Technical Report CS79002-R

INTEGRATED MODEL PLURALISM: AN ALTERNATIVE TO A UNIVERSAL MODEL DESCRIPTION LANGUAGE

by

Bernard P. Zeigler
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, Isreal

This research was done partly while the author was a visiting professor with the Department of Computer Science, Virginia Polytechnical Institute and State University, Blacksburg, Virginia 24061. It is supported by Grant No. DAERO-78-088 of the European Research Office, U.S. Army.

INTEGRATED MODEL PLURALISM: AN ALTERNATIVE TO A UNIVERSAL MODEL DESCRIPTION LANGUAGE

by

Bernard P. Zeigler
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel

This research was done partly while the author was a visiting professor with the Department of Computer Science, Virgina Polytechnical Institute and State University, Blacksburg, Virginia 24061. It is supported by Grant No. DAERO-78-088 of the European Research Office, U.S. Army.

Abstract

This paper elaborates an integrated hierarchy of model description formalisms previously proposed. We discuss the advantages of the hierarchical approach vis a vis a standardized universal model description language. Among the principal advantages are the opportunities provided for concise model specification while at the same time enabling equivalence testing of models expressed in diverse formalisms.

The approach is illustrated in the cases of next event and activity scan formalisms. Descriptions within these formalisms contain only the information needed to distinguish between models of the same class. The missing information is reintroduced by translating these descriptions into the more general discrete event formalism and ultimately into the most general systems theoretic formalism.

To illustrate how the above decomposition is achieved, similarly structured simulation schemes for next event and activity scanning models are expressed in a SIMULA like language with the help of its class construct. Actually several interesting constraints are uncovered in the existing SIMULA which prohibit its meeting the level of generality required.

Finally an application of the approach is made to discerning the relative descriptive powers of the next event and activity scanning formalisms. In the course of this development, we provide a scheme for efficient realization of activity scanning which takes advantage of the structural information asked for in the formalism. The net result includes an improved understanding of the handling of conditional (state) and unconditional (time) events in discrete event simulation languages.

CR Categories: 8.1, 4.2

Key words and phases: model representation, system specification formalisms, model semantics, discrete event, nest event, activity scanning, SIMULA

1. INTRODUCTION

Models begin their lives as more or less vague conceptions in people's minds about how things work. They proceed through various courses and stages of formalization, one such course commonly being the expression of the model as a computer simulation program. Indeed in the computer simulation context, people find it hard to distinguish the program from the underlying model.* When pressed, however, they will usually admit that the model could be expressed quite differently in other languages on other computers and must therefore be something more abstract and distinct from its current program embodiment.

Reasons for the model/program confusion may include:

1) The sheer complexity of a program, once written, makes it difficult to rewrite in the same or another language. And, when a program is not easily transformed pragmatically, it is not easily transformed conceptually, as well.

2) The elaboration of the model from vague conception to concrete realization usually is prompted, guided (and constrained) by the coding process. This further adds to the illusion that the final product (the model) is inseparable from the text in which it is expressed (just as an English poem is precisely a set of English sentences and no others in whatever language).

3) Lack of familiarity with alternate languages may commit the modeler to the limited world view of the language he has learned well. The modeler is thus unaware of any alternative conceptual frameworks in

---

*Indeed, the tendency to identify the two is no better illustrated than in [1] where a model is defined as a program.

which his model may be expressible (and is moreover unaware that he is unaware).

Recognition of the independence of models from programs is important in achieving various methodological objectives including improved computer assistance in the following:

- model documentation (man-man communication)
- model formulation and implementation (man-machine communication)
- program-model-real system equivalence testing (verification, validation)
- model manipulation (simplification, decomposition, coupling, modification, etc.)
- model integration (structured organization of models for question-based storage and retrieval).

(See Oren [2] for more details.)

Nance [3] has suggested that development be undertaken of a standardized model representation mechanism for discrete event models. Such a mechanism should facilitate human comprehensible representations of a model at various levels of abstraction ranging from the high level appropriate to policy application to the low level suitable for program implementation. A model represented in the language component of the mechanism should be amenable to analysis to determine which of the three major world views (next event, activity scanning, process interaction) the model is most "comfortable" in (e.g., in which form would the model find its most efficient simulation).

Such a language would work toward removing the confusion between model and program and would increase user perception of the expressive scopes of the various simulation languages. But, unless carefully designed, as a

general tool, it would be so cumbersome that humans could not conveniently develop a model in it. Moreover, the combinatorics of the case argue against mechanical discovery of the most "comfortable" form of implementation given the abstract model description.

Thus, while program independent means of model description remains a goal, it should be recognized that both specialized and generalized description mechanisms must be employed within an integrated framework.

In this paper we elaborate on an integrated hierarchy of model description formalisms previously proposed [4,5]. Our goal is to provide a theoretical foundation for the design of software systems capable of facilitating effective model representation.

2. PRELIMINARY EXAMPLE

We shall illustrate our approach with a simple, rather contrived, example[1], which is illustrated in Figure 1. Suppose we have a language, call it S, for specialized formalism, which has the following syntax:

---

[1] Some readers may detect unmistakable elements of category theory and programming semantics in what follows and may question why we don't explicitly use such formulations. Some reasons are:

a) These abstractions are (still?) incomprehensible to most audiences.

b) They are too powerful considering our limited objective: getting across some fairly simple and basic ideas.

c) In any case, I remain to be convinced that the apparatus offered by more abstract approaches can be of direct use in the modelling context.

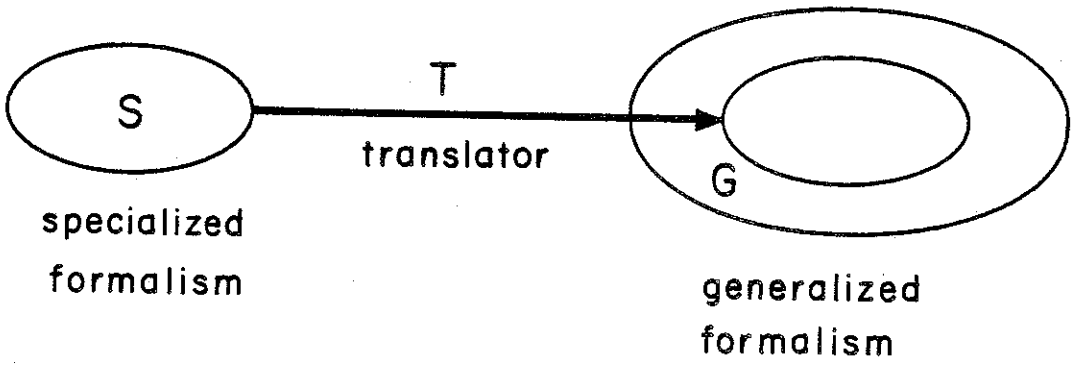


Figure 1. Mapping a specialized formalism into a generalized one.

sentence = <constant; expression>

expression = constant/constant*X/expression+expression/
(expression)(expression)

constant = 0, 1, 2, 3, ...

So for example,

e_1	3
e_2	10*X
e_3	7*X+8
e_4	(7*X+8)(10*X)
e_5	3+(7*X+8)(10*X)

are expressions in S, and

S_1	<5; 7*X+8>
S_2	<100; 3+(7*X+8)(10*X)>

are sentences in S.

Now consider another language, call it G, for generalized formalism. Objects in G are tables of the following form:

1	constant
2	constant
3	constant
.	.
.	.
.	.
constant	constant

where constant=0,1,2,3,... so for example,

	t_1
1	3

	t_2
1	7
2	6
3	5

and

	t_3
1	2
2	4
3	6
4	8
5	10
6	12
7	14
8	16
9	18
10	20

are objects of G.

Next consider a translator T that takes sentences in S and produces tables in G. When receiving a sentence <n; expression> in S, T does the following:

```
Declare TABLE as a one-dimensional array of size n
For X=1 to n in steps of 1
     LET TABLE(X)=expression/2
STOP
```

(i.e., expression is evaluated as an arithmetic expression with the variable X being given successive values from 1 to n).

We shall ignore any distinction between the array "TABLE" and the graphically presented "table" so that for example

<3, 2*X+6> is translated into

	t_4
1	4
2	5
3	6

Now let us make the following observations:

1) The meaning of a sentence in S is not apparent until the translator T and the language G are given. Once this happens, the meaning of a sentence is the table it is translated into,or as we shall say, it specifies. Thus <3; 2*X+6> specifies t_4 above and this table is its meaning.

2) Language G is more general than S because it accomodates many more tables than S can specify. Indeed, a sentence in S can specify only monotonically increasing or all-zeros tables, while G accomodates other tables such as t_2 above.

3) Not every string of characters makes sense in S. That is, it must first of all conform to the syntax of S. Once it is verified that a string is a sentence of S, it may still be meaningless because it fails to specify an object in G. For example, <3; X+5>, although syntactically valid in S is meaningless because it specifies a table

1	3
2	3.5
3	4

which is not a syntactically valid object in G (3.5 is not an integer constant).

4) Some sentences in S may have the same meaning. For example, <3; 7*X+8> and <3; 8+7*X> specify the same object in G.

More generally, for every notion of equivalence of objects in G, there is an induced notion of equivalence in S. For example, let tables in G be regarded as equivalent if one is a scaled version of the other. Then, <3; 7*X+8> and <3; 14*X+16> are equivalent sentences in S (since

the entries in the second table are twice those in corresponding places in the first).

This example illustrates that:

- the semantics (meaning) of objects in one language may be given by the syntax of a second target language,
- the target language may be more general in that it contains more objects than can be specified by the source language,
- not every syntactically correct object in the source need be given a meaning in the target language, and
- equivalence in the source is induced by equivalence in the target.

These ideas are at the base of the following discussion.

3. SKELETON OF THE HIERARCHY

Our basic approach is to construe models as shorthand means of specifying objects called systems. These various shorthand means are organized into a hierarchy [4] whose skeleton is depicted in Figure 2. As one proceeds from the left to right one moves from more specialized to more generalized formalisms. The most generalized formalism is one whose objects are systems. The most specialized formalisms shown (although these are not the most specialized possible by any means) are those called next event, activity scanning, process interaction, etc.

The hierarchy is such that at every level (except the highest) an object, if meaningful in a formalism, specifies a unique object in a next level formalism. The relationship between these two objects is exactly analogous to that between sentences of S and tables of G in our example. But note that this relationship repeats at every level so that ultimately every meaningful object specifies a system - indeed we call

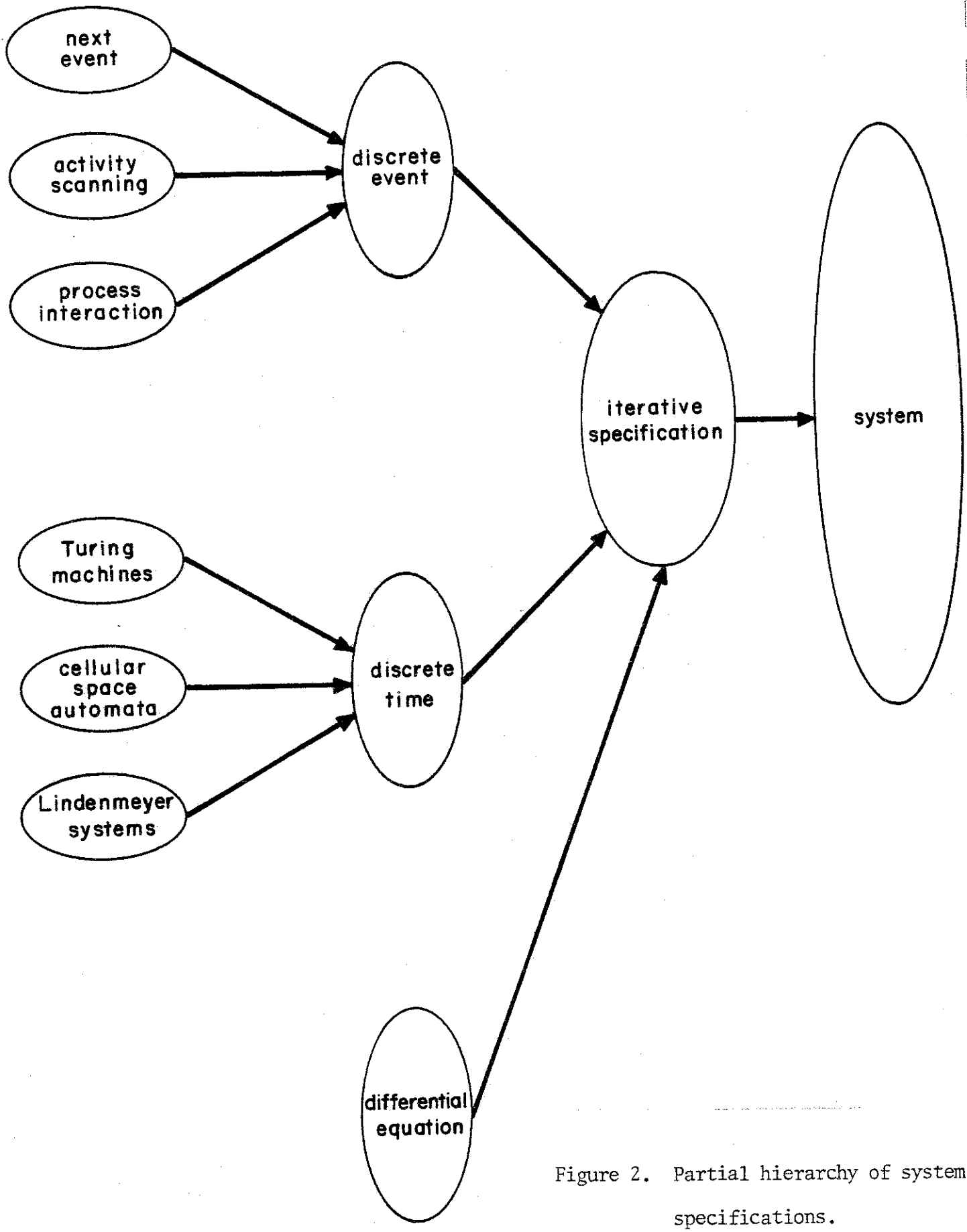


Figure 2. Partial hierarchy of system specifications.

these objects system specifications. Note that we do not formally associate a meaning with a system - it is supposed to be understood, once it is explicated in that ultimate meta-language, English.

Underlying the arrows in Figure 2 are the translators that assign higher level objects to lower level objects. Each formalism has its translator whose rules we call background conventions. These are conventions that are in effect once the formalism is stated. For example, a background convention for the next event formalism is that the time base is continuous. To repeat, without such background conventions, the objects in a formalism are meaningless quantities.

3.1 Construction of the Hierarchy

To understand how these formalisms and background conventions get set up, consider the problem of concisely representing the models in some class of interest. Take for example, next event models. That such a class of models has been recognized indicates that they share many properties in common. Recall that our approach is to consider the models as systems specifications. Thus, if we can ferret out all those properties shared by the class of systems, so specified, these become coded as background conventions. Whatever information is left over is that which uniquely distinguishes systems in the class, one from another. This information is what is coded in the formalism. Another way of saying this is that the formalism is a parameterization of the class it specifies.

Now such a list of background conventions tends to be quite long and difficult for people to understand. So it becomes advisable to break up the translation process into stages, each of which has a shorter list of background conventions. The formalisms constructed as a byproduct of this decomposition also should have interest in themselves. Such an intermediate

formalism may represent a common meeting ground between distinct classes of systems which nevertheless share some recognizable properties in common. For example, next event, activity scanning and process interaction models are all discrete event models so it is natural to introduce a discrete event formalism as a common stage in the translation process for each of the three primary subclasses of discrete event models.

Recall, from our simple example, that a formalism and its background conventions may be chosen so that not every object is meaningful, i.e., not every model description in the formalism actually specifies a system. Why not choose the formalism and background conventions so that every object is meaningful? To use an implementable version of this question, why not shift the decision about the meaningfulness of a model description from "run time" where it may be detected during simulation to "compile time" where it may be detected as a syntactic error by analysis? The answer then becomes obvious: the shift may not be possible if the required decision is unsolvable or it may greatly complicate the syntax even if the decision is solvable.

For example, a discrete event model description may be illegitimate [4], i.e., may fail to specify a system because there it contains a non-terminating cycle of simultaneous events. Detecting such a cycle is not a solvable problem so there cannot be a formalism exactly describing all legitimate discrete event systems. On the other hand, rules can be given to prevent illegitimacy, but incorporating them may greatly complicate the syntax of the formalism. Even if such rules are easily formalized, obeying them may greatly complicate a model description or may even restrict the domain of meaningful models. (Our S language trivially illustrates such restriction, since by disallowing odd constants one elimanates meaningless sentences.) Our discussion further in the paper (Section 16) should make this clear.

3.2 Advantages of the Hierarchy

We can list some of the advantages of hierarchically structured model representations versus a universal language approach without such a structure. These are:

1) Each formalism offers a concise, "shorthand" means of model description. This compares to the longhand description we would have to provide directly in the universal language approach. The latter description would essentially consist of an unstructured mix of the information contained in the formal object at some level of the hierarchy plus all the background conventions compounded from there up to the top of the hierarchy. Of course in the hierarchical approach, the modeler must understand, or at least have access to, the background conventions which give exact meaning to his description at any level.

2) The hierarchy facilitates a structured approach to model equivalence. Since models are construed as system specifications, any equivalence on models defined in a formalism must ultimately be compatible with an equivalence on the systems they specify. We shall now explain the last remark. The most basic system equivalence is behavioral - two systems are equivalent at this level if they have the same set of input-output segment pairs. Successively stronger equivalences are defined [4] and form a hierarchy in themselves. (Actually transitive relations such as "simulates" or "is a model of" are considered more fundamental [4].) As illustrated in Figure 3, any equivalence defined in a formalism must be such that if two model descriptions are equivalent in the formalism they must also specify (behaviorally) equivalent systems. This is the compatibility referred to above.

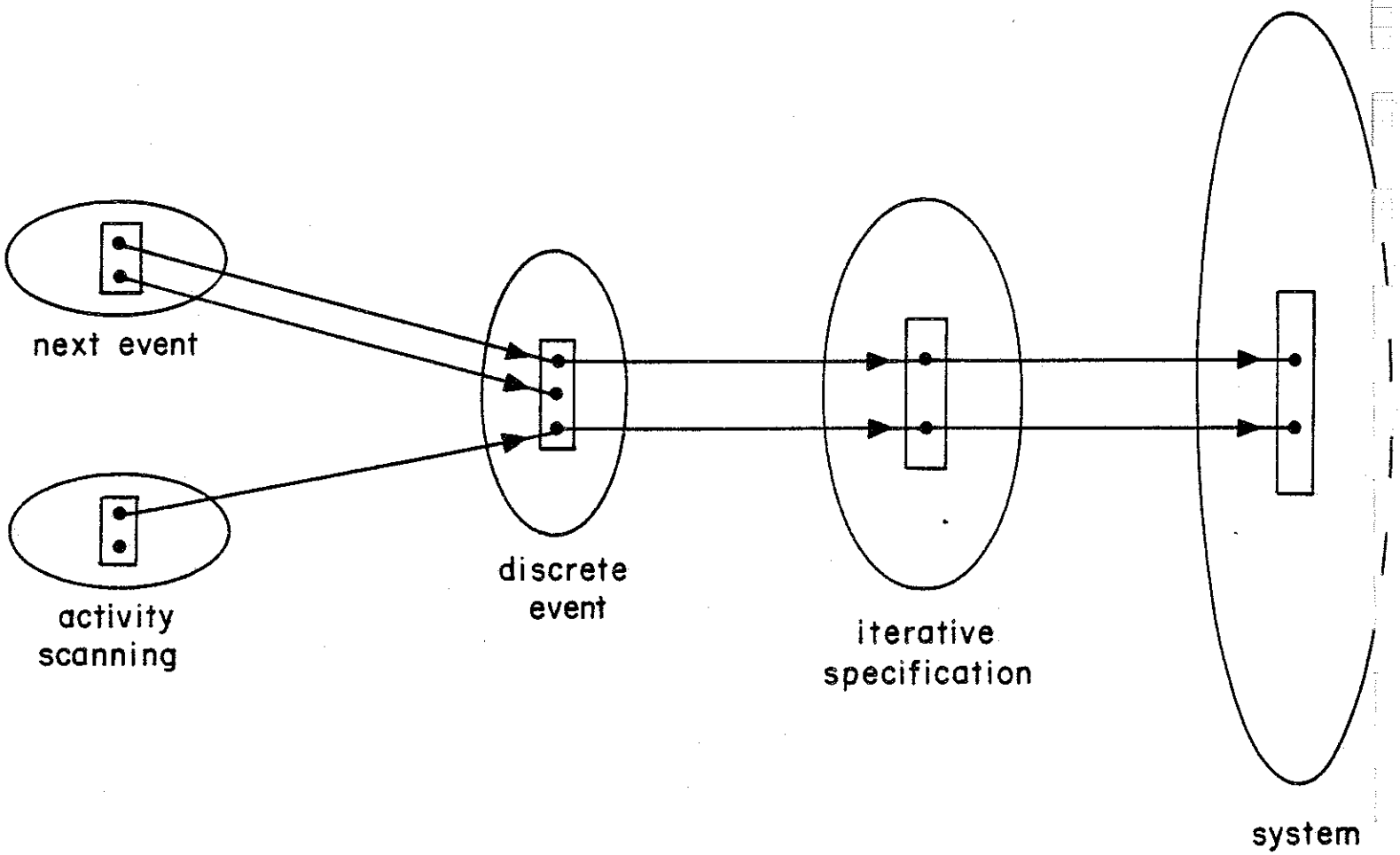


Figure 3. Equivalence of system specifications.

3) The more specialized a formalism, the more easily (e.g., in terms of computational complexity) can an equivalence defined in the formalism be checked. This is because two systems specified in the same formalism are already known to share the background convention properties in common. What remains to be checked is the similarity of the information encoded in the formalism.

4) Equivalence of models in two different formalisms can be discussed intelligibly at any higher level formalism in which both formalisms may be embedded. Indeed by 3) above, the optimal level is the lowest such common formalism. For example, to discuss the equivalence of next event and activity scanning models one has to deal at least with their interpretations as discrete event system specifications but hopes to not have to go to an even more general formalism. To consider equivalence cell space models and next event models one has to deal at least with their iterative specification since there is no lower common formalism.

5) A very natural definition of the representational power of a formalism emerges. The representation power of formalism is the set of systems specified by its model descriptions. However, we may also wish to consider a wider concept of power: the simulation power of a formalism is the set of systems which can be simulated by systems it can represent (i.e., by those in its representation power). Of course each such power is relative to the particular relation of "simulates" (e.g., the particular type of equivalence) employed.

We shall now demonstrate these ideas for the next event and activity scanning formalisms. The results on the equivalence of the two classes are of interest in themselves and serve to illustrate the above ideas.

4. CASE STUDY: NEXT EVENT AND ACTIVITY SCANNING FORMALISMS

In the following sections we shall treat the next event and activity scanning formalisms as a case study for our approach. Each formalism will be defined and its translation to the more general discrete event formalism will be given. The hierarchical decomposition is reflected in simulation schemes expressed in a language based on SIMULA but assumed to have greater capabilities. Limitations of SIMULA in this respect are discussed in the Appendix.

5. NEXT EVENT FORMALISM

A next event system specification is a structure of the form:

(INDEX.SET, COMPONENT.DESCRIPTIONS, SELECT)

where

INDEX.SET is a set of names for the COMPONENTS,

COMPONENT.DESCRIPTIONS is a family consisting of one COMPONENT.DESCRIPTION for each COMPONENT.

and

SELECT is a function (providing the breaking rules for simultaneously scheduled events).

A COMPONENT.DESCRIPTION is a structure of the form:

(TYPE,LOCAL.STATES,INFLUENCEES, TRANSITION.FUNCTION)

where

TYPE is an element of the set {ACTIVE,PASSIVE} (indicating whether the COMPONENT is active or passive).

LOCAL.STATES is a set (of states for the COMPONENT)

INFLUENCEES is a subset of INDEX.SET (giving the COMPONENTS directly influenced by this COMPONENT)

and

TRANSITION.FUNCTION is a function (specifying the effect of the COMPONENT on its INFLUENCEES (and itself) when it is activated)

The above is subject to the further restrictions to be discussed below. First

we shall use a shortening of notation. Let D be the INDEX.SET with $\alpha, \beta, \gamma, \cdots$ typical elements. Let $TYPE_\alpha$, S_α, I_α, and δ_α be the TYPE, LOCAL.STATES, INFLUENCEES, and TRANSITION.FUNCTION of COMPONENT α. Then the next event specification takes the short form

$$(D, \{Des_\alpha | \alpha \in D\}, SELECT)$$

where

$$Des_\alpha = (TYPE_\alpha, S_\alpha, I_\alpha, \delta_\alpha) .$$

The restrictions on Des_α, the COMPONENT.DESCRIPTION for α, are:

If $TYPE_\alpha$ = ACTIVE then

a) $\alpha \in I_\alpha$

b) δ_α maps $\underset{\beta \in I_\alpha(\text{active})}{\times} (S_\beta \times R^+_{0,\infty}) \times \underset{\beta \in I_\alpha(\text{passive})}{} S_\beta$ into itself

where I_α(active) and I_α(passive) are the influencees of α which are active passive, respectively.

Otherwise ($TYPE_\alpha$ = PASSIVE)

a) $I_\alpha = \emptyset$ (the empty set)

b) $\delta_\alpha = \emptyset$. (Thus, a PASSIVE component is characterized only by a state set S_α .)

The restriction on $(D, \{Des_\alpha\}$ SELECT) is that SELECT maps the non-empty subsets of D into D such that SELECT (subset) $\in$ subset.

To understand this formalism let us see how it can be realized in a SIMULA implementation. A PASSIVE component is one which can only be acted upon, and hence can be realized as a class object by the scheme shown in Figure 7.

For example, a queue, a typical passive component, might be generated from the declaration

```
passive component CLASS queue
begin
head CLASS local state; begin end;
ref (local state) s;
s:-new local state;
end;
```

A particular queue COMPONENT named α is generated by the statements:

```
ref (queue)α;
α:- new queue
```

and initialized as an empty queue.

A register which can be set externally into one of two states might be generated from

```
passive component CLASS binary register
                  :
CLASS local state; begin BOOLEAN B; end;
```

A counter on a checkerboard, incapable of initiating motion on its own, might be generated from

```
passive component CLASS counter
                  :
CLASS local state; begin integer  i , j  ; end;
                  :
                  :
```

A counter named α is generated by

```
ref (counter) α ;
α:- new counter;
```

and initially placed at the origin (0,0).

Notice that S_α specified by the component description Des_α is structured by {type declaration for local state} in the SIMULA scheme. More specifically, S_α is the range set of the variable α.s when α:- new passive type i . Thus $S_\alpha = A^*$ (where A is a set) , $S_\alpha = \{T,F\}$, and $S_\alpha = I$ (where I is the integers) in the three examples, respectively.

It should be clear from these examples that the state set S_α need not be a simple finite set but can be of arbitrarily complex structure.

Active, as opposed to passive, components are capable of initiating activity on their own, and so are fittingly realized as processes*. Refering to Figure 7, we see that each active type i specifies its own local state set just as does each passive type i . But in addition, an active component contains a real variable named σ and a ref(head) variable which will later hold a list of influencees. It also defines a procedure to realize its transition function. The real variable σ is automatically declared for any active component – it thus represents a background convention. It implements the α.TIME.LEFT.IN.STATE variable accorded active component α (Zeigler [4] page 145) to keep track of the time remaining until its next activation. Supposing that I_α the INFLUENCEES of α are α_i , i = 1,...,n then Figure 7 indicates how they would be assigned to α.influencees at beginning of run time.

The transition function procedure operates on a list, called temp(orary), of state pairs (s,σ) (as many as there are active influencees of α) and state singletons (s) (one for each passive influencee of α) to produce a new list of th

*actually because of the necessity to treat both passive and active components uniformly and due to the hierarchical nature of the class construct, passive components end up being processes also (see Appendix).

type. Thus it realizes a transformation of the type

$$\delta_\alpha((s_{\alpha_1},\sigma_{\alpha_1}),(s_{\alpha_2},\sigma_{\alpha_2}),\ldots,(s_{\alpha_A},\sigma_{\alpha_A}),s_{\alpha_{A+1}},\ldots,s_{\alpha_m})$$

$$= ((s'_{\alpha_1},\sigma'_{\alpha_1}),(s'_{\alpha_2},\sigma'_{\alpha_2}),\ldots,(s'_{\alpha_A},\sigma'_{\alpha_A}),s'_{\alpha_{A+1}},\ldots,s'_{\alpha_m})$$

specified in the COMPONENT.DESCRIPTION for α . The intended meaning is that when α is activated, it operates on the states of its influencees to change them from their present (unprimed) values to their new (primed) values.

6. INTERPRETATION OF THE NEXT EVENT FORMALISM

Indeed, it is this intended meaning which must be captured in the background conventions for next event model specifications. Before we describe these conventions formally, let us see how the model specification is given a SIMULA-like interpretation.

Figure 4 depicts the class structure of a scheme simulating next event models. Note that an active type i class is a subclass of active component which is a subclass of component, inturn a subclass of next event, inturn a subclass of process. An active type i object therefore has available all the attributes, procedures and code accorded to objects at these higher levels. In particular the code it gets from the next event class definition in Figure 6, carries out the interpretation of the information it provides in the form of local state, influencees and transition function. The next event code makes sense because of the additional procedures and code provided by the process class for filing, retrieving and interpreting next event notices on the SQS set (SIMULA's event notice timing set).

Looking at Figure 6 we see that when a next event cum active component is activated, it tests whether simulation should be continued. If so, it:

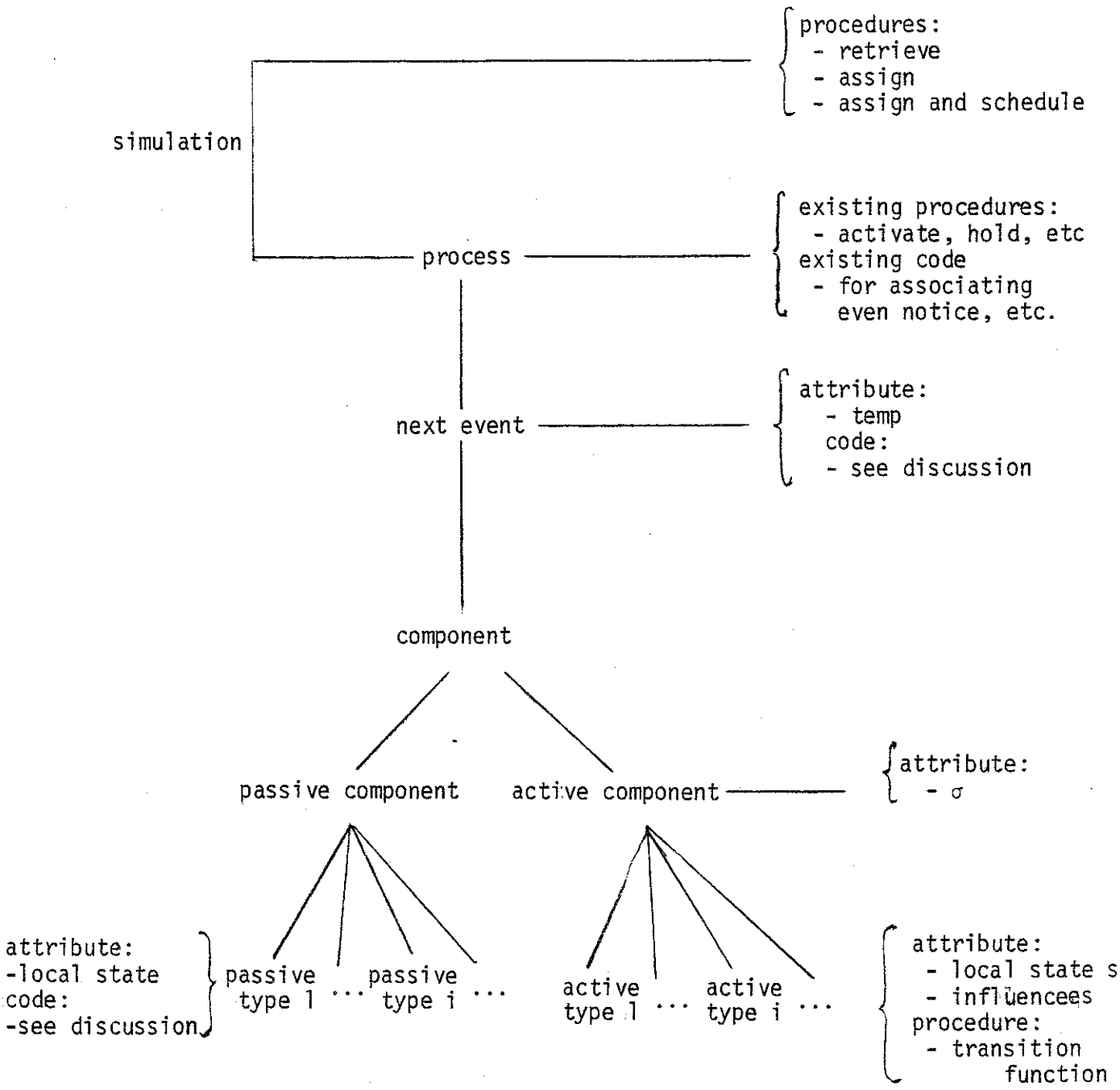


Figure 4. Class structure of Next Event Scheme

```
simulation
begin

linkage   class      state begin end;
 link     class      state singleton(s);ref(state)s;begin end;
 link     class      state pair(s,σ);ref(state)s;realσ; begin end;
 link     class      pointer (comp);ref(component)comp;begin end;


ref(head) procedure retrieve(component list);ref(head)component list;

comment: retrieves present states of components on component list;

begin ref(pointer)i;ref(component)comp;ref(head)temp;

      temp:-new head; i:-component list.first;
      while i=/= none do
            begin c:- i.comp;
               inspect c
                  when active component do
                           new state pair(s,σ).into(temp)
                  when passive component do
                           new state singleton(s).into(temp);
               c:-i. suc;
            end;
        retrieve:-temp;
end of retrieve;



procedure assign(temp,c);ref(component)c;ref(head)temp;

comment:assigns new state to c by looking up temp;

begin
     inspect temp.last
        when state singleton do c.s:=s
        when state pair do c.s =s;c.σ:=σ;
     temp.last:-none;
end of assign;



procedure assign and schedule(temp,component list);
                re(head)temp,component list;

comment:  assigns states to components
          on component list and then schedules if appropriate;

begin  ref(pointer)i;ref(component)c;

       i:-component list.last;
       while i.component=/=this component do
             begin c:-i.comp;
       assign(temp,c);
       If c is active then
       activate c delay c.σ;
       end;

end of assign and schedule;
```

Figure 5. Basic classes and procedures employed in next event simulation scheme

```
process class next event;
comment:interprets the component description;
begin while  t_i ≤ time  ≤ t_f do
     begin  ref(head)temp;
       temp:-retrieve(this active component.influencees);
       temp:-this active component.transition function(temp);
       assign and schedule(temp,this active component.influencee)
       assign(temp,this active component);
       hold(this active component.σ);
comment:in addition need to decrement σ's of non-influencees, not shown here;
     end;
end of next event;
```

Figure 6. Next event specialization of process

```
next event    class    component; begin end;
component     class    passive component; begin end;
component     class    active component;
                       begin real σ ; end;


passive component     class  {passive type i}
comment:  defines local state for component;
begin
     {type declaration for local state};
      ref(local state)s;
      s: - new local state;
      end;


active component {active type i};
comment:  defines component description except for influencees
          which must be enumerated at run time;
begin
     { type declaration for local state};
      ref (local state)s;
      ref (head) influencees;
      ref (head) procedure transition function (temp); ref (head) temp;
      begin
      comment:  transforms a list of state singletons and pairs into
                a list of the same form;
     {body}
      end of transition function;
end of active component;
```

Fig. 7. Subordinates of next event class

```
comment: define names (D) and associate a component with each;
ref(component) α1,α2,...,αA,αA+1,... αn;

     α1:-new active type i1;
          .
          .
          .
     αA:-new active type iA;

     αA+1:-new passive type iA+1;
          .
          .
          .
     αn:-new passive type in;

comment: define influencees of each component,
         each component is first on its own influencees list;
         new pointer (α1).into (α1.influencees);
                        .
                        .
                        .
comment: this completes model description now to initialize model state;
     α1.s:-initial state; α1.σ:-initial sigma ;
          .                        .
          .                        .
          .                        .
     αA.s:-initial state; αA.σ:-initial sigma;
     αA+1.s:-initial state;
          .
          .
          .
     αn.s:-initial state;

comment:schedule active component accordingly, for simplicity treat ∞ as large number
        rather than passivate;
    activate α1 delay  α1.σ;
          .
          .
          .
    activate αA delay αA.σ;

end of scheme;
```

Figure 8. End of model description and its initialization;

1. retrieves the present states of its influencees (using procedure retrieve, Figure 5),
2. applies its transition function to produce new values for these states,
3. sets its influencees (but not itself) into the new states just calculated and reschedules them for activation accordingly (using procedure assign and schedule, Figure 5),
4. sets itself into the new state calculated in 2. (using procedure assign, Figure 5),
5. reschedules itself for activation in the time calculated in 2.

Figure 8 shows how a particular model is set up and its simulation initialized. First the variables for the names of the components (the set D of the specification) are declared. Next for each name, an object of one of the active or passive types is generated and its reference assigned to the variable with the same name. Concerning the influencees, first note that components can be members of more than one influencees set (and if they are active can also be placed on the SQS). Since this is impossible in SIMULA, pointers (Figure 5) are setup to the components and these are placed as appropriate into the influencees of each active component. This completes the model specification.

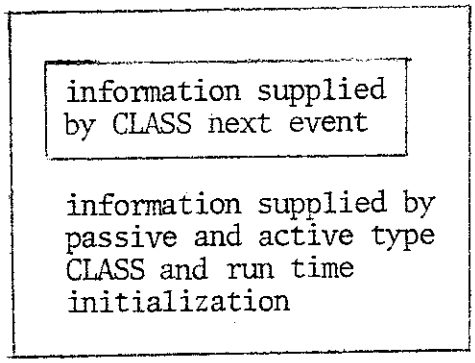
Next comes the state initialization. The local state variable of each component is set to a desired initial value. For active components, the same is done for the σ variables and then these components are scheduled for activation accordingly.

This completes the state initialization. From here on the simulation is initiated by SIMULA by activating the first process whose event notice is the first on the SQS. After executing the code of this process, i.e., the transition

function of the active component, SIMULA selects the event notice now first on the SQS (having removed the previous one), and so on.

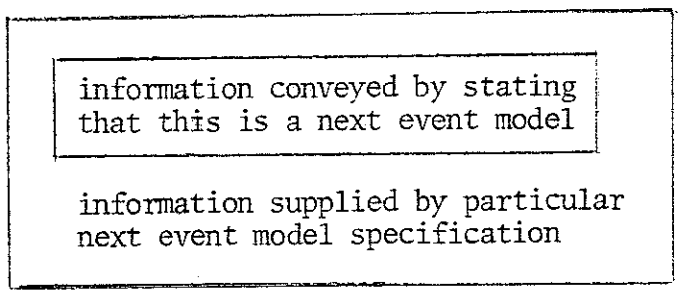
7. FORMALIZATION OF THE BACKGROUND CONVENTIONS FOR NEXT EVENT MODELS

We can readily decompose the information supplied to SIMULA by the next event simulation program in the following way:



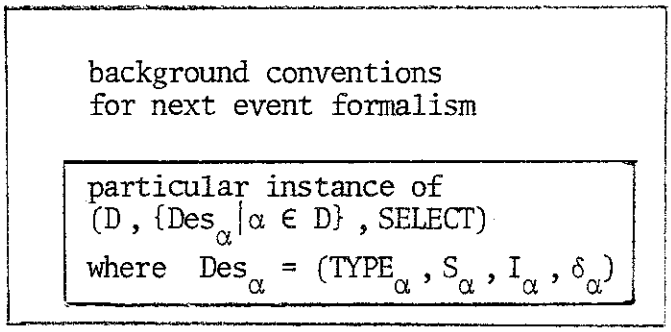
Indeed this decomposition is reflected in the class structure* of Figure 4 where the model-specific information is provided by classes (passive and active type) which are subordinate to the next event class, the latter supplying the more general information specific to next event models.

Now let us reinterpret the above diagram as follows:



*as noted before, certain constraints of SIMULA, which must be relaxed in order to fully realize this decomposition, are discussed in the Appendix.

We need both the inner and outer box information to simulate a particular next event model. If instead of simulating, we think of translating the next event formalism into the more general discrete event formalism, the diagram becomes:



The inner box information is taken by a translator which, using the outer box information, produces an instance of a discrete event model specification (recall Figure 4 and discussions in Section 6).

8. TRANSLATION OF NEXT EVENT INTO DISCRETE EVENT FORMALISM

Before discussing such a translator, we need to recall the definition of discrete event formalism. Since our discussion has been implicitly limited to autonomous (no external event) models this formalism takes the following form:

A discrete event system specification (DEVS) is a structure:

$$(S , \delta_\varphi , t)$$

where

S is a set (of sequential states)

δ_φ is a function (the autonomous transition function)

and

t is a function (the time advance function)

with the restrictions:

$$\delta_\varphi : S \to S$$

and $$t : S \to R^+_{0,\infty} .$$

The informal interpretation is that when arriving in state s the system will remain there a time $t(s)$ after which it immediately jumps to state $\delta_\varphi(s)$, stay for time $t(\delta_\varphi(s))$, etc. The formal interpretation is given by providing a translator to a higher level formalism, in this case, the iterative system specification, just as we now provide the translator for the next event to discrete event conversion.

Given a next event specification $(D, \{Des_\alpha\}, SELECT)$, we associate with it a unique DEVS (S, δ_φ, t) as follows:

$$S = \underset{\alpha \in D(active)}{\times} (S_\alpha \times R^+_{0,\infty}) \times \underset{\alpha \in D(passive)}{S_\alpha}$$

with typical element $s \in S$ of the form

$$s = ((s_{\alpha_1}, \sigma_{\alpha_1}), \ldots (s_{\alpha_A}, \sigma_{\alpha_A}), s_{\alpha_{A+1}}, \ldots s_{\alpha_n}) .$$

The map $t : S \to R^+_{0,\infty}$ is simply defined by

$$t(s) = \min\{\sigma_{\alpha_i} | \alpha_i \in D\}$$

with s above.

To define $\delta_\varphi : S \to S$ we first need the function

$$IMMINENT : S \to \text{subsets of } D$$

defined by

$$IMMINENT\ (s) = \{\alpha_i | \sigma_{\alpha_i} = t(s)\} .$$

Now let

$$\overline{\alpha}(s) = \text{SELECT}(\text{IMMINENT}(s))$$

and let

$$\delta_{\overline{\alpha}(s)}((s_{\beta_1},\sigma_{\beta_1}),\ldots,(s_{\beta_A},\sigma_{\beta_A}),s_{\beta_{A+1}},\ldots,s_{\beta_m})$$

$$= ((s''_{\beta_1},\sigma''_{\beta_1});\ldots,(s''_{\beta_A},\sigma''_{\beta_A}),s''_{\beta_{A+1}},\ldots,s''_{\beta_m})$$

where

$$I_\alpha(\text{active}) = \{\beta_1,\ldots,\beta_A\}$$

and

$$I_\alpha(\text{passive}) = \{\beta_{A+1},\ldots,\beta_m\} .$$

Then

$$\delta_\varphi(s) = s' = ((s'_{\alpha_1},\sigma'_{\alpha_1}),\ldots,(s'_{\alpha_i},\sigma'_{\alpha_i}),s'_{\alpha_{i+1}},\ldots,s'_{\alpha_n})$$

where, if $\beta \in D$ (active) then

$$(s'_\beta,\sigma'_\beta) = \begin{cases} (s''_\beta,\sigma''_\beta) & \text{if } \beta \in I_\alpha(\text{active}) \\ (s_\beta,\sigma_\beta - t(s)) & \text{otherwise} \end{cases}$$

and if $\beta \in D$ (passive) then

$$s'_\beta = \begin{cases} s''_\beta & \text{if } \beta \in I_\alpha(\text{passive}) \\ s_\beta & \text{otherwise} \end{cases}$$

To explain: the translator recognizes that the global state of the model is a composite of the (local state, sigma) pairs of the active types together with local states of the passive types. The time advance function embodies the

fact that the model will stay in the same global state until the time of the next event. At this time, of the possibly several imminent events (components contending for activation), one will be chosen by SELECT. (In the SIMULA implementation all of this is inherent in the simulation process constructs.) The selected component then carries out its activity and its effect is captured in the global transition function. (In the SIMULA program, this action is carried out under control of the next event code and the transition function code supplied by the activated object.) Actually, model-specified tie breaking is not readily available in SIMULA and we have not implemented SELECT in the SIMULA program.)

Note that the translation process embodies part, but not all, of background conventions for next event models. For example, the presence of the time-left (σ) variables and their proper interpretation appear as background conventions, but the fact that next event models operate in continuous time does not. Only background conventions necessary for translating to the discrete event formalism are employed. Other background conventions shared by all discrete event models appear at the next level – the translation of the discrete event formalism into the iterative system specification.

We shall now present an activity scanning formalism together with its translation to the discrete event formalism.

9. ACTIVITY SCANNING FORMALISM

The formalism to be discussed is called combined event-oriented-activity-scanning in Zeigler [4] since it includes event scheduling capability. We shall later discuss the relative powers of the activity scanning and next event formalisms.

An activity scanning system specification is a structure of the form:

(INDEX.SET, COMPONENT.DESCRIPTIONS, SELECT)

where these elements have the same definitions as in the next event formalism, except that:

A COMPONENT.DESCRIPTION is a structure of the form

(TYPE, LOCAL.STATES, INFLUENCEES, INFLUENCERS, ACTIVATION.CONDITION, ACTION.FUNCTION)

where TYPE, LOCAL.STATES and INFLUENCEES have the same definitions as before, and

INFLUENCERS is a subset of INDEX.SET (giving the set of COMPONENTS on which the ACTIVATION.CONDITION is defined)

ACTIVATION.CONDITION is a predicate (on the LOCAL STATES of the INFLUENCERS of the COMPONENT which tests whether it is activateable)

and ACTION.FUNCTION is a function (specifying the effect of a COMPONENT on its INFLUENCEES (and itself) when it is activated).

In short form, an activity scanning model is specified by

$$(D, \{Des_\alpha | \alpha \in D\}, SELECT)$$

where

$$Des_\alpha = (TYPE_\alpha, S_\alpha, I_\alpha, J_\alpha, C_\alpha, f_\alpha)$$

The restrictions on the above are the same as for the next event formalism except that:

If $TYPE_\alpha$ = ACTIVE then

a) $\alpha \in I_\alpha$

b) $C_\alpha: \times_{\beta \in I_\alpha} S_\beta \rightarrow \{TRUE, FALSE\}$

c) f_α: $\underset{\beta \in I_\alpha \cup J_\alpha \text{(active)}}{\times} (S_\beta \times R_{0,\infty}) \underset{\beta \in I_\alpha \cup J_\alpha \text{(passive)}}{\times} S_\beta \rightarrow \underset{\beta \in I_\alpha \text{(active)}}{(S_\beta \times R_{0,\infty})} \underset{\beta \in I_\alpha \text{(passive)}}{\times} S_\beta$

Otherwise ($TYPE_\alpha$ = PASSIVE)

a) $I_\alpha = \varphi$

b) $C_\alpha = \emptyset$

c) $f_\alpha = \emptyset$.

Our SIMULA-like scheme for this formalism is the same as for the next event formalism except that the activity scanning class and its subclasses replace those of the next event class. Figure 10 displays how each active type α declares its own attributes corresponding to Des_α . Figure 9 shows how this information is interpreted in the activity scanning mode. Here we see that when a component becomes activated in the SIMULA sense (as first on the SQS and begins execution) it proceeds to test its ACTIVATION.CONDITION. Looking at the wait until procedure definition (from Franta [6]), we see that if the condition is true it proceeds to the next action statement. Otherwise it passivates and is reawoken to test this condition after every event. If, and when, the wait until is traversed, the ACTION.FUNCTION of the component is applied in a manner similar to the application of the transition function of the next event simulation. Note that the activation condition is tested on the influencers only, while the action function has an affect only on the influencees.

Indeed, a component's influencer and influencee sets may often, but need not necessarily be, distinct. For example, the activation of a processor may depend on the input material and the requisite resources being available (the influencers) while its products are sent to other processors (its influencees).

```
process class activity scanning;
comment: substitutes for Figure 6;
   begin ref(head)neighbors;
         neighbors:-union(this component.influencees,this component.influencers);
         while t_i ≤ time ≤ t_f do
     begin ref(head)temp;
     wait until (this component.activation condition,this component.influencers);
     temp: - retrieve (neighbors);
     temp: - this component.action function(temp);
     assign and schedule (temp,this component.influencees);
     assign(temp), this component);
     hold(max(0,this component.σ));
     end;
     procedure wait until (B,component list);
               ref(boolean procedure)B;ref(head)component list;
      comment: tests B on component list, if false
               places component on wait list to be
               reactivated by monitor; if true,
               returns control to next statement.
   begin ref(head)temp;
L:    comment:-retrieve(component list);
      if B (temp)then go to exit;
           as in Franta, pages 189-190
        go to L;
exit:end of wait until;
end of process;
```

Figure 9. Activity scanning specialization of process

```
activity scanning class component; begin end;
                    .
                    .
                    .
          same as next event
                    .
                    .
                    .
active component {active type i}

begin

          {type declaration for local state};
           ref(local state)s;
           ref(head)influencees, influencers;

           ref(head)procedure action function(temp);
                                    ref(head)temp;

              begin ... end;

           boolean procedure activation condition(temp);
                                          ref(head)temp;

           begin ... end;
```

Figure 10. Subordinates of activity scanning class

10. TRANSLATION OF ACTIVITY SCANNING INTO DISCRETE EVENT FORMALISM

As in the next event case, the next event formalism is given formal meaning by translation into discrete event formalism. This is done as follows:

Given an activity scanning specification $(D, \{Des_\alpha\}, SELECT)$, we associate with it a unique DEVS (S, δ_φ, t) as follows:

$$S = \underset{\alpha \in D(active)}{\times} (S_\alpha \times R_{0,\infty}) \times \underset{\alpha \in D(passive)}{S_\alpha}$$

To define $t: S \to R^+_{0,\infty}$ we first define

$$FUTURE(s) = \{\alpha | \sigma_\alpha > 0\}$$
$$PRESENT(s) = \{\alpha | \sigma_\alpha = 0\}$$
$$PAST(s) = \{\alpha | \sigma_\alpha < 0\}$$

where

$$s = ((s_{\alpha_1}, \sigma_{\alpha_1}), \ldots, (s_{\alpha_A}, \sigma_{\alpha_A}), s_{\alpha_{A+1}}, \ldots, s_{\alpha_m}) \in S .$$

Contenders for activation are

$$ACTIVATEABLE(s) = \{\alpha | C_\alpha(s) = TRUE \text{ and } \alpha \in PRESENT(s) \cup PAST(s)\} .$$

There are three cases:

1. $ACTIVATEABLE(s) \neq \emptyset$
2. $ACTIVATEABLE(s) = \emptyset$ and $FUTURE(s) \neq \emptyset$
3. $ACTIVATEABLE(s) = \emptyset$ and $FUTURE(s) = \emptyset$

Then

$$t(s) = \begin{cases} 0 & \text{case 1} \\ \min\{\sigma_\alpha | \alpha \in FUTURE(s)\} & \text{case 2} \\ \infty & \text{case 3} \end{cases}$$

The selected component is

$$\bar{\alpha}(s) = \begin{cases} \text{SELECT(ACTIVATEABLE(s))} & \text{case 1} \\ \text{SELECT(IMMINENT(s))} & \text{case 2} \end{cases}$$

where

$$\text{IMMINENT}(s) = \{\alpha | \delta_\alpha = t(s)\} .$$

To define $\delta_\varphi : S \to S$, we note that $\delta_\varphi(s) = s'$ where s' is the result of applying $f_{\bar{\alpha}(s)}$ to the INFLENCEEs of $\bar{\alpha}(s)$ obtained in the same manner as was the effect of δ_α in the next event case.

11. RELATIVE POWERS OF NEXT EVENT AND ACTIVITY SCANNING FORMALISMS

It is quickly seen that the power of the activity scanning formalism includes that of the next event formalism. Indeed by restricting the σ variables to non-negative values, the ACTIVATION.CONDITIONs to tautologous (always TRUE) ones and the INFLUENCERs to empty sets we regain the next event formalism from the activity scanning one. The converse direction is the difficult one: it asks the question; are there activity scanning models not representable, or simulateable by, next event models?

As a prelude to discussion of this issue we shall present a more efficient and conceptually pleasing scheme for activity scanning simulation. Examination of this scheme will make clear how to convert an activity scanning specification to next event form.

12. THE EFFICIENT ACTIVITY SCANNING SIMULATION SCHEME

The scheme of Figure 9 is inefficient because it continues to retest an

```
process class activity scanning;

comment: efficient version taking advantage
         of declared model structure;

begin ref(head) neighbors,*affectees;
      boolean due;
      neighbors:-union(this component.influencees,
                       this component.influencers);
      affectees* :-{β | this component.inluencees
                     ∩ influencers.β =/= none}

      while t_i ≤ time ≤ t_f do
      begin ref(head)temp;
            wait until(this component.activation
                  condition, this component.
                        influencers);
            temp:-retrieve(neighbors);
            temp:-this component.action function (temp);
            assign and schedule (temp,this component.influencees );
            assign (temp,this component);
            *wake (affectees);
            hold(max[0,this component.σ]);
      end;

procedure wait until (B,component list;
                ref(boolean procedure)B;ref(head) component list;

*comment: if B is FALSE, due is set to TRUE and this component
                                passivates (σ:=∞);

      begin ref(head)temp;
L:          temp:-retrieve(component list);
            if B(temp) then go to exit;
            * due:=TRUE
            * passivate;
              go to L;
exit:       * due:=FALSE;
      end of wait until;

*procedure wake (component list);ref(head)component list;

*comment: wakes up due affectees;
*begin ref(component)c;
            *for c:-component list.first,c.suc, while c=/= none do
            *if c.due and c.σ=∞then activate c after current;
*end of wake;
```

Figure 11. Modifications (indicated by asterisks) for efficient activity scanning.

activation condition even when the truth status of the condition could not have changed since the last test. Indeed, the status of C_α cannot have altered unless one of the influencers, say β, of γ has changed states.

But such a change can be caused only by an activation of some component, α say, which influences such a β. In other words, let us define

$$\begin{aligned} \text{AFFECTEES}(\gamma) &= \{\alpha | J_\alpha \cap I_\gamma \neq \emptyset\} \\ &= \{\alpha | \text{ there is an influencer of } \alpha \text{ which is} \\ &\qquad \text{also an influencee of } \gamma\} . \end{aligned}$$

Then AFFECTEES(γ) is the set of components α whose truth states can be changed by an activation of γ. For example, let processor α depend on availability of resource β for its activation. Then $\beta \in J_\alpha$ (β is an influencer of α). Now let processor γ produce resource β. Then $\beta \in I_\gamma$ (β is an influencees of γ). Then since $\beta \in J_\alpha \cap I_\alpha$ we have $\alpha \in$ AFFECTEES(γ) i.e., when γ is activated it may change the truth status of C_α (it may produce resourse β for which α is waiting).

The basis for an improved simulation is now clear: when C_α is first tested and found to be false, it should be laid aside only to be reawakened for testing when some component γ is activated and finds α as one of its affectees. The modifications which implement this scheme are shown in Figure 11. Note that in the modified wait until procedure, a boolean variable DUE records the fact that the activation condition has been found false and the component has been passivated. On the other hand, when the activation condition is found to be true, the component goes on to apply its action function, then before rescheduling itself, it wakes up its affectees. Actually, note that in procedure wake, only those affectees whose DUE variable is TRUE are activated for condition testing. It would be wrong to activate an affectee which had been passivated, or indeed otherwise suspended,

under normal model control. For example, if processor α in the above example has passed its activation test and has begun processing, its status on the SQS should not be altered because processor γ makes resource β available. Thus, the DUE variable is a necessary addition to each active component's realization for correct implementation of the scheme.

This simulating strategy is conceptually satisfying because it takes advantage of the underlying structure of the model. But all this requires that the modeller be able to formulate the structure correctly. Essentially he must supply the correct sets of influencers and influencees (this could be done under computer assistance). Over-estimating these sets will lead to a degradation of time efficiency since the gain in the latter depends on these sets being relatively small (an analysis similar to that in Zeigler [5] applies here).

Let us note now that our simulation scheme has effectively transformed the activity scanning model into next event form: Why? Because all condition testing is essentially brought about by direct scheduling. If we hide the activation condition within a next event transition function, no external observer could report any activity scanning behavior (Fig. 12). Note however, that he will see a lot of immediate (delay 0) activations, and this is a topic we shall return to in Section 16.

We can formally express the transformation implicit in improved simulation strategy as follows:

Given an activity scanning model specification $(D, \{Des_\alpha\}, SELECT)$ with $Des_\alpha = (TYPE_\alpha, S_\alpha, I_\alpha, J_\alpha, C_\alpha, f_\alpha)$ we associate with it a next event specification $(D'_\alpha, \{Des'_\alpha\}, SELECT')$ with $Des'_\alpha = (TYPE'_\alpha, S'_\alpha, I'_\alpha, \delta'_\alpha)$ where

```
ref(head) procedure transition function (temp);
                              ref(head)temp;

begin ref(pointer) first affectee, last affectee,
                                   last influencee;
      ref(state pair) first affectee pair;

     if not B(temp) then
     begin due:=FALSE

           temp:-action function (temp);
           x:-first affectee.pair;
           for i:-first affectee, i.suc,
                  while i=/= last affectee do

           begin c:-i comp;
                 if x.local state.due and x.σ=∞
                                   then c.σ:=0;

                    x:-x.suc;
           end;
     end;
     else begin real t;
          i:-influencees.first;
          x:-temp.first;
          x.local state.due:=TRUE;
          t:=x.σ;
          x.σ=∞;
          for  i:-suc,while i=/= last influencee  do
          begin x:-x.suc;
          x.σ:=x.σ-t;
          end.
          end;

    end of transition;
```

Figure 12. The next event version transition function for simulating activity scanning model.

$D' = D$

$SELECT' = SELECT$

$TYPE'_\alpha = TYPE_\alpha$

$S'_\alpha = S_\alpha \times \{TRUE, FALSE\}$ (add the Due_α variable to the local state of α)

$I'_\alpha = I_\alpha \cup J_\alpha \cup AFECTEE(\alpha)$ (add the influencers and the affectees to the influencees of α)

and δ'_α is defined by the following scheme:

1. If C_α is false, set Due_α = TRUE , set $\sigma_\alpha = \infty$ (passivate) and do not affect other influencees.

Otherwise (C_α is true):

2. Set Due_α = false;
3. Apply $\overline{f}_\alpha$ to obtain the next states of the I_α subset of I'_α ;
4. For each affectee β , if Due_β is TRUE and $\sigma_\beta = \infty$ then set $\sigma_\beta = 0$ (activate immediately) otherwise do not affect it.

Here $\overline{f}_\alpha$ is f_α modified as follows: $\overline{f}_\alpha$ examines its input arguments. Any argument of the form $(s_\beta, \infty, TRUE)$ (for $\beta \in I_\alpha$) indicates to it that β is due and it converts this to $(s_\beta, 0)$ for input to f_α . Otherwise the due argument is ignored for input to f_α . If the result of applying f_α is make $\sigma_\beta = \infty$ (the activity scanning model changes β from due to passive), then Due_β is set to FALSE. In this way passivated components are kept distinct from due components.

Let us refer to the associated next event specifications as the "event version" of the activity scanning specification. Then there remains the question: are the activiting scanning specification and its event version equivalent in any way?

The only way to discuss any such equivalence is to consider the translations of the two specifications into a common higher level formalism. In fact, the

discrete event formalism will do for our present purposes.

13. THE UNRESTRICTED ACTIVITY SCANNING FORMALISM HAS GREATER (BUT UNFAIR) POWER

Let us first note that as formulated, the activity scanning models have σ variables which can take on negative values, whereas next event models do not. Non positive values signal the due status, this being explicitly represented in the event version. However, this range can also be cleverly used for other purposes. We have for good reason not allowed the activation condition to depend on the σ variables*. However, we have allowed the action function to employ the σ variables. Through such means, activity scanning models can keep track of elapsed time (e.g., time elapsed since component entered due status) which we have not allowed next event models to do. This does not represent a true increase in power since we could have outfitted next event models with additional variables, which increase with time, and afforded them the same extra power.

13.1 Unrestricted Activity Scanning Formalism

To retain the spirit of the comparison (which lies in the scanning/scheduling trade off), we shall restrict the activity scanning formalism so that clever use cannot be made of the σ variables. We add the following:

---

*Allowing a condition to depend on time and requiring activation at its first satisfaction can lead to meaningless (ill defined) models (cf. Franta, page). Per our discussion (Section 3.1), we choose to obviate this problem by the above restriction.

13.2 Additional Restriction on Des_α

The action function f_α remains unchanged when any of its negative σ arguments are changed to 0 .

With this restriction, we reduce any activity scanning model to canonical form in which the σ variables have range $R^+_{0,\infty}$ (rather than $R_{0,\infty}$). To retain their role in recording due status, we must interpret the subtraction $\sigma - t(s)$ in the translation to DEVS (Section 10) as resulting in 0 for $\sigma - t(s) \leq 0$. We shall rely on intuitive acceptance of the equivalence of the original and reduced versions although it can easily be established formally in the manner of the following discussion.

Another non-essential restriction we shall impose is to require the SELECT function to be generated from a linear order on the index set (see Zeigler [4], page 147). This is standard practice in simulation language tie breaking rules. Actually, we shall require that the following be true:

13.3 Additional Restriction on SELECT

Let $subset_1 \subseteq subset_2 \subseteq D$. If $\alpha = SELECT(subset_2)$ and $\alpha \in subset_1$ then $\alpha = SELECT(subset_1)$.

In other words, a component cannot lose rank by removal of contenders. It can be shown that this innocuous requirement is equivalent to SELECT always choosing the highest element in an input subset according to a fixed linear order.

The above constitute the restricted activity scanning formalism we shall henceforth be dealing with.

14. SIMULATION OF ACTIVITY SCANNING MODEL BY ITS EVENT VERSION

Let Let $M = (S,\text{ŧ},\delta_\varphi)$ be the DEVS associated with the (restricted) activity scanning specification and $M' = (S',\text{ŧ}',\delta'_\varphi)$ be the DEVS associated with its eventversion. We shall establish that M' simulates M under the following definition of "simulates":

M' <u>simulates</u> M if there is a subset $\overline{S} \subseteq S'$ and a map $h : \overline{S} \overset{\text{onto}}{\to} S$ such that for each $s' \in \overline{S}$, either

1. $\text{ŧ}'(s') = \text{ŧ}(h(s'))$ and $h(\delta'_\varphi(s')) = \delta_\varphi(h(s'))$

or

2. $\text{ŧ}'(s') = 0$ and $h(\delta'_\varphi(s')) = h(s')$.

This is a modified version of the DEVS morphism given in Zeigler [4] page 275. To explain: $\overline{S}$ is the subset of states which represent those of S ; line 1) states that in corresponding states M and M' have equal time advances and jump to corresponding states; line 2) states that if line 1) does not hold, it is because the event version has a zero time transition to a state which retains correspondence with the activity scanning state.

Now to define the correspondence map h :

Let $\overline{S} = \{s' | s' = ((s'_{\alpha_1},\sigma'_{\alpha_1},d_{\alpha_1}),\ldots,(s'_{\alpha_A},\sigma'_{\alpha_A},d_{\alpha_A}),s'_{\alpha_{A+1}},\ldots,s'_{\alpha_n})$

such that for $\alpha = \alpha_1,\ldots,\alpha_A$

$\sigma'_\alpha = \infty$ and $d_\alpha = \text{TRUE} \Rightarrow C_\alpha(s') = \text{FALSE}\}$.

Thus a legal simulating state is one in which a due, passivated component always has a false activation condition. The morphism requires implicitly that the simulator preserve this legality of simulating states. Then

$$h : \overline{S} \to S$$

is given by

$$h((s'_{\alpha_1}, \sigma'_{\alpha_1}, d_{\alpha_1}), \ldots, (s'_{\alpha_A}, \sigma'_{\alpha_A}, d_{\alpha_A}), s'_{\alpha_{A+1}}, \ldots, s'_{\alpha_n})$$

$$= (s'_{\alpha_1}, \sigma_{\alpha_1}), \ldots, (s'_{\alpha_A}, \sigma_{\alpha_A}), s'_{\alpha_{A+1}}, \ldots, s'_{\alpha_n})$$

where

$$\sigma_\alpha = \begin{cases} 0 & \text{if } \sigma'_\alpha = \infty \text{ and } d_\alpha = \text{TRUE} \\ \sigma'_\alpha & \text{otherwise .} \end{cases}$$

Note $h(s')$, the activity scanning state represented by event version state s', is "almost" identical to s'. The exceptions are that: s' carries extra baggage — the Due variables, and these make a difference only in the case of a due, passivated component, where $\sigma'_\alpha = \infty$ must be interpreted as $\sigma_\alpha = 0$.

Now we sketch the proof that h is a morphism as defined. Let $s' \in \overline{S}$ be a simulating state and $s = h(s')$ the state it represents. The following cases arise

1. <u>Some α is selected for immediate activation but has a false activating condition</u>

This occurs when $\sigma'_\alpha = 0$, $\alpha = \text{SELECT}(\text{IMMINENT}(s'))$ and $C_\alpha(s') = \text{FALSE}$. By definition of h , we have also $\sigma_\alpha = 0$ and $C_\alpha(s) = \text{FALSE}$ (h preserves local states). Applying δ_α, yields $\sigma'_\alpha = \infty$ and $\text{Due}_\alpha = \text{TRUE}$, so the resulting state $\delta_\varphi(s')$ still corresponds to s . Since $t'(s') = 0$, line 2) holds. Also $\delta_\varphi(s') \in \overline{S}$.

In other words, in this case all that happens is that α is passivated and Due_α is set to TRUE accordingly. Correspondence and legality are maintained.

2. Some α is selected for immediate activation with a true activation condition

As above, $\sigma'_\alpha = 0$ and $\alpha = \text{SELECT}(\text{IMMINENT}(s'))$, but $C_\alpha(s') = \text{TRUE}$. Also $\sigma_\alpha = 0$ and $C_\alpha(s) = \text{TRUE}$. So in M, $\alpha \in \text{ACTIVATEABLE}(s)$. By our restriction on SELECT it is easy to show that $\alpha = \text{SELECT}(\text{ACTIVATEABLE}(s))$. Now by definition of δ_α, the effect of activating α in both M' and M is first to modify the influencees I_α according to action function f_α . This means that both s' and s; transit to corresponding states ($h(\delta'_\varphi(s')) = \delta_\varphi(h(s'))$) and since $t'(s') = t(h(s')) = 0$, line 1) is satisfied.

The legality requirement, $\delta'_\varphi(s') \in \overline{S}$ is more interesting here. Suppose in state $\delta'_\varphi(s')$, $\sigma'_\beta = \infty$ and $d_\alpha = \text{TRUE}$. Then in $h(\delta'_\varphi(s'))$, $\sigma_\beta = 0$. If $C_\beta(h(\delta'_\varphi(s')) = \text{FALSE}$ then so is $C_\beta(\delta'_\varphi(s'))$ and legality is automatic. However if $C_\beta(h(\delta'_\varphi(s')) = \text{TRUE}$ then so is $C_\beta(\delta'_\varphi(s'))$ and we must show this is impossible. Now it could only have been that $\sigma'_\beta = \infty$ and $d_\alpha = \text{TRUE}$ in the previous state s' , since δ'_α cannot create such a situation. By legality, $C_\beta(s')$ was FALSE. So C_β changed from FALSE to TRUE in the transition. But this is possible only if $\beta \in \text{AFFECTEES}(\alpha)$. And in that case, σ'_β would be set to 0 by δ'_α , i.e., $\sigma'_\beta = 0$ in $\delta'_\varphi(s')$ which contradicts $\sigma'_\beta = \infty$ in $\delta'_\varphi(s')$ as supposed.

Thus, both the correspondence in states and the legality of the next state are preserved by the simulator.

3. Some α is selected for future activation

The argument is similar to case 2, after establishing the $\text{ACTIVATEABLES}(s) = \emptyset$.

4. The simulator passivates

Here $\sigma'_\alpha = \infty$ for all α, so $t_L(s') = \infty$. It is then easy to show that for each α, either $\sigma_\alpha = \infty$ or $\sigma_\alpha = 0$ and $C_\alpha(s) = \text{FALSE}$. Therefore, $t_L(s) = \infty$ also and line 1) is satisfied (transition preservation is irrelevant in this case).

This completes the proof of the

Theorem

Let M and M' be the DEVS's associated with an activity scanning specification and its event version, respectively. Then M' simulates M via a modified DEVS morphism.

15. ON THE EQUIVALENCE AND NON EQUIVALENCE OF NEXT EVENT AND ACTIVITY SCANNING FORMALISM

Following the lines of Zeigler [4] page 275, it can be shown that if M' simulates M via a modified DEVS morphism then $S_{M'}$ simulates S_M where these are the systems that M and M' ultimately specify. Indeed, as discussed in Section 3.2, it is this fact that gives meaning to the "simulates" concept employed at the DEVS level. With this accepted, we can assert the

Theorem

Every (restricted) activity scanning model is simulateable by a next event version.

Since we already know that every next event model is representable as an activity scanning model, we have

Theorem

The simulation powers of the (restricted) activity scanning and next event

formalisms are equal.

But note that we cannot assert that the representation powers of these classes are identical. This is because while it is true that every next event specified system is also an activity scanning specified system, we have not shown the converse. To do the latter, it would have been sufficient that the above DEVS M and M' specified the same system. We have only established a homomorphic relation between them. Indeed we have the

Theorem

The representation power of the activity scanning formalism strictly includes that of the next event formalism.

In support of this theorem, we note that a next event specification cannot simulate an activity scanning specification over the same state space. This is because under the background conventions, $\sigma_\alpha = 0$ for some α always implies $t(s) = 0$ while we may have eventually $t(s) > 0$ under the activity scanning conventions. (See Zeigler [4] page 155.) (Indeed it is the presence of the due variable which allows us to overcome this limitation, at a cost of extra memory.)

16. THE PROPER NEXT EVENT FORMALISM

In this vein, let us note that immediate activation – setting $\sigma_\alpha = 0$, is crucial to the next event simulation of activity scanning. But in a fundamental sense this goes against next event philosophy, which is to predict ahead of time when an event will occur rather than be forced to continually test for its occurence. What we have done is to cleverly employ immediate activation as a means of rescheduling activation tests. What would happen if this were not allowed? In other words, what is the power of a SIMSCRIPT without the SCHEDULE NOW statement or of a SIMULA with ACTIVE and HOLD being allowed only non zero delays?

To employ another terminology, can state (conditional) events always be replaced by time (future scheduled) events?

First, let's place the appropriate restrictions on our next event formalism:

A next event specification is called proper (i.e., properly next event) if, for any $\alpha \in D$ and any $\beta \in I_\alpha$, whenever $\sigma_\beta = 0$ in the output of δ_α then $\sigma_\beta = 0$ in its input.

In other words, no component transition function can set a σ variable to zero (unless it was already zero).

Before proceeding we make some general observations.

As indicated in Section 2, not every describable object in a formalism need have a meaning. This is the case in the DEVS formalism where it is possible to write a syntactically correct DEVS which does not ultimately specify a system. As in Zeigler [4] page 237, a DEVS which does specify a system is called a legitimate and is characterized by the fact that its cummulative time advances always grow without limit. The most common reason for illegitimacy is that a model contains a cycle of transitory ($t_a(s) = 0$) states e.g., due to a recurrent series of immediate activations:

α activates β , β activates γ ,..., ζ activates α .

The legitimacy question is undecideable in general. But it is easy to see that proper next event models always specify legitimate DEV's*

Proof: At most a finite number of components are imminent. Time will stand still while components are selected from this set, but eventually time will resume its increase, since the set always contracts.

---

*we limit this statement to computeable models i.e., those having at most a finite number of activated ($\sigma_\beta \neq \infty$) components at any time.

Thus imposing the propriety restriction is an easy way to assure meaningfulness. But do we have to pay the price in reduced power for the proper next event formalism, or in increased model description complexity? The answer is given by the

Theorem

The representation power of the proper next event formalism equals that of the next event formalism. However, this remains true only as long as one permits arbitrary computational complexity in the transition functions.

Note that according to the theorem, activity scanning is simulateable not only by next event, but also by proper next event, models. In these terms, the proof of the theorem shows how to replace testing by prediction at the cost of greatly increased computation.

The proof is sketched as follows: Let M be a legitimate DEVS (only legitimate DEVS's ultimately specify systems) specified by a next event model $(D, \{Des_\alpha\}, SELECT)$. Let $\alpha \in D$ be component $s \in S$ be a state of M. Activate α and then let M run. Since M is legitimate there must be a finite number of immediate activations until a state $s' \in S$ is reached for which $t_a(s') > 0$. Let $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_n$ be the sequence of activated components, then $s' = \delta_{\alpha_n} \cdot \delta_{\alpha_{n-1}} \cdot \ldots \cdot \delta_{\alpha_1}(s)$. Thus we can define $\delta'_\alpha = \delta_{\alpha_n} \cdot, \ldots, \cdot \delta_{\alpha_1}$ with influencees $I'_\alpha = \cup\{I_{\alpha_i} \mid i = 1, \ldots, n\}$ for a proper next event model M which specifies the same DEVS as M.

Note that δ_α' is computed by simulating M until all no immediate activations remain. To describe δ'_α explicitly may be extremely complex.

One measure of complexity is the size of the influencees I'_α. A formal way to bound this complexity is to consider infinite component models ($|D| = \infty$) with

finite influencee sets similar to cellular space automaton models. Then it can be shown that there are next event models of this kind which cannot be simulated by proper next event models – the influencee sets required by the latter cannot be bounded to fixed size.

17. CONCLUSIONS

Our main objective was to illustrate the integrated model description hierarchy approach in the case of the next event and activity scanning formalisms. Along the way, we uncovered constraints in SIMULA which may be amenable to improved language design.

Some comments are in order. First, as in Zeigler [4] page 164, it is not hard to extend the activity scanning formalism to process interaction form. Thus all three common model classes can be mapped into the discrete event formalism. Our results on representation power indicate that there is a significant advantage in descriptive convenience inherent in the process interaction and activity scanning formalisms vis a vis the next event and (certainly) proper next event formalisms.

Second, our technique of efficiently realizing the wait until statement (activity scanning) is reminiscent of the delay chains in GPSS and the demons of Hogeweg [7]. Our contribution in this regard is not in the novelty of this technique but in the generality of its formulation in terms of specificable model structure.

APPENDIX

Constraints Found to Exist in SIMULA

As is hopefully clear, the objective of this paper is to explain our hierarchical systems approach to model description. To illustrate the translation of the next event and activity scanning formalisms into the more general discrete event one, we sought a parallel simulation scheme in SIMULA. The latter seemed to be the ideal simulation language for this purpose because of its class construct facility. Palme [8] has for example shown how a specialized language such as GPSS can be obtained in SIMULA in a top down, hierarchical manner by successive prefixing of classes similar to our Figure 4. At the lowest level of the tree, the user sees GPSS procedures and data structures whose interpretation is provided by the higher level class definitions.

In trying to carry out an analogous extension of SIMULA we met with several difficulties. We shall briefly outline those since we believe that the constraints thus uncovered may be amenable to improved language design.

1. We wished to have each model component define its own local state structure and yet have these structures uniformly accessible at the higher next event level. This means that the class state defined

at the next event level (Figure 5) must be a dummy to be instantiated by each class local state defined by the components (Figure 7). SIMULA permits replacement of a class-owned procedure by a compatible subclass-owned procedure through use of VIRTUAL. I am unaware of the same possibility existing for data structures.

2. The influencee set of a component should be enumerable as part of the component declaration. This is not possible in SIMULA, since only classes are predefined and not individual member objects (which are created dynamically by new). Thus in Figure 7 only the attribute influencees is defined for each component but its content is not established until Figure 8 when the actual components are created and pointers to them placed in the influencee sets. In general, SIMULA does not allow the establishment of relations between individuals prior to their actual creation.

3. The most natural class structure for our simulation scheme is not hierarchical. In Figure 4, both active and passive components should be subordinate to component only active component should be subordinate to next event (since only it requires dynamic interpretation). Thus active component should be subordinate to both component and next event, a non-hierarchical (tree) situation. Hogeweg [7] has also noted the limitations of a strict hierarchical class structure.

4. In procedure wait until (Figure 1) we employ procedure as formal input parameter type. This is not allowed in SIMULA but its feasibility is attested to by its existence in ALGOL68.

References

1. House, P. and J. McLead, Large Scale Models for Policy Evaluation, Wiley, 1978.

2. Oven, T.I. and B.P. Zeigler, "Concepts for Advanced Simulation Methodologies" Simulation, March, 1979.

3. Nance, R., "Model Representation in Discrete Event Simulation: Prospects for Developing Documentation Standards", In: Current Issues in Simulation, N. Adam and A. Dogmmaci (Eds.), Academic Press, 1979.

4. Zeigler, B.P., Theory of Modelling and Simulation, Wiley, 1979.

5. Zeigler, B.P., System Theoretic Model Description: A Vehicle for Reconciling Diverse Modelling Concepts, In: Applied General Systems Research, G. J. Klir (Ed.), Plenum, 1978.

6. Franta, W.R., A Process View of Simulation, North Holland, 1977.

7. Hogeweg, P., "Heterarchical, Self-Organizing, Simulation Systems: Concepts and Applications in Biology", In: Methodology in Systems Modelling and Simulation, B. P. Zeigler (Ed.), North Holland, 1979.

8. Palme, J., Structured Simulation Programming, Swedish National Defence Research Institute, Stockholm, Sweden, 1978.