current task started.
<i>currentWorkAssignment</i>	—the current work assignment
<i>busyWorkstations</i>	—the workstations currently active for this piece of equipment. This is a sorted collection, ordered by increasing time to next work assignment completion. An instance will have at most one busy workstation, but subclasses might have more.
<i>busyOperators</i>	—a collection of operators busy at this equipment.
<i>busyMaterials</i>	—a collection of materials in use at this equipment.
<i>waitingOperators</i>	—a collection of operators waiting for workstation availability.
<i>waitingMaterials</i>	—a collection of materials waiting for workstation availability.
<i>product</i>	—the product of the most recently completed task.
<i>statistics</i>	—statistics for the most recently completed work assignment

The methods in category *DEVS* default to model a piece of equipment that takes 5 minutes to do any task. The product is a string of the form 'Product <time>', where <time> designates the simulation time at which the product was completed.

Testing:

```
| plan wa1 wa2 wa3 o1 o2 o3 equipment |
o1 := Operator new name: 'o1'; yourself.
o2 := Operator new name: 'o2'; yourself.
o3 := Operator new name: 'o3'; yourself.
equipment := Equipment new name: 'E'; yourself.
plan := Plan new.
plan startTime: (SimulationTime date: (Date today) time: (Time now)).
wa1 := WorkAssignment workstation: (Workstation at: equipment) oper-
ators: (Set with: o1) task: nil.
wa2 := WorkAssignment workstation: (Workstation at: equipment) oper-
ators: (Set with: o2) task: nil.
wa3 := WorkAssignment workstation: (Workstation at: equipment) oper-
ators: (Set with: o3) task: nil.
plan add: wa1; add: wa2; add: wa3.
equipment restart.
equipment inject: #plan value: plan elapsedTime: 0.
equipment inject: #optrIn value: o1 elapsedTime: 0.
equipment intTransition.
equipment outputEh.
equipment intTransition.
equipment outputEh.
equipment
equipment inject: #in value: plan elapsedTime: 0.
```

name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e
 location
 description
 number
 status
 manufacturer
 serialNumber
 clock
 plan
 startClock
 toDo
 currentWorkAssignment
 busyWorkstations
 waitingOperators
 waitingMaterials
 product
statistics

accessing

DEVS

displaying
 initialize-release

busyWorkstations
 description
 description:
 location
 manufacturer
 moveTo:
 number
 serialNumber
 waitingMaterials
 waitingOperators
extTransFn:externalInput:
intTransFn
outputFn
restart
 status
initPorts

macros	<i>continue</i> <i>holdIn:forTime:</i> <i>inject:value:</i> <i>inject:value:elapsedTime:</i> <i>noOutput</i> <i>passivate</i> <i>passivateIn:</i> <i>send:toPort:</i>
port creation	,
printing	<u><i>printOn:</i></u>
private	<i>check</i> <i>plan:</i> <i>start:on:by:using:</i>
private-accessing	<i>name:</i> <i>parent</i> <i>parent:</i>
private-logging	<i>teXName</i> <i>logExtTransFn</i> <i>logIntTransition:</i> <i>logOutputEh:</i> <i>logTimeAdvanceEh:</i>
views	<u><i>view</i></u>
instance creation	<i>makePair:</i>
logging access	<i>new:</i> <i>isLogging</i> <i>logStream:</i> <i>teXFormat:</i>

Equipment subclass:

CuttingMachine

A CuttingMachine is a machine used to cut fabric.

Instance Variables:

<i>fabrics</i>	<i>—the fabrics that can be cut by this machine.</i>
<i>maximumCuttingDepth</i>	<i>—the maximum cutting depth</i>

Class Variables:

name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e
location
description
number
status
manufacturer
serialNumber
clock
plan
startClock
toDo
currentWorkAssignment
busyWorkstations
waitingOperators
waitingMaterials
product
statistics
fabrics
maximumCuttingDepth

accessing	fabrics
	fabrics:
	maximumCuttingDepth
	maximumCuttingDepth:
DEVS	extTransFn:externalInput:
	intTransFn
	outputFn
	restart
displaying	status
initialize-release	initPorts
macros	continue
	holdIn:forTime:
	inject:value:
	inject:value:elapsedTime:
	noOutput
	passivate
	passivateIn:
	send:toPort:
port creation	,
printing	printOn:
private	check
	plan:
	start:on:by:using:
private-accessing	name:
	parent
	parent:
	texName
private-logging	logExtTransFn
	logIntTransition:
	logOutputEh:
	logTimeAdvanceEh:
views	view
instance creation	makePair:
	new:
logging access	isLogging
	logStream:
	texFormat:

CuttingMachine subclass:

AutomaticCutter

An AutomaticCutter is a cutter whose blade is under computer control.

name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e
location
description
number
status
manufacturer
serialNumber
clock
plan
startClock
toDo
currentWorkAssignment
busyWorkstations
waitingOperators
waitingMaterials
product
statistics
fabrics
mazimumCuttingDepth

accessing

fabrics
fabrics:
mazimumCuttingDepth
mazimumCuttingDepth:

DEVS	<i>extTransFn:externalInput:</i>
	<i>intTransFn</i>
	<i>outputFn</i>
displaying	<i>restart</i>
initialize-release	<i>status</i>
macros	<i>initPorts</i>
	<i>continue</i>
	<i>holdIn:forTime:</i>
	<i>inject:value:</i>
	<i>inject:value:elapsedTime:</i>
	<i>noOutput</i>
	<i>passivate</i>
	<i>passivateIn:</i>
	<i>send:toPort:</i>
port creation	<i>,</i>
printing	<i>printOn:</i>
private	<i>check</i>
	<i>plan:</i>
private-accessing	<i>start:on:by:using:</i>
	<i>name:</i>
	<i>parent</i>
	<i>parent:</i>
private-logging	<i>teXName</i>
	<i>logExtTransFn</i>
	<i>logIntTransition:</i>
	<i>logOutputEh:</i>
	<i>logTimeAdvanceEh:</i>
views	<i>view</i>
instance creation	<i>makePair:</i>
	<i>new:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>teXFormat:</i>

AutomaticCutter subclass:

LaserCutter

A LaserCutter is a cutter in which a laser beam is used to cut fabric.

*name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e
location
description
number
status
manufacturer
serialNumber
clock
plan
startClock
toDo
currentWorkAssignment
busyWorkstations
waitingOperators
waitingMaterials
product
statistics
fabrics
mazimumCuttingDepth*

accessing

*fabrics
fabrics:
mazimumCuttingDepth
mazimumCuttingDepth:*

DEVS	<i>extTransFn:externalInput:</i>
	<i>intTransFn</i>
	<i>outputFn</i>
displaying	<i>restart</i>
initialize-release	<i>status</i>
macros	<i>initPorts</i>
	<i>continue</i>
	<i>holdIn:forTime:</i>
	<i>inject:value:</i>
	<i>inject:value:elapsedTime:</i>
	<i>noOutput</i>
	<i>passivate</i>
	<i>passivateIn:</i>
	<i>send:toPort:</i>
port creation	<i>,</i>
printing	<i>printOn:</i>
private	<i>check</i>
	<i>plan:</i>
private-accessing	<i>start:on:by:using:</i>
	<i>name:</i>
	<i>parent</i>
	<i>parent:</i>
private-logging	<i>teXName</i>
	<i>logExtTransFn</i>
	<i>logIntTransition:</i>
	<i>logOutputEh:</i>
	<i>logTimeAdvanceEh:</i>
views	<i>view</i>
instance creation	<i>makePair:</i>
	<i>new:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>teXFormat:</i>

AutomaticCutter subclass:

BITECutter

A BITECutter is an automatic cutter that has a cutting blade mounted on a mechanism that allows the blade to be moved to any point in an X-Y coordinate system. The blade is controlled by digital data that represents a marker. The length of a spread to be cut may be longer than the table for the cutter.

*name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e
location
description
number
status
manufacturer
serialNumber
clock
plan
startClock
toDo
currentWorkAssignment
busyWorkstations
waitingOperators
waitingMaterials
product
statistics
fabrics
maximumCuttingDepth*

accessing	<i>fabrics</i>
	<i>fabrics:</i>
	<i>maximumCuttingDepth</i>
DEVS	<i>maximumCuttingDepth:</i>
	<i>extTransFn:externalInput:</i>
	<i>intTransFn</i>
	<i>outputFn</i>
displaying	<i>restart</i>
initialize-release	<i>status</i>
macros	<i>initPorts</i>
	<i>continue</i>
	<i>holdIn:forTime:</i>
	<i>inject:value:</i>
	<i>inject:value:elapsedTime:</i>
	<i>noOutput</i>
	<i>passivate</i>
	<i>passivateIn:</i>
	<i>send:toPort:</i>
port creation	<i>'</i>
printing	<i>printOn:</i>
private	<i>check</i>
	<i>plan:</i>
	<i>start:on:by:using:</i>
private-accessing	<i>name:</i>
	<i>parent</i>
	<i>parent:</i>
private-logging	<i>teXName</i>
	<i>logExtTransFn</i>
	<i>logIntTransition:</i>
	<i>logOutputEh:</i>
	<i>logTimeAdvanceEh:</i>
views	<i>view</i>
instance creation	<i>makePair:</i>
	<i>new:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>teXFormat:</i>

CuttingMachine subclass:

ManualCutter

A ManualCutter is a cutter whose path is controlled completely by an operator.

name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e
location
description
number
status
manufacturer
serialNumber
clock
plan
startClock
toDo
currentWorkAssignment
busyWorkstations
waitingOperators
waitingMaterials
product
statistics
fabrics
mazimumCuttingDepth

accessing

fabrics
fabrics:
mazimumCuttingDepth
mazimumCuttingDepth:

DEVS	<i>extTransFn:externalInput:</i>
	<i>intTransFn</i>
	<i>outputFn</i>
displaying	<i>restart</i>
initialize-release	<i>status</i>
macros	<i>initPorts</i>
	<i>continue</i>
	<i>holdIn:forTime:</i>
	<i>inject:value:</i>
	<i>inject:value:elapsedTime:</i>
	<i>noOutput</i>
	<i>passivate</i>
	<i>passivateIn:</i>
	<i>send:toPort:</i>
port creation	<i>,</i>
printing	<i>printOn:</i>
private	<i>check</i>
	<i>plan:</i>
	<i>start:on:by:using:</i>
private-accessing	<i>name:</i>
	<i>parent</i>
	<i>parent:</i>
private-logging	<i>teXName</i>
	<i>logExtTransFn</i>
	<i>logIntTransition:</i>
	<i>logOutputEh:</i>
	<i>logTimeAdvanceEh:</i>
views	<i>view</i>
instance creation	<i>makePair:</i>
	<i>new:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>teXFormat:</i>

Equipment subclass:

Table

A Table is a raised, flat, rectangular surface.

Instance Variables:

<i>height</i>	<i>—the height of this table.</i>
<i>length</i>	<i>—the length of this table.</i>
<i>width</i>	<i>—the width of this table.</i>

Class Variables:

location
description
number
status
manufacturer
serialNumber
clock
plan
startClock
toDo
currentWorkAssignment
busyWorkstations
waitingOperators
waitingMaterials
product
statistics
height
length
width

<i>accessing</i>	<i>height</i>
	<i>length</i>
	<i>width</i>
<i>DEVS</i>	<i>extTransFn:externalInput:</i>
	<i>intTransFn</i>
	<i>outputFn</i>
	<i>restart</i>
<i>initialize-release</i>	<i>initPorts</i>
<i>printing</i>	<i>printOn:</i>

private

check

plan:

start: on: by: using:

views

view

instance creation

height: width: height:

Table subclass:

SpreadingTable

A SpreadingTable is a table on which fabric is spread, either manually or by using an automatic spreader.

Instance Variables:

Class Variables:

location
description
number
status
manufacturer
serialNumber
clock
plan
startClock
toDo
currentWorkAssignment
busyWorkstations
waitingOperators
waitingMaterials
product
statistics
height
length
width

accessing

DEVS

initialize-release

printing

private

views

height
length
width
extTransFn:externalInput:
intTransFn
outputFn
restart
initPorts
printOn:
check
plan:
start: on: by: using:
view

instance creation	<i>height:width:height:</i>

Table subclass:

CuttingTable

A CuttingTable is a table on which fabric is cut, either manually or automatically.

Instance Variables:

cutters

*--the collection of cutters that
can be used at this table.*

Class Variables:

location

description

number

status

manufacturer

serialNumber

clock

plan

startClock

toDo

currentWorkAssignment

busyWorkstations

waitingOperators

waitingMaterials

product

statistics

height

length

width

cutters

accessing

height

length

width

DEVS

extTransFn:externalInput:

intTransFn

outputFn

initialize-release

restart

initPorts

printing

printOn:

private

check

plan:

TBS

start: on: by: using:

cutters

views

cutters:

view

instance creation

height: width: height:

Equipment subclass:

SpreadingMachine

A spreading machine is a piece of equipment used for spreading. The model has two

Instance Variables:

mountedRolls

—a dictionary of the mounted rolls. The key is some designation of a mount.

weightLimits

—the maximum weight that a roll mount can support, in pounds.

sizeLimits

—the maximum roll diameter that a roll mount can hold.

The last two attributes are ignored in this implementation.

Class Variables:

location

description

number

status

manufacturer

serialNumber

clock

plan

startClock

toDo

currentWorkAssignment

busyWorkstations

waitingOperators

waitingMaterials

product

statistics

mountSymbols

mountedRolls

weightLimits

sizeLimits

spreadRates

returnRate

accessing	mountSymbols
	returnRate
	returnRate:
DEVS	spreadRates:
	<i>extTransFn:externalInput:</i>
	<i>intTransFn</i>
	<i>outputFn</i>
initialization-release	<i>restart</i>
initialize-release	initialize:weightLimits:sizeLimits:spreadRates:
operating	<i>initPorts</i>
	clearOut
	mount:
	mount:on:
	remove:
	removeRoll:
	respondTo:
printing	spreadRateFor:
private	<i>printOn:</i>
	<i>check</i>
	<i>plan:</i>
	<i>start:on:by:using:</i>
views	<i>view</i>
instance creation	new:
	new:spreadRates:
	new:weightLimits:sizeLimits:spreadRates:

SpreadingMachine subclass:

AutomaticSpreader

An AutomaticSpreader is a machine that can be attached to a spreading table and used to spread fabric. Fabric (in rolls or other form) are attached to the spreader. The spreader then spreads the fabric in a single ply.

Instance Variables:

Class Variables:

location
description
number
status
manufacturer
serialNumber
clock
plan
startClock
toDo
currentWorkAssignment
busyWorkstations
waitingOperators
waitingMaterials
product
statistics
mountSymbols
mountedRolls
weightLimits
sizeLimits
spreadRates
returnRate

accessing

mountSymbols

returnRate

returnRate:

spreadRates:

extTransFn:externalInput:

intTransFn

outputFn

restart

DEVS

initialize:weightLimits:sizeLimits:spreadRates:

initialization-release

initialize-release
operating

printing
private

views

initPorts
clearOut
mount:
mount:on:
remove:
removeRoll:
respondTo:
spreadRateFor:
printOn:
check
plan:
start:on:by:using:
view

instance creation

new:
new:spreadRates:
new:weightLimits:sizeLimits:spreadRates:

Appendix C

Test Class Specifications

The classes whose specifications appear in this appendix are used for testing. They correspond to the generator, transducer, and simple processor models described by Zeigler. Not only are these classes useful for testing, they are also useful as a model for writing application classes.

The classes in the testing component of **CZIP** are subclasses of DEVS classes and are shown in **boldface**:

- Entities [68]
 - Models [80]
 - AtomicModel [88]
 - GeneratorModels** [135]
 - SimpleProcessorModel** [139]
 - TransducerModels** [137]
 - CoupledModels [82]
 - DigraphModels [84]
 - Processors [69]
 - Coordinator [73]
 - RootCoordinator [77]
 - Simulators [71]

The format corresponds to that described in Appendix A.

The end of this appendix provides text that can be used to run a simulation (see Section 3.5).

AtomicModel subclass:

GeneratorModels

This class implements the model described by Zeigler in section 5.1.1.

Instance variables:

<i>nextJobNumber</i>	—
<i>interarrivalTime</i>	—

name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e
nextJobNumber
interarrivalTime

<i>accessing</i>	<i>interarrivalTime</i>
DEVS	<i>interarrivalTime:</i> <u><i>extTransFn:externalInput:</i></u> <u><i>intTransFn</i></u> <u><i>outputFn</i></u> <u><i>restart</i></u> <i>status</i> <u><i>initPorts</i></u> <i>continue</i> <i>holdIn:forTime:</i> <i>inject:value:</i> <i>inject:value:elapsedTime:</i> <i>noOutput</i> <i>passivate</i> <i>passivateIn:</i> <i>send:toPort:</i> ,
<i>displaying</i>	<i>printOn:</i>
<i>initialize-release</i>	<i>printString</i>
<i>macros</i>	
<i>port creation</i>	
<i>printing</i>	

private	<i>base VisualComponent</i>
	<i>inport:</i>
	<i>outport:</i>
private-accessing	<i>nextJobName</i>
	<i>nextJobNumber</i>
private-logging	<i>nextJobNumber:</i>
	<i>logExtTransFn</i>
	<i>logIntTransition:</i>
	<i>logOutputEh:</i>
	<i>logTimeAdvanceEh:</i>
views	<i>view</i>

instance creation	<i>makePair:interarrivalTime:</i>
	<i><u>new:</u></i>
	<i>new:interarrivalTime:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>teXFormat:</i>

AtomicModel subclass:

TransducerModels

This class implements the models described by Zeigler in section 5.1.2.

Instance variables:

<i>observationInterval</i>	—
<i>arrivedList</i>	—
<i>solvedList</i>	—
<i>clock</i>	<Duration> — Total time elapsed in the run
<i>totalTa</i>	—

name

parent

processor

inport

outport

pixmap

x

y

sigma

phase

e

observationInterval

arrivedList

solvedList

clock

totalTa

accessing

arrivedList

arrivedList:

clock

clock:

observationInterval:

solvedList

solvedList:

totalTa

totalTa:

extTransFn:externalInput:

intTransFn

outputFn

restart

DEVS

displaying	<i>status</i>
initialize-release	<i><u>initPorts</u></i>
macros	<i>continue</i>
	<i>holdIn:forTime:</i>
	<i>inject:value:</i>
	<i>inject:value:elapsedTime:</i>
	<i>noOutput</i>
	<i>passivate</i>
	<i>passivateIn:</i>
	<i>send:toPort:</i>
port creation	<i>,</i>
printing	<i>printOn:</i>
	<i>printString</i>
private	<i>baseVisualComponent</i>
	<i>inport:</i>
	<i>outport:</i>
private-accessing	<i>name:</i>
	<i>parent</i>
	<i>parent:</i>
	<i>teXName</i>
private-logging	<i>logEztTransFn</i>
	<i>logIntTransition:</i>
	<i>logOutputEh:</i>
	<i>logTimeAdvanceEh:</i>
views	<i><u>view</u></i>
instance creation	<i>makePair:observationInterval:</i>
	<i><u>new:</u></i>
	<i>new:observationInterval:</i>
logging access	<i>isLogging</i>
	<i>logStream:</i>
	<i>teXFormat:</i>

AtomicModel subclass:

SimpleProcessorModel

This class implements the model described by Zeigler in section 4.2.

Instance variables:

jobID —
processingTime —

Testing:

This test is taken from Zeigler, p. 82ff. State is listed as: (sigma phase job-id processing-time)

| p |
p := SimpleProcessorModel makePair: 'P' processingTime: 5 minutes.
p restart.

"a" p x: (Content port: (p , #in) value: #x1).
"b" p e: 0 minutes.
"c" p extTransition. "State: (5 #busy #x1 5)"
"d" p outputEh.
"e" p intTransition. "State: (INF #passive #x1 5)"
p.

"i" p inject: (p , #in) value: #x1 elapsedTime: 0 minutes.
"j" p inject: (p , #in) value: #x2 elapsedTime: 3 minutes.
"k" p outputEh.
"l" p intTransition. "State: (INF #passive #x1 5)"

name
parent
processor
inport
outport
pixmap
x
y
sigma
phase
e
jobID
processingTime

accessing	jobID
	jobID:
	processingTime
	processingTime:
DEVS	<u>extTransFn:externalInput:</u>
	<u>intTransFn</u>
	<u>outputFn</u>
	<u>restart</u>
displaying	status
initialize-release	<u>initPorts</u>
macros	continue
	holdIn:forTime:
	inject:value:
	inject:value:elapsedTime:
	noOutput
	passivate
	passivateIn:
	send:toPort:
port creation	,
printing	printOn:
	printString
private	baseVisualComponent
	inport:
	outport:
private-accessing	name:
	parent
	parent:
	teXName
private-logging	logExtTransFn
	logIntTransition:
	logOutputEh:
	logTimeAdvanceEh:
views	<u>view</u>
instance creation	makePair:processingTime:
	<u>new:</u>
	new:processingTime:
logging access	isLogging
	logStream:
	teXFormat:

This is the text of file "simple test."

```
{\tt "}"simple test{\tt "}"
| genr transd p ef efp r|
```

```
{\tt "}"Processors texFormat: true.{\tt "}"
Processors logStream: 'simp.log' asFilename writeStream.
{\tt "}"Processors logStream: nil.{\tt "}" {\tt "}"Close the log stream--later!!{\tt "}"
```

```
genr := GeneratorModels makePair: 'GENR' interarrivalTime: 10 minutes.
transd := TransducerModels makePair: 'TRANSD' observationInterval: 100 minutes.
p := SimpleProcessorModel makePair: 'P' processingTime: 5 minutes.
```

```
ef := (DigraphModels makePair: 'EF').
    ef
        inport: #(in);
        outport: #(result out);
        specifyChildren: ((OrderedCollection new) add: genr; add: transd; yourself);
        {\tt "}"External input coupling..{\tt "}"
        addCouple: ef port: #in connectedTo: transd port: #solved;
        {\tt "}"External output coupling..{\tt "}"
        addCouple: genr port: #out connectedTo: ef port: #out;
        addCouple: transd port: #out connectedTo: ef port: #result;
        {\tt "}"Internal coupling..{\tt "}"
        addCouple: genr port: #out connectedTo: transd port: #ariv;
        addCouple: transd port: #out connectedTo: genr port: #stop.
```

```
efp := (DigraphModels makePair: 'EF-P').
    efp
        inport: #();
        outport: #(out);
        specifyChildren: ((OrderedCollection new) add: p; add: ef; yourself);
        priorityList: ( (OrderedCollection new)
            addFirst: p;
            addLast: ef; yourself );
        {\tt "}"External input coupling..{\tt "}"
        {\tt "}"External output coupling..{\tt "}"
        addCouple: ef port: #result connectedTo: efp port: #out;
        {\tt "}"Internal coupling..{\tt "}"
        addCouple: p port: #out connectedTo: ef port: #in;
        addCouple: ef port: #out connectedTo: p port: #in.
```

```
r := (RootCoordinator new: 'RC').
r
```

```
startTime: (SimulationTime date: (Date today) time: (Time fromSeconds: 0));  
timeLimit: r startTime + 110 minutes;  
linkToParent: (efp processor).  
SimulationRunView tryOn: r.
```

Bibliography

- [RG89] Mary Beth Rosson and Eric Gold. Problem- solution mapping in object-oriented design. In *OOPSLA '89 Proceedings*, 1989.
- [Weg87] Peter Wegner. Dimensions of object-based language design. In *OOPSLA '87 Proceedings*, New York, 1987. ACM.
- [Zei90] Bernard P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. Academic Press, 1990.