Proceedings of the 1986 Winter Simulation Conference
J. Wilson, J. Henriksen, S. Roberts (eds.)

HIERARCHICAL MODULAR MODELING/KNOWLEDGE REPRESENTATION*

Bernard P. Zeigler
Department of Electrical and Computer Engineering
The University of Arizona
Tucson, Arizona 85721 U.S.A.

This tutorial will emphasize concepts and methodology and will relate them to languages and software environments which are becoming available to support these concepts. We will show how high level specification of discrete event models with hierarchical and modular properties is crucial to the sound integration of knowledge representation approaches of artificial intelligence. We will discuss examples of hierarchical modular models exhibiting self-modifying structure capabilities and show how they may be implemented in conventional, as well as object-oriented symbolic languages. Finally, structuring of model bases for simulation environments will be presented and example tools illustrated.

Specifically, we shall discuss the following topics:

1. High level specification of discrete event models with hierarchical and modular properties using pseudo-code formalism.

2. Implementation of such specifications in procedural languages: How to express such pseudo-code in conventional languages such as SIMSCRIPT II.5 as well as in LISP-based languages.

3. Examples of such hierarchical modular models exhibiting self-modifying structure capabilities and embedded artificial intelligence.

4. Model/Knowledge Base Design: Organizing models using the system entity structure to constitute a reusable knowledge base.

Background for this tutorial may be found in (Zeigler 1984, 1985).

1. Modularity and Model Base Concepts

Figure 1 illustrates the fundamental concepts of modularity and model bases. Suppose that we have models A and B in the model base. If these model descriptions are in the proper modular form, then we can create a new model by specifying how the input and output ports of A and B are to be connected to each other and to external ports, an operation called coupling. The resulting model, AB, called a coupled model is once again in modular form. As can be seen, the term modularity, as used here, means the description of a model in such a way that it has recognized input and output ports through which all interaction with the external world is mediated. Once placed into the model base, AB can itself be employed to construct yet larger models in the same manner used with A and B. This property, called closure under coupling enables hierarchical construction of models.

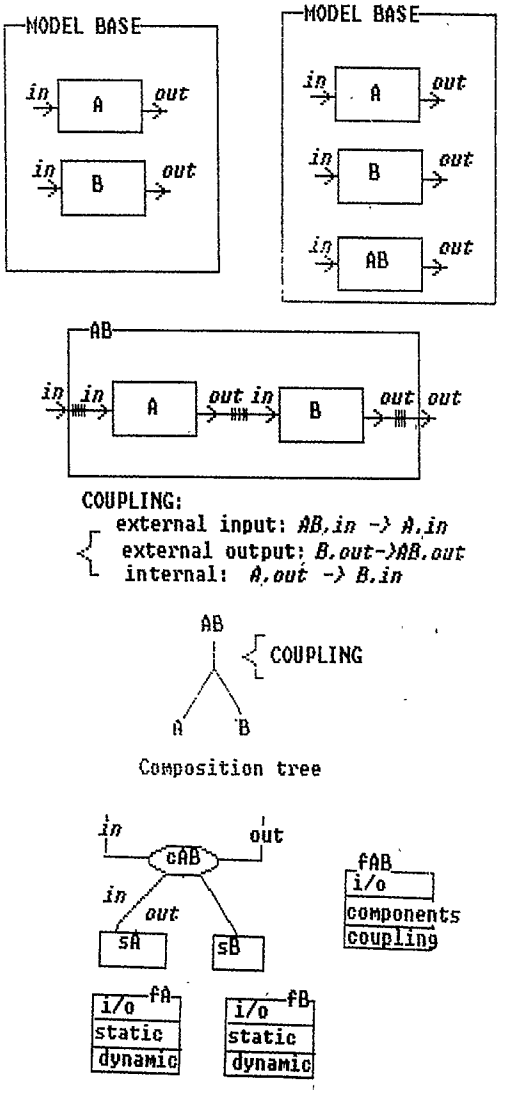


Figure 1

*Research reported here was supported by NSF Grant DCR 8514348, "Distributed Simulation of Hierarchical Multilevel Methods".

An important benefit of such modular construction is that a model in the model base can be readily, and independently, tested by coupling a test module to it. The ability to do such testing at each stage of a hierarchical construction facilitates reliable and efficient verification of large simulation models, not otherwise attainable. Test modules for models can be developed in a systematic manner using the concept of experimental frame, which specifies the input, control and output variables and constraints desired of the experimentation.

1.1 Coupling Specifications

Looking more closely at the coupling scheme in Figure 1, we see that it has three parts:

1) External input coupling tells how the input ports of the composite model are identified with the input ports of the components. For example, the notation AB. in - A. in, means that input port in of AB is connection to the input in of A. Note we employ the dot notation which prefixes that name of a component in front of a port so as to uniquely identify it (this obviates having to give different names to all the ports).

2) External output coupling tells how the output ports of the composite model are identified with the output ports of the components. Thus, the notation B. out - AB. out, means that the output port out of B is connected to the output of AB.

3) Internal coupling specifies how the components inside the coupled model are interconnected by telling how the output ports of components are connected to input ports of others. The notation, A. out - B. in, means that the output port out of B is connected to the input port in of B.

1.2 Hierarchical, Modular Composition

The composition tree in Figure 1 summarizes how components A and B are coupled together to form the coupled model AB. We think of the coupling specification as being associated with the line descending from AB before it splits into the components A and B. In other words, the coupling is associated with the decomposition of AB into components A and B. We will later consider that there may be more than one decomposition of a system, and that each one has a coupling specification associated with it.

Figure 2 illustrates the general pattern by which hierarchical models can be constructed. We see that a component can be either an atomic model or a coupled model. In the latter case, it is built from one or more components. (Later, we will return to describe the notation used in this presentation.). Since the term component appears twice, the diagram unfolds to an arbitrary depth. For example if we stop the recursion after one round, we get the diagram shown to the right of the arrow. We can

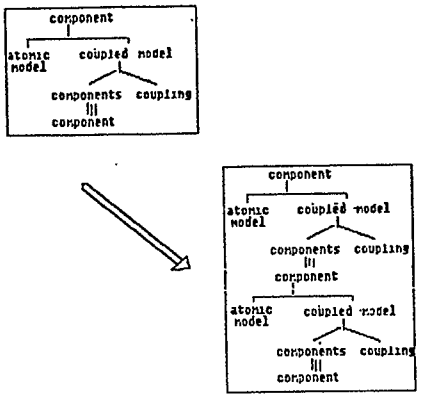
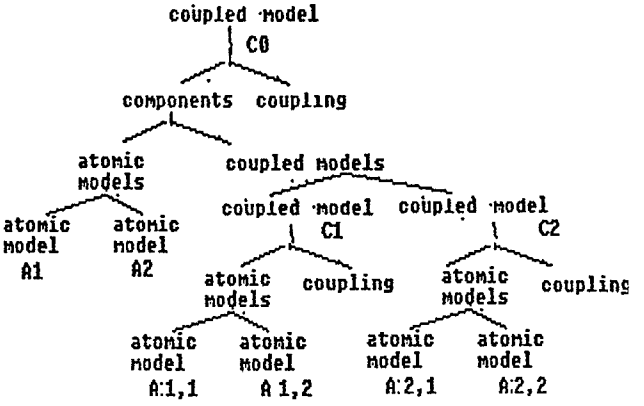


Figure 2

construct an actual hierarchical model from such a pattern such as shown in Figure 3. We start at the top (root of the tree), making the coupled model choice, we label C0. This choice necessitates specifying a coupling scheme and a set of components which can be atomic models or coupled models. We choose two atomic models, A1 and A2 and two coupled models C1 and C2. Each of the latter, in turn, requires a coupling scheme and a set of components. Selecting atomic models A1,1 and A1,2 for C1 and A1,1 and A2,2 for C2, completes the hierarchial construction.



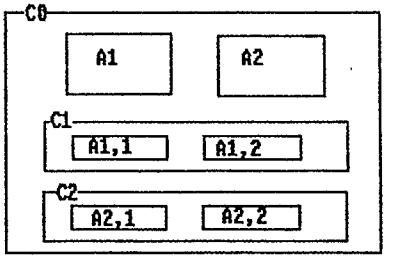
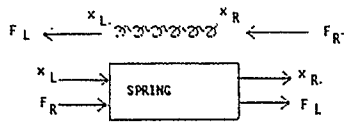


Figure 3

While the modularity property is a highly desirable one for the flexible construction of models, some work must be put into preparing model descriptions appropriately. Figure 4 demonstrates a modular specification of a simple spring, formulated in such a way that spring component models can be readily coupled together to represent the physical connection of the represented springs. To achieve modularity, positions of, and forces on, both ends of the spring, are made explicit. Figure 4b provides constant inputs of 0 magnitude to represent a single spring standing alone with its left end fixed and subject to zero external force. This is the usual oscillator. Figure 4c shows how springs may be connected. Note that the right end of spring 1 is coupled to the left end of spring 2 (spring1.xR -> spring2.xL). Also, the force generated internally in spring 2 on the left appears as the external force on the spring on the right (spring2.FL -> spring 1.FR). These coupling pairs constitute the internal coupling of the spring components. In like manner, one can concatenate successive spring components to represent, for example, a physical structure being assembled step by step by a robot in space. Figure 4d shows code that is generated by a LISP program to implement successive stages of this composition process.



INPUT VARIABLES:

$x_{L.}$: position of left end

$F_{R.}$: external force on right end

OUTPUT VARIABLES:

$x_{R.}$: position of right end

F_L : internally generated force on left end

STATE VARIABLES:

$x_{R.}$ $x_{R.dot}$

DYNAMIC STRUCTURE

$d/dt\ x_{R.} = x_{R.dot}$

$d/dt\ x_{R.dot} = (F_R - F_L)/m$

$F_L = k(x_{R.} - x_L - L)$

PARAMETERS:
m = mass, k = spring constant, L = length

Figure 4a

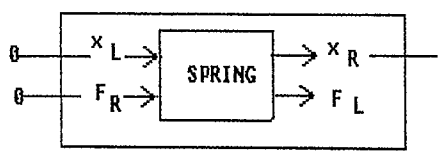


Figure 4b

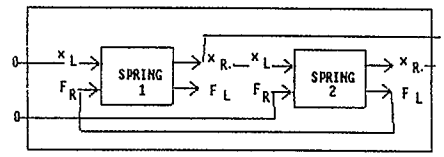


Figure 4c

```
MODEL EXTENSION 0 IS:

(0 1 D/DT XR (0) = XRDOT (0))
(0 2 D/DT XRDOT (0) = (FR (0) - FL (0)) /M)
(0 3 ==========================================MODEL======EQNS. FOR RIGHT END)
(1 1 FL (0) = K (XR (0) - XL (0) - L))
(1 2 ==========================================MODEL=======FORCE ON LEFT END)
(2 1 FR (0) = 0)
(2 2 ==========================================MODEL=======RIGHT END IS FREE)
(3 1 XL (0) = 0)
(3 2 ==============INITIAL CONDITION==================START AT 0)

MODEL EXTENSION 1 IS:

(0 1 D/DT XR (0) = XRDOT (0))
(0 2 D/DT XRDOT (0) = (FR (0) - FL (0)) /M)
(0 3 ==========================================MODEL======EQNS. FOR RIGHT END)
(1 1 FL (0) = K (XR (0) - XL (0) - L))
(1 2 ==========================================MODEL=======FORCE ON LEFT END)
(2 1 FR (0) = 0)
(2 2 ==========================================MODEL=======RIGHT END IS FREE)
(3 1 XL (0) = 0)
(3 2 ==============INITIAL CONDITION==================START AT 0)
(100 1 D/DT XR (1) = XRDOT (1))
(100 2 D/DT XRDOT (1) = (FR (1) - FL (1)) /M)
(100 3 ==========================================MODEL======EQNS. FOR RIGHT END)
(101 1 FL (1) = K (XR (1) - XL (1) - L))
(101 2 ==========================================MODEL=======FORCE ON LEFT END)
(102 1 FR (1) = 0)
(102 2 ==========================================MODEL=======RIGHT END IS FREE)
(2 1 FR (0) = FL (1))
(2 2 ==================COUPLING========NEW SPRING FORCES OLD RT.END)
(103 1 XL (1) = XR (0))
(103 2 ==================COUPLING========NEW LEFT CONNECTS TO OLD RT.)

MODEL EXTENSION 2 IS:

(0 1 D/DT XR (0) = XRDOT (0))
(0 2 D/DT XRDOT (0) = (FR (0) - FL (0)) /M)
(0 3 ==========================================MODEL======EQNS. FOR RIGHT END)
(1 1 FL (0) = K (XR (0) - XL (0) - L))
(1 2 ==========================================MODEL=======FORCE ON LEFT END)
(2 1 FR (0) = 0)
(2 2 ==========================================MODEL=======RIGHT END IS FREE)
(3 1 XL (0) = 0)
(3 2 ==============INITIAL CONDITION==================START AT 0)
(100 1 D/DT XR (1) = XRDOT (1))
(100 2 D/DT XRDOT (1) = (FR (1) - FL (1)) /M)
(100 3 ==========================================MODEL======EQNS. FOR RIGHT END)
(101 1 FL (1) = K (XR (1) - XL (1) - L))
(101 2 ==========================================MODEL=======FORCE ON LEFT END)
(102 1 FR (1) = 0)
(102 2 ==========================================MODEL=======RIGHT END IS FREE)
(2 1 FR (0) = FL (1))
(2 2 ==================COUPLING========NEW SPRING FORCES OLD RT.END)
(103 1 XL (1) = XR (0))
(103 2 ==================COUPLING========NEW LEFT CONNECTS TO OLD RT.)
(200 1 D/DT XR (2) = XRDOT (2))


(200 2 D/DT XRDOT (2) = (FR (2) - FL (2)) /M)
(200 3 ==========================================MODEL======EQNS. FOR RIGHT END)
(201 1 FL (2) = K (XR (2) - XL (2) - L))
(201 2 ==========================================MODEL=======FORCE ON LEFT END)
(202 1 FR (2) = 0)
(202 2 ==========================================MODEL=======RIGHT END IS FREE)
(102 1 FR (1) = FL (2))
(102 2 ==================COUPLING========NEW SPRING FORCES OLD RT.END)
(203 1 XL (2) = XR (1))
(203 2 ==================COUPLING========NEW LEFT CONNECTS TO OLD RT.)
```

Figure 4d

2.0 Hierarchical, Modular Discrete Event Model Development

To specify modular discrete event models requires that we adopt a different view than that fostered by traditional simulation languages. As with modular specification in general, we must view a model as possessing input and output ports through which all interaction with the environment is mediated. In the discrete event case, events determine values appearing on such ports. More specifically, when external events, arising outside the model, are received on its input ports, the model description must determine how it responds to

them. Also, internal events arising within the model, change its state, as well as manifesting themselves as events on the output ports to be transmitted to other model components.

A pseudo-code has been developed which makes such model specification straight forward. Each input port requires specification of an external transition, in the form of a when receive x on input port p... phrase. The internal state transition can be specified in the form of a process description which contains phrases of the form send y to output port p.

As an example, consider a simple buffering model. Initially, there will be two input ports: in, for receiving items to be queued and done, for receiving the acknowledgement of the down stream process. Later, we shall add an input port stop-send for flow control from the down stream process. The output port out, is for sending items down stream. The pseudo-code description appears as:

```
------external transition specification------

     when receive x on port in
          insert(x,queue)
          if one(queue) then goto SEND
             else passivate

     when receive done on port done
          if not empty(queue) then goto SEND
             else passivate

------internal transition specification------

     SEND: hold(preparation-time)
           send first(queue) to port out
           queue:=rest(queue)
           passivate
```

Note that the external transition specification has two when receive phrases, one for each input port. The first says that when a input value x is the only member of the queue, control should be sent to the phase SEND, otherwise the model should passivate (no next internal event will be scheduled). The internal transition specification has only one phase, SEND, in which the model stays for a period, preparation-time -- this causes the scheduling of an internal transition to occur at time = current time + preparation-time. Upon occurrence of the event, the model sends the first value in its queue to the output port out, removes it from the queue, and then passivates. The PASSIVE phase (in which the model passivates) represents a "ground" phase of the model in which it waits for external events while engaging in internal activity of its own.

The above description is not strictly correct since it immediately takes the model out of phase SEND when receiving an input rather that waiting for the resting time in the phase to fully elapse. The following modification of the external transition specification shows how to handle such interrupts:

```
     when receive x on port in
        insert (x,queue)
        if phase is not SEND
          and one (queue) then goto SEND
        else continue

     when receive done on port done
        if not empty (queue) then goto SEND
        else continue
```

The phrase "continue" replaces the "passivate" in the external transition specification. This indicates that time remaining in the phase in which the model finds itself is not to be changed as result of the external event processing. Of course, to express an interruption requiring a change in scheduling we would not use such a continue statement as the following specification for the input port stop-send shows:

```
   when receive x on port stop-send with
                                elapsed time e
      if phase = SEND and x = stop then
               processing-time-left :=   - e
               passivate

      if passive and x = start then
                 := processing-time-left
               goto SEND
      else continue
```

This external event causes the model to leave phase SEND, where it is holding for preparation-time, and abort the current transmission. Supposing that the time already spent in preparing the output need not be repeated when transmission is resumed, we store the remaining time in processing-time-left for recovery upon reentry to the SEND phase. Were there several jobs that could be in suspended states of this kind at once, we would save a processing-time-left with each one. The pseudo-code assumes that variables e (elapsed time in current phase) and (time left in current phase) are managed by the simulation medium (see Zeigler (1984)). Since the resting time in phase SEND is determined by , the associated hold statement is modified and must be determined upon initial entry to it as follows.

```
-----external transition specification-----

  when receive x on port in
       insert (x,queue)
       if one (queue) then
                  := preparation-time
               goto SEND
          else continue

  when receive done on port done
       if not empty (queue) then
              := preparation-time
            goto SEND
          else continue

-----internal transition specification-----

  SEND: hold(e)
        send first (queue) to port out
        queue:=rest(queue)
        passivate
```

2.1 Implementation in Simulation Environments

Figure 5 shows how such a pseudo-code description is coded in DEVS-Scheme, a simulation system developed to support hierarchical, modular discrete event modelling in an onjected oriented LISP language. Although we do not go into further details here to explain this implementation, the interested reader may obtain further information upon request to the author. A flow chart symbolism, equivalent to the pseudo-code specification, has been developed by the Siemens Co. (Ehr and Wnuk, 1985) and a translator into the Borris language (Hogrefe, 1985) is being written. A hierarchical modular environment for finite state models is described by Oren (1985).

```
-------------------------------------------
SPECIFICATION OF BUFFER MODEL IN DEVS-SCHEME

(define-structure state
                        queue
                        phase
 )
(make-state 'queue NIL 'phase 'PASSIVE),

(define (delta-ext s e x)  ; external transi-
   (case (content-port x)                tion
       ('in (insert ((state-queue s)
          (content-value x))
   (if (one state-queue)
     (set! (state-phase s) SEND) ; else
     (set! (state-phase s) PASSIVE))
   s
 )
   ('done (if (not (empty (state-queue s)))
            (set! (state-phase s) 'SEND) ;else
            (set! (state-phase s) 'PASSIVE))
       s
    )
  )

)

(define (delta-int s)       ; internal transi-
   (case (state-phase s)                  tion
       ('SEND (set! (state-phase s) 'PASSIVE)
              (set! (state-queue a)
              (rest (state-queue s)))
               s
       )
    )
)

(define (ta s)              ; time advance
   (case (state-phase s)
       ('SEND 0)
       ('PASSIVE NIL)
   )
)

(define (output s)          ; output
    (case (state-phase s)
       ('SEND (make-content 'port 'out 'value
              (first queue)))
       ('PASSIVE '()
    )
)
```

Figure 5

Returning to Figure 1, we see that each atomic model has three parts to its description: 1) the input/output specification giving the input and output ports and their ranges (values these variables can assume), 2) static structure giving the state and auxiliary variables and ranges; and 3) the dynamic structure, which provides the external and internal transition specification. Although other realizations are possible, we can have a file allocated for each such description with corresponding extensions. Thus we have files BUFFER.io, BUFFER.stat and BUFFER.dyn for the BUFFER model. A SIMSCRIPT 11.5 realization is shown in Figure 6. Note that the .io file in this case is for the model base user's information (although a processor could use it to generate appropriate coupling and range consistency checks). The .stat file is merged into the PREAMBLE file when this model is simulated, and likewise the .dyn file is appended at the end of the MAIN routine which will initiate simulation.

In the .dyn file, the external input specifications are represented by routines, one for each input port, each routine representing the body of the when receive phrase associated with that port. The external transition specification is realized by processes and/or events, as desired. The send to output port operation of the pseudo-code is realized by a call statement to a dummy routine associated with the output port. When the model is coupled to other components, this dummy routine is replaced by a routine representing an input port--the input port to which the output port has been coupled.

```
File BUFFER.io
input ports: in(x) , x: packet
           done   , done:flag
           stop-send(x)  x:(stop,start)
output ports: out(y), y: packet
==========================================
File BUFFER.stat
 process
every buffer has a phase and owns a queue
  temporary entities
every packet belongs to a queue
define queue as a fifo set
==========================================
File BUFFER.dyn

-----external transition specification------
 routine in(x)
 ...
 end

 routine done
 ...

 end

 routine stop-send(x,e)
 ...

 end

-----internal transition specification------
      process buffer
 SEND: wait σ
       call out (first(queue))
       queue:=rest(queue)
       suspend
==========================================
```

Figure 6

A coupled model has a different description: 1) the input/output specification, just as for atomic models, 2) the names of the components that are coupled together, and 3) the coupling specification, as discussed above, Thus, for example, model AB of Figure 1, has three files AB.io, AB.comp, and AB.coup.

To construct a coupled model, the files associated with its components are retrieved from the model base and merged with the files describing the coupled model. For example, to construct AB, we merge together the files for A, B, and AB, as shown in Figure 7. Note that the .coup file can be thought of as implementing a co-ordinator which channels the received inputs to the proper outputs. The co-ordinator routines are a straightforward translation of the pseudo-code coupling specification.

```
AB.io   (describes the input and output
         ports of AB)

A.stat  (static structure (variable decla-
B.stat   rations) placed in SIMSCRIPT
         preamble)

A.dyn   (dynamic structure (port routines
B.dyn    and processes) placed after MAIN
         routine)

AB.coup (coupling specification in form
         of "co-ordinator")

 routine AB.in(x)   (external input
 call A.in(x)         coupling)
 end

 routine B.out(x)   (external output
   call AB.out(x)     coupling)
 end

 routine A.out(x)   (internal coupling)
   call B.in(x)
 end.
```

Figure 7

2.2. Class Specification and Multiple Model Composition

Object-oriented programming supports the concepts of object classes and message passing between objects, which we wish to examine visa vis hierarchical, modular model specification. Message passing in the discrete event context arises naturally -- we need only interpret the sending of external events from output port to coupled input port as message transmission. Class concepts are not new to simulation having been introduced by the SIMULA language. Some work however, must be done to realize the modularity and coupling properties as we shall demonstrate. Finally, we note that hierarchical construction, made possible by successive couplings of larger and larger components, goes beyond the standard object oriented programming features.

Figure 8 depicts the concept of model class. A model class specification is actually a template for generating identical instances of the same model, either atomic or coupled. Note that there must be a naming scheme which assignes distinct names to instances as they are generated. The set of such instances itself constitutes a coupled model, called a multiple model. For example, the multiple model, AS denotes the set of instances of class A, currently existing in a simulation The composition tree relating AS to its possibly changing set of components is depicted using 3 parallel bars as shown. As in any coupled model, the components A1, A2, ... of AS may be coupled together.

Thus, there are three additional files needed to realize the class A: AS.io (the input and output ports of AS), AS.gen (which contains the code to generate the next instance of A), and AS.coup (which specifies the coupling of AS: the external input and output coupling that links the individual instances to the external input and output, respectively, as well as the internal coupling that links the instances to each other).

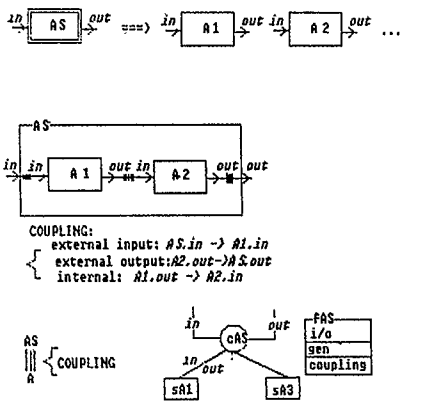


Figure 8

Figure 9 illustrates how model class instances (of different classes) may intercommunicate. The coupled models AS and BS may be coupled together to construct a coupled

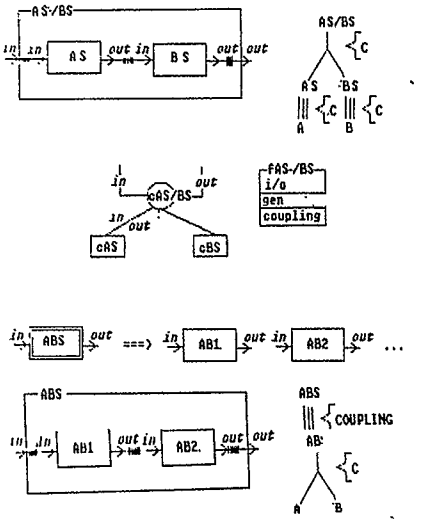


Figure 9

model AS/BS, with composition tree representation shown. Alternatively, instances of the coupled model AB may be generated to form the multiple ABS, composition tree shown.To appreciate the difference, the reader may think of A as a terminal and B as a computer. Then AS/BS represents a bank of terminals coupled to a bank of computers while ABS represents, a bank of terminal-computer pairs. The two representation are equivalent in that the same communication patterns are realizable in both, however, some are more easily expressed in one form or the other.

Figure 10 outlines how a model description can be converted to a class specification. Starting with an atomic model A, the A.dyn file is modified so that instances can be created and given identities. To keep track of the identities of the class instances, the external events are carried by messages that also carry the source and destination in any transmission from, or to, the co-ordinator for the multiple model. Referring to the example of

```
atomic model A          coupled model AS
A.io                    AS.io: A.in(x,m), A.out(x,m)
A.in(x),A.out(y)        AS.gen
A.stat                     activate A.p called idA
A.dyn                   A.stat
 ....                   A.dyn'
 routine A.in(x)         routine A.in(x,m)(m =
                                    (source: s;
                                    destination:
                                          idA)
 ....                    .....
 process A.p             process A.p(m)
 ....                    ....
 call A.out(y)           call A.out(y,m')(m'= (sour-
                                     ce: idA,
                                   destination:
                                        AS)
 ....                   ....
                        AS.coup
                         routine AS.in(x,m)
                         call A.in(x,m') (m' = sour-
                                     ce: AS,
                                    dest:idA1))
                         end
                         routine AS.out(y,m)
                          if source.m = idA1, call
                                       A.in(y,m')
                                 m' = (AS,m.dest)
                          if source.m = idA2, call
                                    (AS.out(y,m')
                                 m' = (AS,m.dest)
                         end
```

```
coupled models AS,BS      coupled model AS/BS
AS.io,BS.io               AS/BS.io
AS.gen,BS.gen             AS/BS.gen
AS.coup,BS.coup            call AS.gen to generate
                            A1,A2,...
                           call BS.gen to generate
                            B1,B2,...
                          AS/BS.coup
                            routine AS/BS.in(x,m)
                             call AS.in(x,m)
                            routine AS.out(y,m)
                             call BS.in(x,m)
                          end
                            routine BS.out(y,m)
                             call AS/BS.in(x,m)
                          end
```

Figure 10

Figure 10, each instance A1, A2,..,is given a unique identity variable idA1, idA2,.., respectively. When the co-ordinator for AS receives an input message, the external input coupling determines to which instance(s) it should so (in the example, to A1; examine routine AS.in in file AS.coup). The message sent by the co-ordinator to the instance bears the identity of the instance as destination. When an instance sends a message to one of its output ports, the message bears its identity as source, and the co-ordinator can decide where to channel the message from this information (in the example, if the source is A1 then it is channelled to A2, otherwise to the out port of AS).

3.0 The System Entity Structure

As we have seen, the composition tree of a hierarchical, modular model portrays its recursive structure of components and couplings. By generalizing the composition tree concept so as to represent not just a single model, but a family of possible models, we arrive at the concept of system entity structure. Full exposition of this concept is beyond the scope of this paper (refer to Zeigler, 1984; Rozenblit et.al., 1986). An example is given in Figure 11 for an adaptive computer architecture, full exposition of which is again beyond the scope of this paper. We see that the system consists of Flexible Processors (FPS), a multiple model consisting of individuals of the class Flexible Processor. As shown by the label "FP-Decomposition", each FP is a coupled model consisting of the components EXECUTIVE, SUPERVISORY, BUFFER, and F-COMPUTER.

In the entity structure we use the term entity rather than model, since the entity may have several possible models to represent it. We also refer to a decomposition as an aspect since there may be several possible decompositions for a given entity. The three vertical lines connecting a multiple entity and its singular instance is called a multiple aspect.

3.1 Variable Structure Modular, Hierarchical Models

The coupling between these components is illustrated in Figure 12, where we see that there are basically two types of extern- al event messages in circulation. Packet messages carry problem descriptions which are routed by the SUPERVISORY component to BUFFER queues and eventually, via other SUPERVISORY components to F-COMPUTER components which do the actual solution. Hire/fire messages carry commands to hire (take on additional FPs) or fire (return some FPs to an availability pool). These commands are handled by the EXECUTIVE components which decide whether to accept them and to which other EXECUTIVEs to pass them on. Since the number of FPs may change dramatically during a simulation run, this is an example of a variable structure model, i.e., a hierarchical, modular model, whose composition tree description may vary.

Returning to the entity structure of Figure 11, we see that the SUPERVISORY component is further decomposed "via the SUP-

Decomposition" into two components: SUPERVISOR and COORDINATOR (this model component should not be confused with the co-ordinator that handles coupled models in a simulation). The COORDINATOR can be one of three types, MS (multiserver), D&C (divide and conquer) or PL (pipeline) as denoted by the specialization "COORD-Specialization". Such a specialization indicates that a model class contains both a generic description, as well as specialized descriptions that can generate instances of several variants. To construct a SUPERVISORY component instance, we couple together a SUPERVISOR instance (for which no variation is required) and one of the MS, D&C, or PL COORDINATOR types. Another illustration of class specialization is shown by the "F-COMP-Specialization" which indicates that the F-COMPUTER class can have the indicated variants representing different styles of problem processing appropriate to the role that the F-COMPUTER is playing in the co-ordination configuration controlling it.

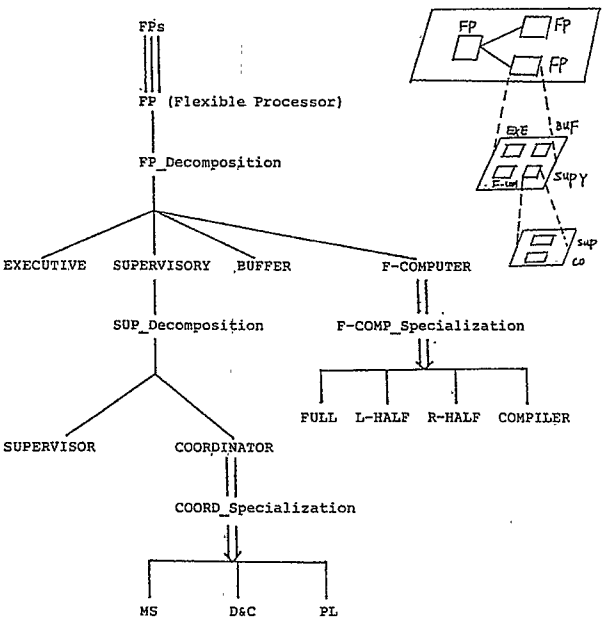


Figure 11

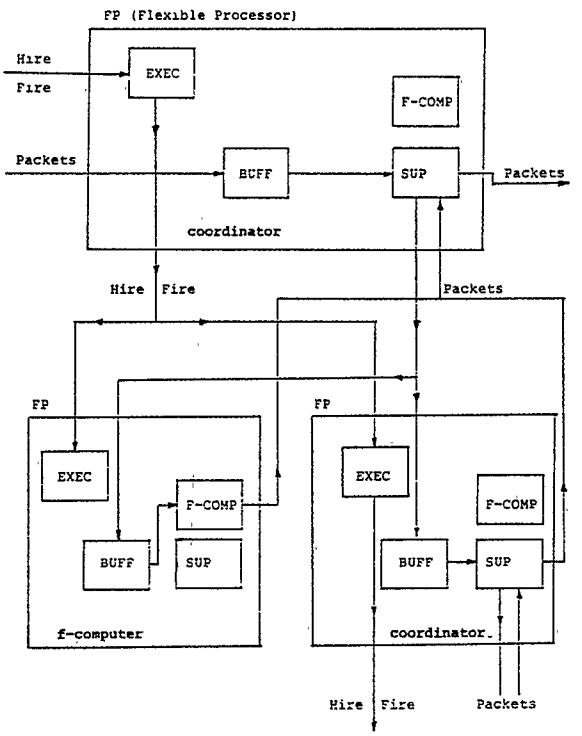


Figure 12

4.0 Entity Structure Pruning: Synthesis of Models

We can use the foregoing example to illustrate how the entity structure and model base combine to facilitate model construction. In this example, the model base contains files for the various model classes: FP, EXECUTIVE, SUPERVISORY, etc. Although the adaptive computer architecture model contains the intelligence to do its own reconfiguration, let us imagine constructing a particular configuration by a process called pruning of the entity structure. Starting at the root of the entity structure, we make a choice of how many FPS to use in the architecture (each multiple entity has a variable, e.g., FPS.NUMBER, to which we can assign a value). Having chosen the desired FPs, we must now specify how they are to be coupled together (the allowable possibilities are given by the entity struc- ture associated with the multiple aspect con- necting FPS and FP).

Each FP will be a coupled model composed of EXECUTIVE, SUPERVISORY, BUFFER and F-COMPUTER components. We must select one of the variants given by the "F-COMP-Specialization" for the F-COMPUTER of each FP. Moving to the SUPERVISORY component we see that it will be a coupled model with a SUPERVISOR and a CO-ORDINATOR, the type of which we must select from the "CO-OORD-Specialization". In this way, moving from top to bottom, we construct a number of FPs, each of which having a (different) F-COMPUTER and CO-ORDINATOR.

Having done top-down pruning of the entity structure to select the desired componets in the model base, we now follow bottom-up synthesis to construct the new model. Coupling a selected CO-ORDINATOR instance together with a SUPERVISOR (following the coupling specification associated with the "SUP-Decomposition)", we construct a SUPERVISORY component. The latter is then coupled with an EXECUTIVE instance, a BUFFER instance, and a selected F-COMPUTER instance to construct in FP (following the coupling specification associated with the "FP-Decomposition"). We repeat the same process to construct the chosen number of FP instances. Finally, we couple these FP instances together (following the selected coupling specification) to construct the final architecture.

The adaptive architecture model contains intelligence to carry a similar synthesis process. This intelligence takes the form of adaptive strategies that are sensitive to the

current workload conditions and issue hire or fire commands appropriately. In the models studies so far, the FPs are coupled in a tree structure, and hiring and firing takes place only near the leaf nodes of the tree. Thus, the structure does not radically differ from one synthesis call to the next, a feature one would expect in real self modifying systems.

4.1 The System Entity Structure/Model Knowledge Base

The entity structure/knowledge base combination provides a unifying conception of knowledge consistent with system theoretic insights. System theory distinguishes between system structure (the inner constitution of a system) and behavior (its outer manifestation). Regarding structure, we have seen that decomposition, coupling and taxonomies (class specialization definitions) should be fundamental relations in a knowledge representation scheme. Regarding system behavior, we shall distinguish between causal and empirical representations. By empirical representation we refer to actual records of data (time history of variable values) gathered from a real system or model. Causal relationships are integrated into units called models which can be interpreted by suitable simulators to generate data in empirical form.

The decomposition, taxonomic, and coupling relationships are combined in the system entity structure, a declarative scheme related to frame-theoretic and object-based representations. The model base contains models which are procedural in character, expressed in classical and AI-derived formalisms such as production rules and logic programming.

The entity structure methodology is supported by ESP-4 (entity structurer and pruner), a PASCAL system available for VAX under VMS and being ported to MS-DOS environments.

Conclusions

Fundamental concepts in hierarchical, modular model construction and knowledge base management have been briefly discussed. Applications of these concepts can be expected to grow as reusable bases of models are developed to support design, management and control of complex multifacetted systems. Especially, the design of systems incorporating artificial intelligence is a fertile ground for self-modifying, variable structure models. Another application, not mentioned here yet is to distributed simulation, the use of multiprocessors to achieve significant speedup in simulation of highly complex models. Hierarchical, modular model specification provides a natural way to exploit parallelism inherent in multicomponent models.

Bibliography

Concepcion, A.I. and B.P. Zeigler (in press), "DEVS Formalism: A Framework for Hierarchical Model Development", IEEE Transactions on Software Engineering.

Ehr, W. and A. Wnuk (1985), "Discrete Event Simulation of a Model Family with Boris", Proc. 11 IMACS World Congress, Oslo, Norway.

Hogrefe, D. (1985), "Tool Support for Model Description with SDL and Simulation", in: Cybernetics and Systems, (ed: R. Trappl), D. Reidel Pub. Co.

Oren, T.I. (1984), "Design of SEMA: A Software System for Computer-Aided Modelling and Simulation of Sequential Machines", Simulation J.

Rozenblit, J.W., S. Sevinc and B. P. Zeigler (1986), "Knowledge-based Design of LANs Using System Entity Structure Concepts", Proc. Winter Simulation Conf.

Zeigler, B.P. (1984), Multifaceted Modelling and Discrete Event Simulation, Academic Press, London and Orlando, FL.

Zeigler, B.P. (1985), Theory of Modelling and Simulation, Krieger Pub. Co., Malabar, FL. (Reprint Edition of original published by Wiley, 1976).