Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

USING GRAPHS TO TRANSLATE BETWEEN WORLD VIEWS

C. Michael Overstreet
Computer Science Department
Old Dominion University
Norfolk, VA 23529-0162

ABSTRACT

The standard world views for discrete event simulation of *event scheduling, activity scanning,* and *process interaction* are each used for model implementation, each supported by one or more simulation programming languages. We present transformations of a model representation in one form into model representations into each of these world views. This transformation process is interesting in that 1) it requires a characterization of each of these world views and 2) it demonstrates the potential of each world view to simplify a model specification.

1. INTRODUCTION

The world views offered by different simulation programming languages such as SIMULA (Dahl 1966), (Birtwistle 1973), SIMSCRIPT (Markowitz, Hausner and Karr 1962), (C.A.C.I 1983), GPSS (Schriber 1974), (IBM 1971), (Henriksen and Crain 1983), and ECSL (Buxton and Laski 1963), (Clementson 1966) provide alternative approaches for creating an executable model, that is, a program, which can be used to enhance understanding about a real or imagined system. We present an informal characterization of the three world views described by Lackner (1962, 1964) and Kiviat (1969) and then present transformations of a model specification in the form of a condition specification (discussed below) into each of these world views. These world views are usually called *event scheduling, activity scanning,* and *process interaction.* See, for example, Fishman (1973, pp. 24-25), Zeigler (1976, p. 144), Gordon (1978, pp. 289-290), and Hooper (1986, p. 153).

Section 2 provides an informal characterization of the three discrete event simulation world views, and Section 3 gives an overview of *conditions specifications*, the starting point for each of the transformations, along with an example which is used to illustrate the transformations. Section 4 presents transformations from a condition specification into each world view. Section 5 is a summary.

2. WORLD VIEW CHARACTERISATION

Our understanding and characterization of these world views is most directly influenced by Fishman and Nance. What we describe is an extension of their characterizations. Directly relevant pieces of their work are quoted here.

Fishman gives the following definitions and brief discussion of world views.

> The concepts of *event, process,* and *activity* are especially important when building a model of a system. As already defined, an *event* signifies a change in state of an entity. A *process* is a sequence of events ordered on time. An *activity* is a collection of operations that transform the state of an entity. (Fishman 1973, p. 24)
>
> These three concepts give rise to three alternative ways of building discrete event models [Fishman here references (Kiviat 1967)]. The *event scheduling approach* emphasizes a detailed description of the steps that occur when an individual event takes place. Each type of event naturally has a distinct set of steps associated with it. The *activity scanning approach* emphasizes a review of all activities in a simulation to determine which can be begun or terminated each time an event occurs. The *process interaction approach* emphasizes the progress of an entity through a system from its arrival event to its departure event. The development of the three concepts is related to the development of discrete event computer simulation programming languages. In particular, SIMSCRIPT and GASP use the event scheduling approach; GPSS and SIMULA, the process interaction approach; and CSL, the activity scanning approach (Fishman 1973, p. 25).

Nance provides a careful characterization of the concepts underlying these world views. These definitions have influenced the world view formulation presented below.

> — An *activity* is the state of an object over an interval.
>
> — An *event* is a change in object state, occurring at an instant, that initiates an activity precluded prior to that instant.
>
> An event is *determined* if the only condition on the event occurrence can be expressed strictly as a function of system time. Otherwise, the event is *contingent.*
>
> — An *object activity* is the state of an object between the events describing successive state changes for the object.
>
> — A *process* is the succession of states of an object over a span (or the contiguous succession of one or more object activities). (Nance 1981, p. 176)

The activity scanning approach is little used in the United States. For example, while Fishman discusses three world views in his 1973 text (quoted above), his later 1978 text presents only two: event scheduling and process interaction (Fishman 1978). Likewise in their 1982 text, Law and Kelton only mention event scheduling and process interaction (Law and Kelton 1982). While Banks and Carson describe three world views, they substitute "continuous" for activity scanning. (Banks and Carson 1985, p. 225). Activity scanning has wider acceptance in the United Kingdom, providing the basis for languages such as SIMON (Hills 1967), CAPS (Hutchinson 1975), and the various DRAFT systems (Mathewson 1976, 1977a, 1977b, 1984, 1985).

The current version SIMSCRIPT no longer emphasizes the event scheduling world view but has joined what seems to be, in the United States at least, a process interaction band wagon. Indeed, in Russell's tutorial text published by CACI, the vendor of the SIMSCRIPT II.5 compiler, *event* is defined as "a pending (re)activation of a process" (Russell 1983, p. 2-6) and his examples all use PROCESS rather than EVENT routines (except one example included explicitly to illustrate the syntax of EVENT routines).

In spite of the influential work of Zeigler (1984), which provides formal definitions of these world views from a general systems theory perspective, no generally accepted definitions exist. Instead, each provides an alternate approach to organizing a specification of model behavior. This, rather than a formal, mathematical formulation, is what is described below.

Based on what seems to be the essence of the model decomposition approach encouraged by the different world views, we present an informal characterization of each world view. We characterize each world view as emphasizing a different type of *locality*. Locality, as defined by Weinberg, is "that property when all relevant parts of a program are found in the same place" (Weinberg 1971, p. 229). This definition is appealing since it emphasizes, perhaps unintentionally, that the property of locality is problem dependent. Different pieces of a program are required to understand different aspects of the program, that is, different pieces are relevant to different problems.

Each world view emphasizes a different type of locality.

- **Event scheduling** emphasizes locality of *time*. Each event routine in a model specification describes a collection of actions which may normally all occur in one instant, i.e., with no time advance.
- **Activity scanning** emphasizes locality of *state*. Each activity routine in a model specification describes a collection of actions which will occur once the model achieves a specified state. These resulting actions may occur at different values of time, but once the state is achieved, they must all occur.
- **Process interaction** emphasizes locality of *object*. Each process routine in a model specification describes all actions taken by one model object (or, more properly, one class of model objects). The description of each object may incorporate both event and activity orientations.

Each world view is illustrated below.

3. CONDITION SPECIFICATION AND AN EXAMPLE

To illustrate how each world view provides alternate organizations of the same information, we will describe transformations of a model specification into each of these world views. The language used here for model specification is described in Overstreet and Nance (1985) but we present a brief description and example here.

A discrete event model specification in the form of a *condition specification* (CS) consists of three components:

1. An *interface specification* which identifies all model inputs and outputs.
2. A specification of model dynamics, consisting of
   a. a set of object specifications, defining the attributes comprising each class of objects in the model.
   b. a set of *action clusters* (ACs). These are discussed below.
3. A *report specification* of the data that are to be produced, including how results are to be computed.

In the object specification, one attribute may be associated with several model objects. These are similar to Nance's *relational attributes* (Nance 1981, p. 26). The object attribute association must be provided by the modeler.

Each action cluster consists of a condition and a list of model actions. The ACs have simple semantics. The actions associated with each condition are to occur whenever, and as long as, the condition is true. Possible actions include the standard things done in a simulation programming language such as computation and assignment of new attribute values, input/output of attribute values, generation and elimination of model objects.

Three constructs are included for time management: *Set Alarm, When Alarm,* and *After Alarm.* While modeled on Dijkstra's semaphores, they have different semantics. The *Set Alarm* operation causes a specified alarm (an attribute with type "time-based signal") to signal at a specified time. *When Alarm* and *After Alarm* are similar in their function. Each is a Boolean-valued component of an AC and name a time-based signal as an argument; each detects a specified alarm. The *When Alarm* will test true only at an instant the time-based signal is set to occur; the *After Alarm* will test true at the instant the alarm is scheduled and will continue to test true until the entire Boolean expression of which it is a component tests true.

A sample specification is given below. We also include an informal model description and a study objective. Since our interest here is model transformation, we have omitted the report specification; examples of report specifications can be found in Overstreet (1982).

Informal Model Description:

System Description: Parts arrive for processing by one of three identical machines. A machine is selected at random from the set of available machines. Processing is sometimes interrupted due to machine failure. Parts that have their processing interrupted will be finished once their machine resumes operation. Interarrival times, machine processing times, and inter-machine-failure times are negative exponential.

Model Objective: Estimate the amount of time required to process a fixed number of parts.

CS Specification

Interface and object specifications are presented in Tables 1 and 2 respectively, actions clusters in Table 3. The syntax used for model actions is similar to that of Pascal. A comment precedes each action cluster and includes a two letter identifier which is used in the graphs presented below.

Condition specifications also can be used to provide a basis for model analysis to assist in model formulation and model implementation. See Nance and Overstreet (1987, 1986), Bauer, Kochar and Talavage (1985), Moose and Nance (1983), and Overstreet (1982) for illustrations of the types of analysis which have been developed.

4. WORLD VIEW TRANSFORMATIONS

We now describe transformations of conditions specifications into a model specification in any of our three world views. Each transformation involves a similar process of generating a collection of specification components which describe sequences of model actions and which demonstrate locality of time, state, or object. This is achieved in a three step process. First valid sequences of model actions are identified. These are then grouped into event,

Table 1: Interface Specification for Condition Specification

Name	Description	Type
Inputs:		
PartInterarrivalTime	Average time between part arrivals	positive real
ProcessingTime	Cycle time required for a machine to process a part	positive real
MachineRepairTime	Average time to repair a failed machine	positive real
MachineUpTime	Average time until a repaired machine fails again	positive real
NumPartsToProcess	Number of parts processed before simulation terminates	positive integer
Outputs:		
SystemTime	The time at which the last part completes processing	positive real

Table 2: Object Specification for Condition Specification

Object	Attribute	Type
System	Initialization	Time-based signal
	SystemTime	Nonnegative real
	NumPartsToProcess	Nonnegative integer
Machine[1 .. 3]	Initialization	Time-based signal
	Status	{ busy, idle, failed }
	FailedStatus	{ busy, idle }
	RemProcessingTime	Nonnegative integer
	EndService	Time-based signal
	EndRepair	Time-based signal
	MachineFailure	Time-based signal
	MachineUpTime	Positive real
	MachineRepairTime	Positive real
	ProcessingTime	Nonnegative real
	EndProcessingTime	Nonnegative real
	RemProcessingTime	Nonnegative real
	MachineFailureFlag	Boolean
	EndRepairFlag	Boolean
Parts	Initialization	Time-based signal
	PartArrival	Time-based signal
	PartInterarrivalTime	Positive real
	NumPartsToProcess	Nonnegative integer
	NumPartsProcessed	Nonnegative integer
	RemProcessingTime	Nonnegative real
	EndService	Time-based signal
	PartsWaiting[1 .. 3]	Nonnegative integer
	EndProcessingTime	Nonnegative real

activity, or process sequences (depending on world view). The individual event, activity or process descriptions are then simplified. Reliance on graphical representations is central to this process.

We identify potential sequences of model actions by first constructing an *Action Cluster Incidence Graph (ACIG).* The graph for the above example is presented in Figure 1. An algorithm to construct this graph is presented in Nance and Overstreet (1986). The ACIG captures is the ability of one action cluster to directly

Table 3: Action Clusters (ACs)

Condition	Action
{ Initialization (IN) } Initialization	Create(Parts) Set Alarm(PartArrival, 0) NumPartsProcessed = 0 Read(PartInterarrivalTime, ProcessingTime, MachineRepairTime, MachineUpTime, NumPartsToProcess) For i := 1 to 3 Create(Machine[i]) Set Alarm(MachineFailure(i), NegExp(MachineUpTime)) Status[i] = idle PartsWaiting[i] = 0 End For

{ Termination (TE) }
NumPartsProcessed = NumPartsToProcess
	Output(SystemTime)
	Stop

{ Part Arrives (PA) } When Alarm(PartArrival)	Set Alarm(PartArrival, NegExp(PartInterarrivalTime)) TargetMachine = Rand(1, 3) PartsWaiting[TargetMachine] = PartsWaiting[TargetMachine] + 1
{ Begin Service (BS) } For Some i (PartsWaiting[i] > 0 & Status[i] = idle)	PartsWaiting[i] = PartsWaiting [i] - 1 EndProcessingTime[i] = NegExp(ProcessingTime) Set Alarm(EndService(i), EndProcessingTime[i]) Status[i] = busy
{ End Service (ES) } For Some i (When Alarm(EndService(i)))	Status[i] = idle NumPartsProcessed = NumPartsProcessed + 1
{ Machine Failure Flag (MF) } For Some i (When Alarm(MachineFailure(i)))	MachineFailureFlag[i] = true
{ Busy Machine Failure (BF) } For Some i (MachineFailureFlag[i] & Status[i] = busy)	MachineFailureFlag[i] = false RemProcessingTime[i] = EndProcessingTime[i] - SystemTime FailedStatus[i] = busy Status[i] = failed Cancel(EndService(i)) Set Alarm(EndRepair(i), NegExp(MachineRepairTime))
{ Idle Machine Failure (IF) } For Some i (MachineFailureFlag[i] & Status[i] = idle)	MachineFailureFlag[i] = false FailedStatus[i] = idle Status[i] = failed Set Alarm(EndRepair(i), NegExp(MachineRepairTime))
{ End Repair Flag (ER) } For Some i (When Alarm(EndRepair(i)))	EndRepairFlag[i] = true
{ Busy Machine Repair (BR) } For Some i (EndRepairFlag[i] & FailedStatus[i] = busy)	EndRepairFlag[i] = false Status[i] = busy Set Alarm(MachineFailure(i), NegExp(MachineUpTime)) Set Alarm(EndService(i), RemProcessingTime[i])
{ Idle Machine Repair (IR) } For Some i (EndRepairFlag[i] & FailedStatus[i] = idle)	EndRepairFlag[i] = false Status[i] = idle Set Alarm(MachineFailure(i), NegExp(MachineUpTime))

cause the occurrence of other action clusters either in the same instant of time or at a future instant of time. The nodes in the graph represent the Action Clusters of Table 3. Two notations for edges are used: ". . ." denotes the ability of one Action Cluster to cause the occurrence of another at a future instant; "---" denotes the ability of one Action Cluster to cause the occurrence of another without a time delay (i.e., at the same instant). Overstreet and Nance (1986) describe simplification procedures for this and related graphs.

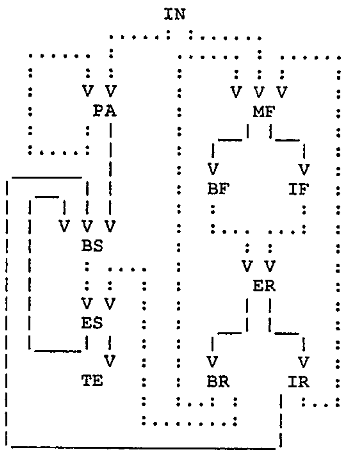

Figure 1. Action Cluster Incidence Graph

The transformation into each of the three world views is based on the above graph and the world view characterizations of Section 4.

- To create an *event scheduling representation*, the ACIG is used to identify all model actions which are directly linked and which can occur in a single instant of time.

The sequences of Action Clusters are represented by the Event Subgraphs of Figure 2.

- To create an *activity scanning representation*, the ACIG is used to identify all model actions which will occur once a specified condition has been met, though the actions may occur at different instants.

The sequences of Action Clusters are represented by the Activity Subgraphs of Figure 3.

- To create a *process interaction representation*, the object specification is used to associate attributes with objects, and the ACIG is again used to describe sequences of actions associated with each object. Sequences of noncurrent, sequential behavior form the basis for each process description.

The sequences of Action Clusters are represented by the Process Subgraphs of Figure 4.

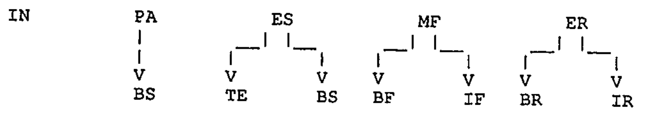

Figure 2: Event Subgraphs

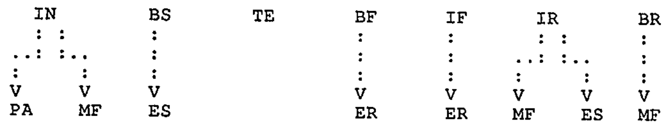

Figure 3: Activity Subgraphs

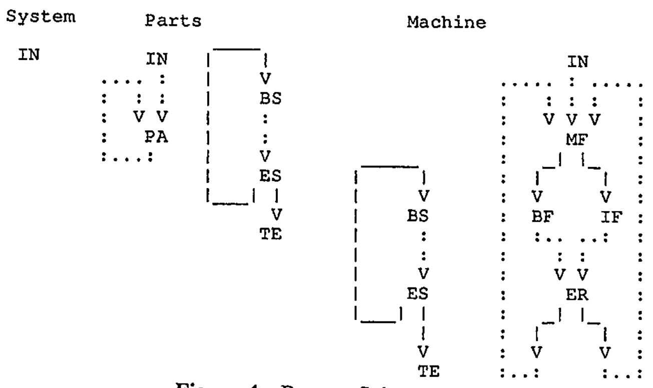

Figure 4: Process Subgraphs

Unsimplified Event

```
Event End Repair ( EndRepair( i ) )
  EndRepairFlag[ i ] = true
 { Busy Machine Repair }
   While ForSome i ( EndRepairFlag[ i ] &
       FailedStatus[ i ] = busy )
     EndRepairFlag[ i ] = false
     Status[ i ] = busy
     Set Alarm( MachineFailure( i ),
       NegExp( MachineUpTime ) )
     Set Alarm( EndService( i ),
       RemProcessingTime[ i ] )
     End While
 { Idle Machine Repair }
   While ForSome i ( EndRepairFlag[ i ] &
       FailedStatus[ i ] = idle )
     EndRepairFlag[ i ] = false
     Status[ i ] = idle
     Set Alarm( MachineFailure( i ),
       NegExp( MachineUpTime ) )
     End While
   End Event
```

Simplified Event

```
Event End Repair ( i )
  If FailedStatus[ i ] = busy )
    Status[ i ] = busy
    Schedule( EndService( i ),
      RemProcessingTime[ i ] )
  Else
    Status[ i ] = idle
    End Else
  EndRepairFlag[ i ] = false
  Schedule( MachineFailure( i ),
    NegExp( MachineUpTime )
  End Event
```

Figure 5: Simplification of End Repair Event Routine

For each transformation, the subgraphs guide the creation of event, activity, and process routines. After the sequence of actions for each routine have been created, simplification of each routine, at least in terms of reducing the number of statements and the type of control structures used, is usually possible. Unfortunately, space limitations prohibit full discussion of this topic. See Overstreet and Nance (1986) for a more complete discussion. Examination of the simplification possibilities for various models resulting from using alternative world views illustrates that each world view allows simplier model specifications for some models; no one is generally superior to the others.

The results of a typical simplification process are illustrated in Figure 5 which presents the End Repair event before and after simplification. Note that both the total number of statements have been reduced and the control flow simplified.

Simplified specifications for each of the world views for the example model are presented in their entirety in figures 6, 7, and 8. The keywords of timing and sequencing constructs have been altered to conform to those commonly associated with each world view.

```
Event Initialization
  Create( Parts )
  Schedule( PartArrival, 0 )
  NumPartsProcessed = 0
  Read( PartInterarrivalTime,
    ProcessingTime,
    MachineRepairTime,
    MachineUpTime,
    NumPartsToProcess )
  For i := 1 to 3
    Create( Machine[ i ] )
    Schedule( MachineFailure( i ), NegExp( MachineUpTime ) )
    Status[ i ] = idle
    PartsWaiting[ i ] = 0
    End For
  End Event

Event Part Arrives
  Schedule( PartArrival, NegExp( PartInterarrivalTime ) )
  TargetMachine = Rand( 1, 3 )
  If Status[ TargetMachine ] = idle
    EndProcessingTime[ TargetMachine ] = NegExp( ProcessingTime )
    Schedule( EndService( TargetMachine ), EndProcessingTime[ TargetMachine ] )
    Status[ TargetMachine ] = busy
  Else
    PartsWaiting[ TargetMachine ] = PartsWaiting[ TargetMachine ] + 1
    End Else
  End Event

Event End Service( i )
  NumPartsProcessed = NumPartsProcessed + 1
  If NumPartsProcessed = NumPartsToProcess
    Output( SystemTime )
    Stop
    End If
  If PartsWaiting[ i ] > 0
    PartsWaiting[ i ] = PartsWaiting [ i ] - 1
    EndProcessingTime[ i ] = NegExp( ProcessingTime )
    Schedule( EndService( i ), EndProcessingTime[ i ] )
  Else
    Status[ i ] = idle
    End Else
  End Event

Event Machine Failure( i )
  If Status[ i ] = busy
    RemProcessingTime[ i ] = EndProcessingTime[ i ] - SystemTime
    FailedStatus[ i ] = busy
    Cancel( EndService( i ) )
  Else
    FailedStatus[ i ] = idle
    End Else
  Status[ i ] = failed
  Schedule( EndRepair( i ), NegExp( MachineRepairTime ) )
  End Event

Event End Repair ( i )
  If FailedStatus[ i ] = busy )
    Status[ i ] = busy
    Schedule( EndService( i ), RemProcessingTime[ i ] )
  Else
    Status[ i ] = idle
    End Else
  EndRepairFlag[ i ] = false
  Schedule( MachineFailure( i ), NegExp( MachineUpTime )
  End Event
```

Figure 6: Event Representation

```
Activity Initialization( Initialization )
  Create( Parts )
  NumPartsProcessed = 0
  Read( PartInterarrivalTime,
    ProcessingTime,
    MachineRepairTime,
    MachineUpTime,
    NumPartsToProcess )
  TargetMachine = Rand( 1, 3 )
  PartsWaiting[ TargetMachine ] = PartsWaiting[ TargetMachine ] + 1
  For i := 1 to 3
    Create( Machine[ i ] )
    Status[ i ] = idle
    PartsWaiting[ i ] = 0
    ConcurrentWait( NegExp( MachineUpTime ) )
    MachineFailureFlag[ i ] = true
    End For
  End Activity

Activity Begin Service( ForSome i ( PartsWaiting[ i ] > 0 & Status[ i ] = idle )
  PartsWaiting[ i ] = PartsWaiting [ i ] - 1
  EndProcessingTime[ i ] = NegExp( ProcessingTime )
  Status[ i ] = busy
  Wait( EndProcessingTime[ i ] )
  Status[ i ] = idle
  NumPartsProcessed = NumPartsProcessed + 1
  End Activity

Activity Termination( NumPartsProcessed = NumPartsToProcess )
  Output( SystemTime )
  Stop
  End Activity

Activity Busy Machine Failure( ForSome i ( MachineFailureFlag[ i ] & Status[ i ] = busy )
  MachineFailureFlag[ i ] = false
  RemProcessingTime[ i ] = EndProcessingTime[ i ] - SystemTime
  FailedStatus[ i ] = busy
  Status[ i ] = failed
  Cancel( EndService( i ) )
  Wait( NegExp( MachineRepairTime ) )
  EndRepairFlag[ i ] = true
  End Activity

Activity Idle Machine Failure( ForSome i ( MachineFailureFlag[ i ] & Status[ i ] = idle )
  MachineFailureFlag[ i ] = false
  FailedStatus[ i ] = idle
  Status[ i ] = failed
  Wait( NegExp( MachineRepairTime ) )
  EndRepairFlag[ i ] = true
  End Activity

Activity Idle Machine Repair( ForSome i ( EndRepairFlag[ i ] & FailedStatus[ i ] = idle )
  EndRepairFlag[ i ] = false
  Status[ i ] = idle
  ConcurrentWait( NegExp( MachineUpTime ) )
    MachineFailureFlag[ i ] = true
    End Wait
  ConcurrentWait RemProcessingTime[ i ] )
    Status[ i ] = idle
    NumPartsProcessed = NumPartsProcessed + 1
    End Wait
  End Activity

Activity Busy Machine Repair( ForSome i ( EndRepairFlag[ i ] & FailedStatus[ i ] = busy )
  EndRepairFlag[ i ] = false
  Status[ i ] = busy
  Wait( NegExp( MachineUpTime ) )
  MachineFailureFlag[ i ] = true
  End Activity
```

Figure 7: Activity Representation

```
Process System
{ Initialization }
  Create( Parts )
  Read( PartInterarrivalTime,
     ProcessingTime,
     MachineRepairTime,
     MachineUpTime,
     NumPartsToProcess )
  For i := 1 to 3
     Create( Machine[ i ] )
     End For
  End Process System

Process Parts
{ Initialization }
  NumPartsProcessed = 0
  Loop
  { Part Arrives }
     TargetMachine = Rand( 1, 3 )
     PartsWaiting[ TargetMachine ] = PartsWaiting[ TargetMachine ] + 1
     Hold( NegExp( PartInterarrivalTime ) )
     End Loop
  End Process Parts

Process Machine1( i )
{ Initialization }
  Status[ i ] = idle
  PartsWaiting[ i ] = 0
  Hold( NegExp( MachineUpTime ) )
  Loop
  { Busy Machine Failure }
    If Status[ i ] = busy
      RemProcessingTime[ i ] = EndProcessingTime[ i ] - SystemTime
      FailedStatus[ i ] = busy
      Passivate( Machine2( i ) )
  { Idle Machine Failure }
    Else
      FailedStatus[ i ] = idle
      End Else
    Status[ i ] = failed
    Hold( NegExp( MachineRepairTime ) )
  { Busy Machine Repair }
    If FailedStatus[ i ] = busy
      Status[ i ] = busy
      Activate( Machine2( i ) )
    Else
    { Idle Machine Repair }
      Status[ i ] = idle
      End Else
    Hold( NegExp( MachineUpTime ) )
     End Loop
  End Process Machine1

Process Machine2( i )
{ Begin Service }
  Loop
    Wait Until( PartsWaiting[ i ] > 0 & Status[ i ] = idle )
    PartsWaiting[ i ] = PartsWaiting [ i ] - 1
    EndProcessingTime[ i ] = NegExp( ProcessingTime )
    Status[ i ] = busy
    Hold( EndProcessingTime[ i ] )
    If Passivated
      Hold( RemProcessingTime[ i ] )
      End If
  { End Service }
    Status[ i ] = idle
    NumPartsProcessed = NumPartsProcessed + 1
  { Termination }
    If NumPartsProcessed = NumPartsToProcess
      Output( SystemTime )
      Stop
      End If
    End Loop
  End Process Machine2
```

Figure 8: Process Representation

5. Summary and Conclusions

Transformations into the event scheduling and activity scanning world views are straightforward, at least into the unsimplified forms. The symmetry of these two representational forms is appealing. Part of the simplification process, which allows the representations to take advantage of the target world view, can be automated, though we have not demonstrated that here. Complete simplification, which takes full advantage of the target world view, is, however, impossible to automate. This is proven in Overstreet (1982). We also observe that use of activity scanning need not result in inefficient execution.

Transformation into process interaction requires more information than is available in the Action Cluster Incidence Graph though the ACIG is sufficient for event scheduling and activity scanning. Our experience with a variety of models demonstrates that the results the process interaction transformations are very dependent on the Action-Cluster Object association provided by the specifier. We know no clear rules for forming these associations. For the examples with which we have dealt, we find that we provide an initial Action-Cluster Object association, perform the transformation using the Action Cluster Incidence Graph (which is a simple process), evaluate the results of this transformation, and often revise the initial Action-Cluster Object association until a "more satisfactory" result is obtained.

In addition, if the model of what a process should look like is based on Simula, and we restrict ourselves to control constructs similar to those of Simula, then the description of each object generated as subgraphs of the Action Cluster Incidence Graph may require addition decomposition. This is because Simula provides no facilities for concurrency (even simulated concurrency) *within* a single process; concurrency is represented by the use of separate processes. The Process Subgraphs generated from the Action-Cluster Incidence Graph may require further decomposition into nonconcurrent components. This was the case in the example presented in this paper. The automation of this decomposition requires further study.

We have demonstrated that much of the model transformation process can be automated. The transformation process enhances our understanding of the relationship and relative advantages of each of the world views. The graphs which support these transformation are useful in a variety of contexts.

Acknowledgements

Many discussions with Richard E. Nance and Osman Balci of Virginia Tech have contributed to and refined the ideas presented here.

REFERENCES

Banks, Jerry and Carson II, John S. (1985). Process-interaction Simulation Languages. *Simulation* 44, 5, May, 225-235.

Bauer, Kenneth W., Kochar, Bipin, and Talavage, Joseph J. (1985). Simulation Model Decomposition by Factor Analysis. In: *Proceedings of the 1985 Winter Simulation Conference* (E D. Gantz, E G. Blais, and E S. Solomon, eds.), (Dec. 11-13), 185-188.

Birtwistle, Graham, Dahl, Ole-Johan, Myhrhaug, Bjorn, and Nygaard, Kristen (1973). *SIMULA Begin.* Studentlitteratur, Auerbach.

Buxton, John N. and Laski, John G. (1963). Control and Simulation Language. *The Computer Journal* V, 194-199.

CACI (1983), *SIMSCRIPT II.5 Programming Language.* C.A.C.I, Los Angeles, CA 90049.

Clementson, Alan T. (1966). Extended Control and Simulation Language. *The Computer Journal* 9, 3, Nov., 215-220.

Dahl, O.-J. and Nygaard, Kristen (1966). SIMULA -- An ALGOL-Based Simulation Language. *Communications of* *the ACM* **9**, Sept., 349-395.

Fishman, George S. (1973). *Concepts and Methods in Discrete Event Digital Simulation.* John Wiley & Sons, New York.

Fishman, George S. (1978). *Principles of Discrete Event Simulation.* John Wiley & Sons, New York.

Gordon, Geoffrey (1978). *System Simulation, Second Edition.* Prentice-Hall, Inc., Englewood Cliffs, NJ.

Henriksen, James O. and Robert C. Crain (1983). *GPSS/H User's Manual.* Wolverine Software Corporation, Annadale, VA 22003-2653.

Hills, P. R. (1967). SIMON -- A Computer Simulation Language in ALGOL. In: *Digital Simulation in Operations Research* (S. H. Hollingdale, ed.). American Elsevier Publishing Co., New York, NY, 105-115.

Hooper, James W. (1986). Strategy-Related Characteristics of Discrete-Event Languages and Models. *Simulation* **46**, 153-159. (See O'Keefe's technical comment and author's response, *Simulation* **47**, 5, Nov. 1986, 208-211).

Hutchinson, G. K. (1975). In Introduction to CAPS -- Computer-Aided Programming for Simulation. *ACM Simuletter* **7**, 1, Oct, 35-51.

IBM (1971), *General Purpose Simulation System V User's Manual.* SH20-0851, IBM, White Plains, NY.

Kiviat, Philip J. (1967). Digital Computer Simulation: Modeling Concepts. RAND Report RM-5378-PR. RAND Corp., Santa Monica, CA, (Aug).

Kiviat, Philip J. (1969). Digital Computer Simulation: Computer Programming Languages. RAND Report RM-5993-PR, RAND Corp., Santa Monica, CA.

Lackner, Michael R. (1962). Toward a General Simulation Capability. In: *Proceedings of the SJCC.* AFIPS Press, (May 1-2), 1-14.

Lackner, Michael R. (1964). Digital Simulation and System Theory. Systems Development Corporation SP-1612, (Apr. 6).

Law, Averill M. and Kelton, W. David (1982). *Simulation Modeling and Analysis.* McGraw-Hill, New York.

Markowitz, H. M., Hausner, B., and Karr, H. W. (1962). SIMSCRIPT: A Simulation Programming Language. Rept. RM-3310-PR, RAND Corp., Santa Monica, CA.

Mathewson, S. C. and Beasley, J. E. (1976). DRAFT/SIMULA. In: *Proceedings of the Fourth Simula Users Conference.* National Computer Conference.

Mathewson, S. C. (1977). *DRAFT.* Department of Management Science, Imperial College of Science and Technology, London, England.

Mathewson, S. C. and J. H. Allen (1977). DRAFT/GASP -- A Program Generator for GASP. In: *Proceedings of the Tenth Annual Simulation Symposium.* Tampa, FL, 211-225.

Mathewson, S. C. (1984). The Application of Program Generator Software and Its Extensions to Discrete Event Simulation Modelling. *IIE Transactions* **16**, 3-18.

Mathewson, S. C. (1985). Simulation Program Generators: Code and Animation on a PC. *Journal of the Operational Research Society* **36**, 583-589.

Moose, Robert L. and Nance, Richard E. (1983). *Model Analysis in a Model Development Environment.* Department of Computer Science, Virginia Tech, Blacksburg, VA. Preliminary Draft, (May 20).

Nance, Richard E. (1981a). The Time and State Relationships in Simulation Modeling. *Communications of the ACM* **24**, 4, Apr., 173-179.

Nance, Richard E. (1981b). Model Representation in Discrete Event Simulation: The Conical Methodology. Technical Report CS81003-R. Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, (Mar. 15).

Nance, Richard E. and Overstreet, C. Michael (1986). Diagnostics Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications. Technical Report TR-86-007. Computer Science Department, Old Dominion University, Norfolk, VA 23508-8508, (March 26).

Nance, Richard E. Overstreet, C. Michael (1987). Exploring the Forms of Model Diagnosis in a Simulation Support Environment. In: *Proceedings of the 1987 Winter Simulation Conference.* Atlanta, GA, (Dec. 14-16).

Overstreet, C. Michael (1982). Model Specification and Analysis for Discrete Event Simulation. PhD Dissertation. Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, (Dec.).

Overstreet, C. Michael and Nance, Richard E. (1985). A Specification Language to Assist in Analysis of Discrete Event Simulation Models. *Communications of the ACM* **28**, 190-201.

Overstreet, C. Michael and Nance, Richard E. (1986). World View Based Discrete Event Model Simplification. In: *Modelling and Simulation Methodology in the Artificial Intelligence Era* (Bernard P. Zeigler, ed.). North-Holland Publishing Co., Amsterdam, The Netherlands, 165-179.

Russell, Edward C. (1983). *Building Simulation Models with SIMSCRIPT II.5.* C.A.C.I., Los Angeles, CA 90049.

Schriber, Thomas J. (1974). *Simulation Using GPSS.* John Wiley & Sons, New York, NY.

Weinberg, Gerald M. (1971). *The Psychology of Computer Programming.* Van Nostrand Reinhold, New York.

Zeigler, Bernard P. (1976). *Theory of Modelling and Simulation.* John Wiley & Sons, New York.

Zeigler, Bernard P. (1984). *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, Orlando, Florida.

AUTHOR'S BIOGRAPHY

C. MICHAEL OVERSTREET is an Assistant Professor of Computer Science at Old Dominion University. He received his Ph.D. in 1982 at Virginia Polytechnic Institute and State University. He is currently Principal Investigator for a Navy-funded project in the development of software analysis tools and vice-chair of the ACM Special Interest Group on Simulation (SIGSIM). Dr. Overstreet is a member of ACM, IEEE CS, and SCS.

C. Michael Overstreet
Computer Science Department
Old Dominion University
Norfolk, VA 23529-0162, U.S.A.
(804) 440-4545