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

IMPLEMENTING THE PRODUCT AUTOMATON FORMALISM

Frederick J. Portier
Department of Mathematics
The University of North Carolina
at Greensboro
Greensboro, NC 27412, U.S.A.

ABSTRACT

A formalism for discrete simulation is a set of conventions for the construction of discrete simulation models. In this paper we define the product automaton formalism. The formalism is defined as a mathematical object and is independent of any programming language. Implementing the formalism in a specific programming language entails associating the various components of the formalism with language constructs. We show how the formalism can be implemented in any procedural language. Furthermore, the implementation closely parallels the formalism in both structure and function. As an example, we present an implemenation in Modula-2. A novel feature of Modula-2 is used in this recursive implementation. The paper concludes with a brief discussion of a tool being developed that will provide support for modelling using this formalism.

1. INTRODUCTION

A formalism for discrete simulation allows one to specify a discrete simulation model in a precise, unambiguous manner. A well-known formalism, developed by Zeigler (1984), is the discrete event system specification (DEVS) formalism. One form of the DEVS has one specify a system in a modular fashion; the system is decomposed into a hierarchy of component subsystems. This DEVSM (DEVS in Modular form) formalism has been implemented (Zeigler 1986).

Formalism, modularity, and hierarchy combined provide a powerful expressive capability. Formalism reduces the ambiguity of model specification. Modularity allows one to specify and understand a model as a collection of smaller entities where each entity is precisely defined and communicates with other entities through standard ports. Hierarchy allows one to specify and understand a model as a decomposition of subordinate models.

In this paper we present a formalism, the Product Automaton (PA) formalism, that is related to the DEVSM formalism. Like the DEVSM formalism, it makes extensive use of modularity and hierarchy. However, there are major differences between the two, especially with respect to world view. Both formalisms view a system as a finite decomposition of subsystems and both maintain subsystem states in a modular fashion. However, the definition of state and the manner in which state transitions occur differ in the two formalisms.

The PA formalism alone does not immediately provide a simulation program. In fact, the formalism is independent of any programming language. To implement the formalism, one needs to associate the components of the formalism with constructs in the language chosen. In particular, one would like a set of language constucts that closely parallels the components in the formalism. A set of program constructs is provided in Section 3. As an example, constructs in Modula-2 are given. A set of programming constructs in FORTRAN corresponding to an earlier version of the PA formlism can be found in Haymond (1978). The paper concludes with a brief discussion of a tool being developed that will provide support for modelling using this formalism.

2. THE PRODUCT AUTOMATON FORMALISM

Central to the PA formalism is the notion of coupling of subsystems; a system is defined as either either indecomposable or made up of subordinate subsystems. A <u>decomposition tree</u> describes this hierarchy. The root of the tree represents the entire system, the children of a node represent the component subsystems of that node, and the leaves represent atomic subsystems. The definition of the PA formalism starts with a decomposition tree and defines the manner in which subsystems associated with the nodes of this tree interact.

Let T be a rooted tree. Then

$$T \equiv <N,E,r>$$

where N is a finite set called the <u>node set</u> and E is a subset of $N \times N$ called the <u>edge set</u>. The node $r \in N$ is called the <u>root</u> and <N,E> is a tree.

We can write N as the disjoint union of the sets {r} ,Inter_N, and Atom_N where Inter_N (Intermediates) is the set of nodes that have an ancestor and a descendent and Atom_N (Atoms) is the set of nodes that have an ancestor but no descendents. A rooted tree is shown in Figure 1. The node set is N = {A,B,C,D,E} with r = A, Inter_N = {B} and Atom_N = {C,D,E}.

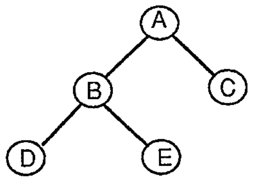


Figure 1: A Rooted Tree

We can now define the Product Automaton Formalism. We first present the product automaton as a mathematical object (subject to a number of constraints) and then later discuss its use as a formalism for simulation.

A **Product Automatom** is a structure

$$PA \equiv <N,E,r,\{M_i\},\{Z_{i,j}\}>$$

where <N,E,r> is a rooted tree and for all $i \in N$

$$M_i \equiv <X_i,S_i,\partial_i,ta_i>.$$

Each M_i is called a <u>component</u>. Defining a component are the following:

X_i is a finite set, the <u>set of inputs</u> to i,
S_i is a set, the <u>state set</u> of i,
∂_i is function, the <u>state transition function</u> of i,
$ta_i \in R^+_{0,\infty}$ is the <u>natural update time</u> of i ($R^+_{0,\infty}$ is the extended non-negative real numbers)

The objects defined above are subject to the following constraints (Axioms):

(1) for all $i \in N$ there exists sets $\{C_{i,k}\}, k = 1, \ldots, n_i$, such that
$$S_i \equiv \underset{k}{\times} C_{i,k}.$$

(2) for all $i \in N$ we have that
$$C_{i,1} \equiv R^+_{0,\infty}$$
and if $s \in S_i$ is the current state of i, then ta_i is s_1.

(3) for all $i \in N$ we define
$$\partial_i: X_i \times S_i \rightarrow S_i \text{ for } i \in \text{Atom}_N$$
$$\partial_i: \underset{\substack{j \\ (i,j) \in E}}{\times} S_j \rightarrow S_i \text{ for } i \notin \text{Atom}_N$$

(4) there exists inputs $\upsilon, \sigma, \varnothing$ such that
$\upsilon \in X_i$ for all $i \in N$,
$\varnothing \in X_i$ for all $i \neq r$, and $X_r \equiv \{\upsilon, \sigma\}$

(5) $Z_{i,j}: X_i \times S_i \rightarrow X_j$ for all $(i,j) \in E$

(6) $ta_i \equiv \min\limits_{\substack{j \\ (i,j) \in E}} ta_j$

(7) $Z_{i,j}(\upsilon,s) \equiv \begin{cases} \upsilon & \text{provided } ta_i = ta_j \\ \varnothing & \text{provided } ta_i \neq ta_j \end{cases}$

(8) $Z_{i,j}(\varnothing,s) \equiv \varnothing$ for $(i,j) \in E$
$\partial_i(\varnothing,s) \equiv s$ for $i \in \text{Atom}_N$

(9) $Z_{i,j}(x,s) \neq \upsilon$ for $x \neq \upsilon$

(10) The sequence of inputs that r receives is $\{\upsilon, \sigma, \upsilon, \sigma, \ldots\}$.

Thus, a product automaton is a collection of interacting and interrelated components. The components are arranged as in a decomposition tree and interact only with adjacent components as defined by the tree. To make the relationships between the components precise, we define the terms factor and product. A factor is a component that is part of some decomposition and a product is a component that has factors. The interactions between components are then only between products and factors.

Every component i maintains a current state (an element of S_i), accepts inputs (from X_i), and changes state (using ∂_i). Also, every non-atom component supplies inputs to its factors (using the fuctions $\{Z_{i,j}\}$). Thus, a component can be thought of as an automaton and a product can be thought of as a product of automata.

State transitions occurs in two forms (axiom 3). An atom has state transitions based soley on the input received and on its internal state whereas a product changes state based on the state of its factors.

We have defined three special inputs υ, σ, and $\varnothing$. The input υ plays a central role in the timing of the model and is discussed later. The input $\varnothing$ is the null input; when a component receives $\varnothing$ it does not change state (axiom 8). The input σ is the input that the root receives that is not the special input υ. The root alternately receives the inputs υ and σ (axiom 10); as we will show later, this defines two distinct phases in the simulation. Note that the root receives external inputs and all other components receive inputs from their product.

A product automaton serves as an abstraction of a physical system. The root represents the entire system and the factors of a product represent subsystems of that product. The PA formalism is a product automaton with an associated algorithm. Briefly, the simulation algorithm is as follows:

Algorithm 1

1) the root receives an input
2) based on state and input received, products send inputs to factors
3) when an atom receives an input, a state transition occurs
4) after all factors of a product have changed state, the product changes state
5) after the root has changed state, go to step 1

Associated with each component is a number called the natural update time for that component; it is the first entry in the current state of the component (axiom 2). This number is interpreted as the scheduled time of the next state transition for this component. By axiom 6 we know that the natural update time for a product is the minimum of the natural update times of its factors. This is consistent with the interpretation given above since the time scheduled for a product to change state must be the minimum of the times scheduled for the components composing the product.

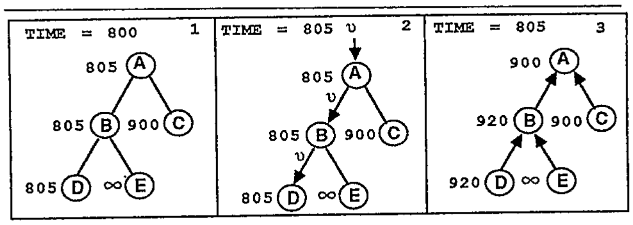
The time at which the next state transition is scheduled to occur for the entire system is the natural update time of the root. This corresponds, ultimately, to some set of atoms with this same natural update time. When simulated time is progressed to the natural update time of the root, this set of atoms must experience state transitions. This is accomplished using the special input υ as follows:

1) The root receives υ as an input
2) υ is passed from product to those factors with equal natural update time (axiom 7)
3) The correct set of atoms receive υ as an input and experience state transitions

We now define a variable called TIME which is external to the product automaton. Algorithm 2 incorporates TIME into the simulation algorithm. Figure 2 describes the behavior of a product automaton upon the receipt of υ; the natural update times are shown to the left of each node.

Algorithm 2

```
1) Set TIME to the natural update time of
   the root
2) the root receives the input $\upsilon$
3) after the root changes state, the root
   receives the input $\sigma$
4) After the root changes state, go to 1)
```



1 - This is the initial state of the system. The next scheduled state transition is at 805.

2 - TIME is progressed to 805 and A receives the input υ. Products supply υ to factors that have natural update time equal to 805.

3 - Once the atoms have changed state, the states of the products are defined in terms of the states of their factors. D changes state followed by B and then finally A.

Figure 2: Behavior of the Product Automaton

As described above, the input υ is related to scheduled state transitions. All other non-null inputs are related to nonscheduled, conditional state tansitions. When the root receives the input σ, a decision must be made by the root as to the necessity of performing conditional state transitions in the system. If conditional state transitions are to occur, the root sends an input (not υ, by axiom 9) to the affected components. Those intermediates that receive non-null inputs decide what conditional state transitions must be performed in their subsystems and send inputs as necessary. This continues until some set of atoms receive inputs, at which time they change state.

Let us define branching as the decision process conducted by a product that results in a set of inputs being supplied to its factors. Branching is based on both the current state of the component and the input it receives. The set of functions $\{Z_{i,j}\}$ accomplishes branching for i.

Branching comes in two varieties. Upon the receipt of υ, branching insures that the correct set of scheduled state transitions occurs. Upon the receipt of an input other that υ or $\varnothing$, branching causes a hierarchical decision process to occur; the input received is an encoding of the results of a decision process conducted by its ancestors and the set of inputs it sends out is a continuation of this same decision process.

The state of a component must contain sufficient information for the branching to take place. This information is communicated to the component by way of the states of its factors. Although it would suffice to define the state of a product as the cartesian product of the states of its factors, it is likely that processed information is what is really needed. Thus, we define the state of a product as a function of the states of its factors.

It is convenient to think of the simulation algorithm as consisting of two phases. During the natural phase scheduled state transitions occur and during the unnatural phase conditional state transitions occur. The natural phase is initiated when the root receives the input υ and continues until the root changes state. The unnatural phase begins once the root receives σ and continues until the root changes state. Since the root alternately receives υ and σ, the algorithm alternates between these two phases.

We can now describe the entire simulation algorithm. We incorporate initialization, output, and a test for termination.

Algorithm 3

```
1) Initialize TIME
2) Initialize the product automaton
3) Output the event vector
4) Let TIME = the natural update time of r
5) Test for termination
6) Natural Phase
7) Unnatural Phase
8) go to step 3
```

The event vector is a vector of values maintained internally within the components at the simulated time TIME. They can be considered part of the states of the components.

Initialization of the product automaton must drive each of the states of the components to some set of initial states that are consistent with the value of TIME and the modelled initial system. Since the state of a product is a function of the states of its factors, initial states of all factors of a product must be defined prior to initializing the product. Initialization can then proceed as in a postorder traversal of the decomposition tree.

A Small Example

To illustrate how the formalism is applied to a physical system, we consider a small example. Figure 3 represents a collection of conveyor belts. Conveyor belts A and B feed into conveyor belt C. Objects originate on belt A and B according to random processes. It takes an object 12 seconds from the time it enters belt A to the time it enters belt C; likewise, it takes 15 seconds to travel from B to C. An object on belt C travels to the end of C and falls off (8 seconds). Objects enter belt C on a first come, first serve basis; if there is a tie, the object on A enters with probability .65. An object held up due to a tie will enter belt C exactly 1 second later. We assume that objects are generated no sooner than 2 seconds apart so that the system does not backup.

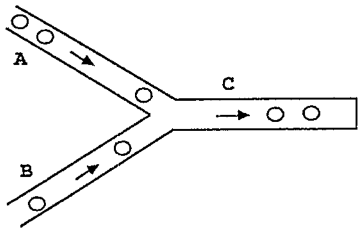


Figure 3: The Conveyor Belt System

One possible decomposition tree for the system is shown in Figure 4. The node set is **S** (SYSTEM), **E** (Entering), **A** (Belt A), **B** (Belt B), and **C** (Belt C). Beside each node are variables which define the state of the component. "AtIntersection" (**S**) has the value TRUE if there is any object ready to enter belt C. "Present" (**B**) has one of the values AOnly, BOnly, Both, or Neither indicating the possible arrangement of objects ready to enter belt C. Note that "AtIntersection" can be determined from the value of "Present" (AtIntersection = Present <> Neither). "AtIntersection" for **A** is TRUE if there is an object on belt A ready to enter belt C. "AtIntersection" for **B** is similarly defined. Note that "Present" can be determined from the values of "AtIntersection" for **A** and **B**.

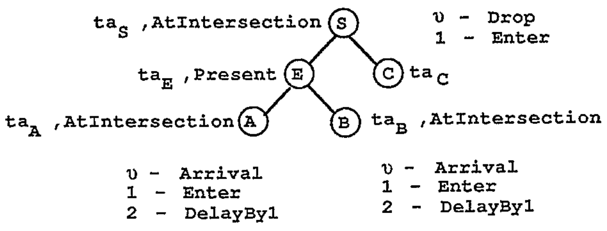


Figure 4: Decomposition Tree for Conveyor Belt System

Near each atom in Figure 4 is a list of the possible inputs and the kind of state transition such an input will cause. These operations are informally descibed as follows:

Arrive: An object either arrives onto the conveyor belt or arrives at the intersection with belt C

Drop: An object falls off conveyor belt C.

Enter: An object at the intersection with belt C enters belt C.

DelayBy1: An object that has arrived at the intersection with C is delayed entering C for 1 second.

Note that Arrive and Drop are scheduled state transitions. Thus, when an "Arrive" occurs, it is necessary to schedule another or set the natural update time for the component equal to ∞.

The branching during the unnatural phase is described by the following pseudo-code.

```
S receives $\sigma$:
   IF AtIntersection THEN
      Send 1 to E
      Send 1 to C
   END

E receives 1:
   CASE Present OF
      AOnly:   Send 1 to A
      BOnly:   Send 1 to B
      Both:    Determine who enters
               IF A enters THEN
                  Send 1 to A
                  Send 2 to B
               ELSE
                  Send 2 to A
                  Send 1 to B
               END
      Neither: < this is not possible>
   END
```

Missing from this brief discussion are the internal workings of each of the atoms, the initialization for each of the components, and a designation of an event vector.

3. THE PROGRAM TEMPLATE

Implementing the PA formalism involves associating the various components of the formalism with language constructs. In this section we will describe a general scheme for implementing the formalism in any procedural programming language. In the next section we will be more specific and present an implementation in Modula-2.

A module is a collection of program objects. These objects are either private (accessible only within the module) or public (accessible to objects outside of the module). Thus, a module is a structuring tool that allows one to group program objects (procedures, variables, constants, etc.) into a single entity that communicates with other such entities only through those objects that are public.

Our implemention of the PA formalism begins by defining one module for each of the components of the product automaton. Each module is structurally similar and communicates with other modules in a formal manner. In particular, we define module templates for the atoms, intermediates, and the root. A module template is a specification of the module's objects and the visibilty of these objects. Figure 5 gives the module templates.

Root	STATE	
	INPUT SET	
	LOCAL VARIABLES	
	BRANCH ROUTINE	
	STATE ROUTINE	
	INIT ROUTINE	
	PRINT ROUTINE	
Intermediate	STATE	(visible to parent)
	INPUT SET	(visible to parent)
	LOCAL VARIABLES	
	BRANCH ROUTINE	
	STATE ROUTINE	
	INIT ROUTINE	
	PRINT ROUTINE	
Atom	STATE	(visible to parent)
	INPUT SET	(visible to parent)
	LOCAL VARIABLES	
	STATE ROUTINE	
	INIT ROUTINE	
	PRINT ROUTINE	

Figure 5: Module Templates

STATE is a set of variables defining the state of the component. The INPUT SET is a collection of constants. LOCAL VARIABLES are variables used internally within the modules and are not part of the state. The BRANCH ROUTINE is a procedure that does the branching (supplying of inputs to factors) for all inputs other than υ. Branching upon the receipt of υ is completely specified and is handled seperately. The STATE ROUTINE accomplishes state transition for the component and the INIT ROUTINE sets the initial state of the component. The PRINT ROUTINE outputs the component's portion of the event vector.

The module defined for a component contains procedures that are encodings of the operations defined by that component. External to all of these modules, we supply the code necessary to execute the procedures in the order defined by the simulation algorithm. Figure 6 describes algorithms that accomplish this.

```
PROCEDURE SIMULATE
   INITIALIZE
   LOOP
      PRINT_EVENT_VECTOR
      TIME := natural update time of root
      IF <termination condition> THEN EXIT END
      NATURAL_PHASE(root)
      UNNATURAL_PHASE(root)
   END
END SIMULATE

PROCEDURE INITIALIZE
   FOR all nodes in a postorder traversal DO
      INIT for node
   END
END INITIALIZE

PROCEDURE PRINT_EVENT_VECTOR
   FOR all nodes DO
      PRINT for node
   END
END PRINT_EVENT_VECTOR

PROCEDURE NATURAL_PHASE(node)
   IF node is an atom THEN
      REPEAT
         STATE for node with input $\upsilon$
      UNTIL natural update time of node $\neq$ TIME
   ELSE
      FOR all children of node DO
         NATURAL_PHASE(child)
      END
      STATE for node
      UPDATE(node)
   END
END NATURAL_PHASE

PROCEDURE UNNATURAL_PHASE(node)
   IF Input = $\emptyset$ THEN RETURN END
   IF node is an atom THEN
      STATE for node
   ELSE
      BRANCH for node
      WHILE node sent some input $\neq \emptyset$ DO
         FOR all children of node DO
            UNNATURAL_PHASE(child)
         END
         STATE for node
         UPDATE(node)
         BRANCH for node
      END
   END
END UNNATURAL_PHASE

PROCEDURE UPDATE(node)
   Make the natural update time of node equal to
   the minimum of the natural update times of
   the children of node
END UPDATE
```

Figure 6: Pseudocode for Simulation Algorithm

The pseudocode found in Figure 6 uses several conventions. Procedures are all in capital letters and procedure references are indicated by naming the procedure. The use of "for node" signals that the given procedure is the one associated with the node "node" and found in the module defined for "node".

The six procedures defined are based on the tree structure of the product automaton only. They are easily implemented once the issue of tree representation has been decided. These six procedures along with the module templates define the program template. To write the simulation program one need only supply the procedures given by the program template.

To illustrate the program template concept, we consider a small example. Here **A** is a product of **B** and **C**, and **B** is a product of **D** and **E**. Graphically, the program template is given in Figure 7. Since there are five components, we define five module templates. The order of execution is from top down, the same direction in which inputs are passed. The flow of state information is from bottom up.

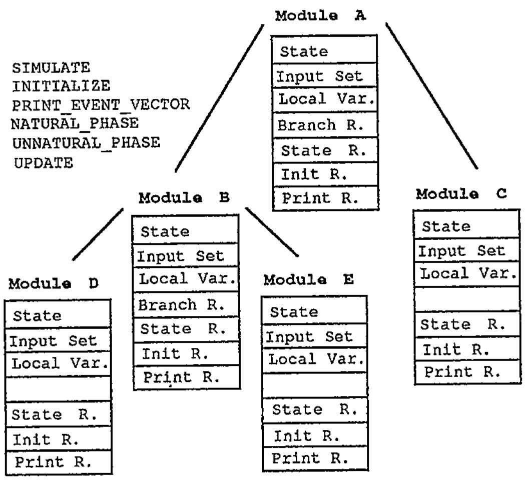


Figure 7: The Program Template

There are several reasons for adopting such a formal manner of programming. These are as follows:

1) The implementation is very closely related to the formalism. A large proportion of time can be spent on model specification rather than on programming.

2) The properties of modularity and hierarchy are maintained.

3) The design of the program is explicitly given. This aids in documenation, debugging, and maintainance.

4) A modification to the model results in a similar modification to the program. Even major modifications, such as replacing an atom by a product or adding an additional factor to a product can be implemented in an unambiguous manner.

5) Computer assistance in the form of partial code generation is possible.

There two recognized problems with this manner of implementation. The first is common to all simulation programs written in general purpose languages; namely, statistics gathering must be done mannually. This implementation prints the raw data (event vector) from which statistics can be gathered.

A second problem is one involving the communication of information. The only flow of information thus far has been 1) inputs from product to factor and 2) state information from factor to product. It is possible that some piece of information maintained within a component, **A**, is needed by another component, **B**, and **A** and **B** are not related by the factor/product relationship. This problem is overcome by defining additional procedures in the existing modules whose sole purpose is the communication of information. Thus, **B** references a routine in the module for **A** that transfers the desired information. Although this does solve the problem, it also lessens the degree of modularity and complicates the design of the program.

4. MODULA-2 IMPLEMENTATION

Modula-2 is a language created by Niklaus Wirth (1982). It was developed in responce to several inadequacies of Pascal. Although a new language, it is sufficiently similar to Pascal to allow an easy transition from Pascal. For our purposes, only two of the new features of Modula-2 are necessary: modules and procedure variables. For a complete discussion of Modula-2 and a comparison with Pascal see Gleaves (1984).

Modules appear as language elements in Modula-2. Objects, such as variables, procedures, constants, or type definitions, are defined within a module and may be EXPORTed to other modules. Likewise, a module may use an object exported by another module by IMPORTing that object.

There are four types of modules: program modules, local modules, definition modules, and implementation modules. Definition and implementation modules are used for the seperate compilation of modules. A program module is a Modula-2 program which may import objects from seperately compiled modules. A local module is a module contained within another module or within a procedure. We will use only program modules and local modules since seperate compilation is more involved

The modules defined in our program template are implemented as local modules. In particular, they are local modules of the program module; i.e., the nesting of local modules is only to one level. Objects defined within one of these modules and exported are then available within the main module and available for import by other local modules. Also, variables defined as private (local) to the local module retain their values throughout program execution.

As described by the module templates, the BRANCH, STATE, INIT, and PRINT routines are included in the various modules. These procedures are exported since they will be referenced by the program module. A naming convention allows one to name every Branch Routine "PROCEDURE BRANCH", every State Routine "PROCEDURE STATE", etc. If we name the module associated with component **A** by "MODULE A", we can refer to the Branch Routine for **A** by "A.BRANCH".

The STATE is conveniently implemented as a RECORD data structure and the INPUT SET as a collection of INTEGER constants. Both of these are defined within a module and exported where they are then imported by the product's module.

The implementation of the remaining program template relies on a somewhat novel feature of Modula-2. A procedure variable is a variable whose values are the names of procedures. Once assigned, the variable may then be used to reference the procedure. As with all variables in Modula-2, it is of a specified type; in this case, type PROCEDURE.

We will use procedure variables to invoke the routines BRANCH, STATE, INIT, and PRINT for each of the modules. Conceptually, we "embed" the decomposition tree in the program module and "attach" to each node of this tree the routines BRANCH, STATE, INIT, and PRINT. Implementation of the recursive routines such as INITIALIZE is then little more than a textbook example of a tree traversal.

In particular, we represent the tree as a dynamic data structure where each node of the tree has the following record structure:

```
NodePtr = POINTER TO Node;
Node = RECORD
         FChild :NodePtr;
         NxChild:NodePtr;
         Branch :PROCEDURE (VAR BOOLEAN);
         State  :PROCEDURE;
         Init   :PROCEDURE;
         Print  :PROCEDURE;
         Update :INTEGER;
         Input  :INTEGER;
        END;
```

This method of tree representation is sometimes called the binary correlative representation of a general tree. The leftmost child of a node, A, is pointed to by the FChild field found in the record associated with node A; subsequent children are referenced by traversing the NxChild fields as in a singly linked list. The fields Branch, State, Init, and Print are procedure valued and are assigned the names of the corresponding procedure. Update is the natural update time of the node; technically, this is part of the state, but it is convenient to locate it here. Input is the input supplied to the node.

To illustrate some of these ideas, the procedures Simulate and Natural_Phase are given in Figure 8. The numbers to the right are for reference purposes.

```
VAR                                                (1)
   root:NodePtr;                                   (2)
   TIME:INTEGER;                                   (3)
                                                   (4)
PROCEDURE Simulate;                                (5)
   BEGIN                                           (6)
      TIME := <initial time>;                      (7)
      Initialize;                                  (8)
      LOOP                                         (9)
         Print_Event_Vector;                       (10)
         TIME := root^.Update;                     (11)
         IF <termination condition> THEN           (12)
             EXIT                                  (13)
          END;                                     (14)
         Natural_Phase(root);                      (15)
         root^.Input := Unnatural;                 (16)
         Unnatural_Phase(root);                    (17)
      END;                                         (18)
   END Simulate;                                   (19)
                                                   (20)
PROCEDURE Natural_Phase(node:NodePtr);             (21)
   VAR                                             (22)
      nx:NodePtr;                                  (23)
   BEGIN                                           (24)
      IF node^.FChild = NIL THEN                   (25)
         node^.Input := Natural                    (26)
         REPEAT                                    (27)
            node^.State;                           (28)
         UNTIL node^.Update <> TIME                (29)
      ELSE                                         (30)
         nx := node^.FChild;                       (31)
         REPEAT                                    (32)
            IF node^.Update = nx^.Update THEN      (33)
               Natural_Phase(nx);                  (34)
            END;                                   (35)
            nx := nx^.NxChild;                     (36)
         UNTIL nx = NIL;                           (37)
         node^.State;                              (38)
         Update(node);                             (39)
      END;                                         (40)
   END Natural_Phase;                              (41)
```

Figure 8: Procedures Simulate and Natural_Phase

Prior to referencing the procedure Simulate, the tree must be created and the correct assignments made to the fields Branch, State, etc. For example, if A is a product, and we name the module for A by MODULE A and the state routine is called State, then we can expect the following assignments:

```
NEW(node);     (* create the node for A *)
node^.State := A.State
```

The state routine for A is then executed when node^.State is encountered as in lines 28 and 38. Note that an atom is detected by the FChild field of a node having the value NIL (line 25).

5. COMPUTER ASSISTANCE

The PA formalism provides a well defined representation for a simulation model. By adhering to the program template approach, the implementation is partially specified. If one now chooses a specific programming language and a method for representing the tree in that language, one has a very precisely defined methodology for the simulation modelling process.

Currently under development is a tool that will aid in the production of simulation programs using this methodology. The tool allows one to graphically specify the decomposition tree. Code for the objects in the module template for each of the components is then supplied in a multiple window environment. The tool then generates the remaining code, yielding a complete simulation program. Ultimately, the tool will allow one to perform high level operations, such as combining existing models, in a graphical manner.

6. CONCLUDING REMARKS

In this paper we have presented the PA formalism and discussed its implementation. The formalism assumes a very specific world view; namely, a system is decomposed into a hierarchy of subsystems each of which can experience scheduled and conditional state transitions. Since the appropriatness of a world view depends on the system to be modelled, one would not expect the PA formalism to be the formalism of choice for all systems. One possible extension to this formalism is allow a subsystem to be a defined as a network of components. This is currently being studied.

BIBLIOGRAPHY

Gleaves, R., (1984). Modula-2 for Pascal Programmers, Springer Verlag, NY

Haymond, R., (1978). "A Programming Theory for Discrete Simulation", Proceedings of the 1978 Winter Simulation Conference

Wirth, N., (1982). Programming in Modula-2, Springer-Verlag, NY.

Zeigler, B., (1984). Multifacetted Modelling and Discrete Event Simulation, Academic Press.

Zeigler, B. (1986). "Hierarchical Modular Modeling/Knowledge Representaion". Proceedings of the 1986 Winter Simulation Conference

AUTHOR'S BIOGRAPHY

FREDERICK J. PORTIER is an Assistant Professor in the Department of Mathematics at The University of North Carolina at Greensboro. He received his B.S. in Mathematics from Nicholls State University, La., in 1980 and his M.S. and Ph.D. degrees in Mathematical Sciences from Clemson Univeristy in 1982 and 1985 respectively. His interests include discrete simulation, computational mathematics, and modelling. He is a member of SCS, ACM, and AMS.

Frederick J. Portier
Department of Mathematics
The University of North Carolina at Greensboro
Greensboro, NC 27412
(919) 334 - 5836
Bitnet Address: PORTIER@UNCG