Abstract Simulator for the Parallel DEVS Formalism

Alex ChungHen Chow

Object Technology Products

IBM Corp.
Austin, TX 78758
alexc@austin.ibm.com

Bernard P. Zeigler
Doo Hwan Kim

Department of
Electrical and Computer Engineering
The University of Arizona
Tucson, AZ 85721
zeigler@ece.arizona.edu
dhkim@ece.arizona.edu

Abstract

A recent paper introduced the Parallel DEVS formalism which exploits the parallelism of transition collisions in the simulation of DEVS models. Here we present a design for the abstract simulator needed to prove the formalism's soundness and to serve as a reference for implementation. The abstract simulator is composed of cooperating simulation engines, (simulators and co-ordinators) that use bag-like messages to synchronize the parallel activities that are distributed across autonomous asynchronous processors. The approach suggests engines that are efficient in both sequential and distributed/parallel environments. After describing the abstract simulator we briefly discuss a prototype implementation that affords a high degree of flexibility by mechanizing the "closure under coupling" property of the Parallel DEVS formalism and the characteristics of object-oriented systems.

Keywords:
Discrete Event Simulation, DEVS formalism, Object-Oriented modeling and simulation, Distributed/parallel simulation.

1 Introduction

The advantages of hierarchical modeling capability such as reduction in model development time, support for reuse of a database of models, and aid in model verification and validation are becoming well accepted[10]. Environments supporting hierarchical modeling are transitioning from research [16][7][9][4] into practice[6][3].

The necessary compute power for executing complex hierarchical models lies in distributed and parallel simulation[2][8][5]. Thus it is timely to reexamine the basic formalisms of discrete event modeling in the light of future high performance simulation requirements.

The *Discrete Event System Specification*(*DEVS*) formalism was introduced in the early 70's and later extended to enable constructing discrete event simulation models in a hierarchical, modular manner[14][15]. *DEVS* introduces a strong modularity between model specification and simulation. Not only does it provide a powerful modeling methodology but also a framework for model behavior generation via its abstract simulator concepts[16]. Since it is language and platform independent, *DEVS* affords an excellent vehicle for investigating alternative parallel/distributed mappings and architectures[17][13] [12].

Parallel DEVS(*P-DEVS*)[1] is a revision of the hierarchical, modular *DEVS* modeling formalism. The revision distinguishes between transition collisions and ordinary external events in the external transition function of *DEVS* models. Such separation extends the modeling capability of the collisions. The revision also does away with the necessity for tie-breaking simultaneously scheduled events, as embodied in the *select* function (a heritage of the sequential simulation paradigm in which *DEVS* originated). The latter is replaced by a well-defined and consistent formal construct that allows all transitions to be simultaneously activated. The revision provides a modeler with both conceptual and parallel-execution benefits.

An earlier article[1] presented the *P-DEVS* formalism and showed it to be closed under coupling, thus preserving hierarchical, modular, construction properties. This construct leads to the definition of its abstract simulator which correctly implements the for-

malism and exploits the increased parallelism. Here, we briefly review the *P-DEVS* formalism and proceed to discuss the abstract simulator concepts that form the basis of its concrete implementation.

2 The Parallel DEVS

The *P-DEVS* model is a structure:

$$M =< X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta, >$$

X: a set of input events.
S: a set of sequential states.
Y: a set of output events.
$\delta_{int} : S \rightarrow S$: internal transition function.
$\delta_{ext} : Q \times X^b \rightarrow S$: external transition function,
  X^b is a set of bags over elements in X,
  $\delta_{ext}(s, e, \phi) = (s, e)$.
$\delta_{con} : S \times X^b \rightarrow S$: confluent transition function.
$\lambda : S \rightarrow Y^b$: output function.
$ta : S \rightarrow R_{0+-\infty}$: time advance function,
  where $Q = \{(s, e) | s \in S, 0 < e < ta(s)\}$,
    e is the elapsed time since last state transition.

The *P-DEVS* formalism enables a modeler to explicitly define the collision behavior by using the so-called confluent transition function, δ_{con}. δ_{con} gives the modeler complete control over the collision behavior when a component receives events at the time of its internal transition, $e = 0$ or $e = ta(s)$. Rather than serializing model behavior at collision times, the *P-DEVS* formalism leaves this decision of what serialization to use, if any, to the modeler. Indeed, if so desired, the *E-DEVS*[11] formalism can be recovered by setting $\delta_{con}(s, x^b)$ to $\delta_{ext}(s_n, 0, x_n)$, where $n \geq 1$, $s_1 = \delta_{int}(s)$, $s_n = \delta_{ext}(s_{n-1}, 0, x_{n-1})$ when $n > 1$, and x_n is a desired serialization defined by $Order(x^b)$.

The semantics of the *Parallel DEVS* are as follows: the internal transitions are carried out at the next event time for all imminent components receiving no external events. Also, external events generated by these imminents trigger external transitions at receptive non-imminents (those components for which there are no internal transitions scheduled at the event receiving time). However, for those components for which the internal and external transitions collide, the *confluent transition function* is employed instead of either the internal or external transition function to determine the new state.

The structure of the revised *coupled model* is —

$$DN =< X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$$

X: a set of input events.
Y: a set of output events.
D: a set of components.
for each i in D,
  M_i is a component.
for each i in $D \cup \{self\}$, I_i is the influencees of i.
for each j in I_i,
  $Z_{i,j}$ is a function,
  the i-to-j output translation.

The structure is subject to the constraints that for each i in D,
$M_i =< X_i, S_i, Y_i, \delta_{inti}, \delta_{exti}, \delta_{coni}, ta_i >$ is a *P-DEVS* structure,
I_i is a subset of $D \cup \{self\}$, i is not in I_i,
$Z_{self,j} : X_{self} \rightarrow X_j$,
$Z_{i,self} : Y_i \rightarrow Y_{self}$,
$Z_{i,j} : Y_i \rightarrow X_j$.

Here $self$ refers to the coupled model itself and is a device for allowing specification of external input and external output couplings.

Closure of the *P-DEVS* formalism under coupling was done by constructing the resultant of a coupled model and showing it to be a well defined *P-DEVS*. The *resultant* of a coupled model ($DN =< X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$) is a *P-DEVS* model ($M =< X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$), where
$S = \times Q_i$ where $i \in D$.
$ta(s) = \text{minimum}\{\sigma_i | i \in D\}$,
  where $s \in S$ and $\sigma_i = ta(s_i) - e_i$.
Let
  $s = (..., (s_i, e_i), ...)$,
  $IMM(s) = \{i | \sigma_i = ta(s)\}$,
  $INF(s) = \{j | j \in \cup_{i \in IMM(s)} I_i\}$,
  $CONF(s) = IMM(s) \cap INF(s)$,
  $INT(s) = IMM(s) - INF(s)$,
  $EXT(s) = INF(s) - IMM(s)$.

We partition the components into four sets at any transition time. $INT(s)$ contains the components ready to make an internal transition without input events. $EXT(s)$ contains the components receiving input events but not scheduled for an internal transition. $CONF(s)$ contains the components receiving input events and also scheduled for internal transitions at the same time. $UN(s)$ contains the remaining components. Then,
$\lambda(s) = \{Z_{i,self}(\lambda_i(s_i)) | i \in IMM(s) \wedge self \in I_i\}$.
$\delta_{int}(s) = (..., (s'_i, e'_i), ...)$,
where
  $(s'_i, e'_i) = (\delta_{inti}(s_i), 0)$ for $i \in INT(s)$,

$(s'_i, e'_i) = (\delta_{exti}(s_i, e_i + ta(s), x^b_i), 0)$ for $i \in EXT(s)$,
$(s'_i, e'_i) = (\delta_{coni}(s_i, x^b_i), 0)$ for $i \in CONF(s)$,
$(s'_i, e'_i) = (s_i, e_i + ta(s))$ otherwise $i \in UN(s)$,
and
$x^b_i = \{Z_{o,i}(\lambda_o(s_o)) | o \in IMM(s) \wedge i \in I_o\}$.

The resultant internal transition comprises of four kinds of component transitions: internal transitions of $INT(s)$ components, external transitions of $EXT(s)$ components, confluent transitions of $CONF(s)$ components and the remainder,$UN(s)$, whose elapsed times are merely updated by $ta(s)$.

The δ_{ext} of the resultant is defined by:
$\delta_{ext}(s, e, x^b) = (..., (s'_i, e'_i), ...)$,
where
$(s'_i, e'_i) = (\delta_{exti}(s_i, e_i + e, x^b_i), 0)$ for $i \in I_{self}$,
$(s'_i, e'_i) = (s_i, e_i + e)$ otherwise,
and
$x^b_i = \{Z_{self,i}(x) | x \in x^b \wedge i \in I_{self}\}$.

The incoming event bag, x^b is translated and routed to the event bag, x^b_j, of each influenced child, j. The resultant's external transition comprises all the external transitions of the influenced children.

Finally, the δ_{con} of the resultant is defined by:
Let
$INF'(s) = \{j | j \in \cup_{i \in (IMM(s) \cup \{self\})} I_i\}$,
$CONF'(s) = IMM(s) \cap INF'(s)$,
$INT'(s) = IMM(s) - INF'(s)$,
$EXT'(s) = INF'(s) - IMM(s)$.
$\delta_{con}(s, x^b) = (..., (s'_i, e'_i), ...)$,
where
$(s'_i, e'_i) = (\delta_{inti}(s_i), 0)$ for $i \in INT'(s)$,
$(s'_i, e'_i) = (\delta_{exti}(s_i, e_i + ta(s), x^b_i), 0)$
for $i \in EXT'(s)$,
$(s'_i, e'_i) = (\delta_{coni}(s_i, x^b_i), 0)$ for $i \in CONF'(s)$,
$(s'_i, e'_i) = (s_i, e_i + ta(s))$ otherwise,
and
$x^b_i = \{Z_{o,i}(\lambda_o(s_o)) | o \in IMM(s) \wedge i \in I_o\} \uplus$
$\{Z_{self,i}(x) | x \in x^b \wedge i \in I_{self}\}$.

The critical difference in the *P-DEVS* compared with the original DEVS is that to establish closure under coupling, we must also define the δ_{con} of the resultant. Fortunately, it turns out that the difference between δ_{con} of the resultant and its δ_{int} is simply the extra confluent effect produced by the incoming event bag, x^b, at simulation time $ta(s)$. By redefining the influencee set to $INF'(s)$ that includes the additional influencees from the incoming couplings, $z(self, i)$, we come up with three similar groups for δ_{con}. The hierarchical consistency is achieved here by the $\uplus$ operation that gathers all external events, whether internally or externally generated, at the same time into one single event group.

¿From the definition of the δ_{int}, δ_{con}, and δ_{ext}, we see that they are special cases of a more generic transition function $\delta(s, e, x^b)$[15]. δ_{int} is applied to the cases when $(s, e, x^b) = (s, ta(s), \phi)$, δ_{con} to the cases when $(s, e, x^b) = (s, ta(s), x^b)$ where $x^b \neq \phi$, and, δ_{ext} to (s, e, x^b) where $0 \leq e < ta(s)$ and $x^b \neq \phi$.

3 The Abstract Simulator

We now describe the abstract simulator needed to demonstrate soundness of the *P-DEVS* formalism. As in the original definition, we specialize the processors into two different simulation engines, *simulator* and *co-ordinator* [15].

Both δ_{con} and δ_{ext} depends on the events in the bag, x^b. An event in the bag is a result from an output function and all the translations on the event path. An output function depends on a state prior to a transition at the same instance. It is clear that the output function must be invoked before any transition function. We use $(@, t)$ and $(done, t)$ messages to synchronize this activity while (y, t) and (q, t) messages trasport the output content. We also assume that if two messages are sent from the same source, the ordering between them is preserved at the receiving end.

The *simulator* attached to an atomic model is given first:

when a $(@, t)$ message is received
if $t = t_N$ **then**
  $y := \lambda(s)$
  send (y, t) to the parent coordinator
  send $(done, t)$ to the parent coordinator
end if
else raise error
end when

when a (q, t) message is received
lock the *bag*
Add event q to the *bag*
unlock the *bag*
send $(done, t)$ to the parent coordinator
end when

when a $(*, t)$ message is received
case $t_L \leq t < t_N$ and *bag* is not empty
  $e := t - t_L$
  $s := \delta_{ext}(s, e, bag)$

empty bag
$t_L := t$
$t_N := t_L + ta(s)$
end case
case $t = t_N$ and bag is empty
$s := \delta_{int}(s)$
$t_L := t$
$t_N := t_L + ta(s)$
end case
case $t = t_N$ and bag is not empty
$s := \delta_{con}(s, bag)$
empty bag
$t_L := t$
$t_N := t_L + ta(s)$
end case
case $t > t_N$ or $t < t_L$
raise error
end case
send $(done, t_N)$ to parent coordinator
end when

The *simulator* uses one single message, $(*, t)$, to synchronize three different transitions of the atomic model. Obviously, other implementations with more synchronization messages for different transitions can remove the need of case statements for possibly faster simulation. The implementation introduced here serves as an example to indicate the correct semantic application of each transition function which enables us to use the generic transition function described above for a *co-ordinator*. The implementation of a *co-ordinator* is given.

when a $(@, t)$ message is received from parent coordinator
if $t = t_N$ **then**
$t_L := t$
for all imminent child processors i with minimum t_N
send $(@, t)$ to child i
cache i in the *synchronize* set
end for
wait until $(done, t)$'s are received from all imminent processors
send $(done, t)$ to the parent coordinator
else raise an error
end when

when a (y, t) message is received from child i
for all influencees, j of child i
$q := z_{i,j}(y)$
send (q, t) to child j
cache j in the *synchronize* set
end for
wait until all $(done, t)$'s are received from j's
if $self \in I_i$ (y is to be transmitted upward) **then**
$y := z_{i,self}(y)$
send (y, t) to the parent coordinator
end if
end when

when a (q, t) message is received from parent coordinator
lock the bag
Add event q to the bag
unlock the bag
end when

(y, t) messages are always processed within the *wait* statement when receiving a $(@, t)$ message. This synchronization ensures that the outputs of any model, either atomic or coupled, are routed to their immediate influencees' bags. All children ready for a transition are cached in a set called *synchronize* set to eliminate the activities of $UN(s)$ components. The elapsed time can always be calculated from the t_L associated with each component and the absolute global clock, t.

From the construction described in the previous section, we see that
$x_i^b = \{z_{o,i}(\lambda_o(s_o)) | o \in IMM(s) \wedge z_{o,i} \in Z\} \uplus \{z_{self,i}(x) | x \in x^b\}$

After the processing of $(@, t)$ is over, output events are distributedly stored in input bags of influencees throughout the hierarchy. Though the first part of x_i^b is ready now, the $\uplus$ with the second part must be done by sending (q, t) messages to influencees of $self$ at the beginning of the each $(*, t)$ phase. This operation assures the uniformity of the hierarchy. All events are routed down to the atomic influencees by successive $(*, t)$ phases of nodes from root to atomic components. A transition is completed when finally one of the transition functions is invoked at the atomic level.

when a $(*, t)$ message is received from parent coordinator
if $t_L \leq t \leq t_N$ then
for all receivers, $j \in I_{self}$ and all $q \in bag$
$q := z_{self,j}(q)$
send (q, t) to j
cache j in the *synchronize* set
end for
empty bag

wait until all $(done, t)$'s are received
for all i in the *synchronize* set
send $(*, t)$ to i
end for
wait until all $(done, t_N)$'s are received
$t_L := t$
$t_N :=$ minimum of components' t_N's
clear the *synchronize* set
send $(done, t)$ to parent coordinator
else raise an error
end when

Elements in the *synchronize* set are imminent components, influencees or both. Because of the consistent application, we delay the distinction of a transition only until the notification arrives at the atomic level.

The implementation of this coordinator routes down the output events during the $(*, t)$ phases. It simply reflects the construction of the transition functions of a coupled model. Another implementation might choose to route the events during the $(@, t)$ phase directly to the final atomic influencees. The bag implementation of the coupled model can thus be omitted. Both implementations are equivalent and render the same simulation result.

The topmost coordinator is driven by a special coordinator called the root coordinator which constantly advances the global simulation time to the next simulation time of a simulation, sends $(@, t)$ and $(*, t)$ messages to the topmost coordinator, asks the next simulation time, and repeats until the next simulation time is infinite.

Root coordinator
$t := t_N$ of the topmost coordinator
while $t \neq \infty$
send $(@, t)$ to the topmost coordinator
wait until $(done, t)$ is received from it
send $(*, t)$ to the topmost coordinator
wait until $(done, t_N)$ is received from it
end while
raise simulation completed

The simulation procedure exposes the parallelism among transitions of elements in *synchronize* set and abstract simulator design handles *transitory* states in a well defined manner.

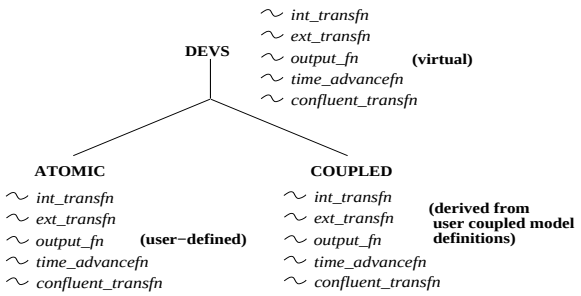

Figure 1: Abstract Class and Inheritance Hierarchy Exploiting Closure Under Coupling

4 Flexibility of Hierarchical Model Mappings

The standard mapping of a hierarchical *P-DEVS* model onto an abstract simulator results in a hierarchical architecture with a one-one correspondence to the model's composition tree structure. However, many alternative mappings exist and some are likely to be much better, depending on the model behavior and the host platform characteristics. Some possibilities have been investigated[17][13]. Here, we note that closure under coupling enables any coupled model in the model composition tree to be mapped into an equivalent resultant model. This mapping, described above, can be implemented within an object-oriented framework as illustrated in Figure 1.

Here, *atomic* model and *coupled* model objects present the same interface to clients, one that is abstracted in a *devs* superclass with virtual transition functions. This greatly increases the flexibility with which mappings can be done. In partcular it helps overcome limitions of conventional high performance architectures which do not support hierarchical clusters. Only one coordinator process is needed at the top level for managing the intercommunication and synchronization of nodes. Simulator processes run on other nodes and are linked to either atomic or coupled models, the encapsulation mapping embodied in the common interface blinds them to the difference.

We are currently investigating the application of this concept to large scale ecosystem simulation. As illustrated in Figure 2, a landscape, such as a watershed, is represented by a "base" model with a large number,e.g., one million, of cells. This number is orders of magnitude larger than the number of nodes in the highest performance massively parallel computers such the 1000 node CM-5. Therefore the base model cannot be mapped in a one-one manner and some par-

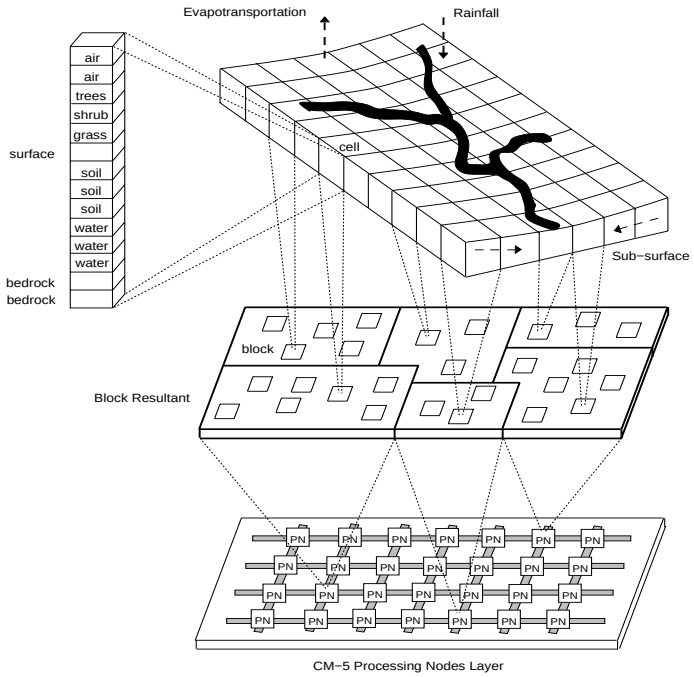

Figure 2: Mapping of landscape base model onto a flat massively parallel architecture (e.g. CM-5)

titioning is required as illustrated in Figure 2. To retain the base model dynamcs, each block becomes a coupled model over the component cells within its scope. Using the closure under coupling concept, each coupled model is represented by its resultant *P-DEVS* model and these resultants are coupled together in a manner preserving the coupling behavior of orginal base model. The outputs of blocks are collections of outputs of the enclosed cells. Management of such collections is nicely handled by the bag construct in the *P-DEVS* formalism. The block resultants are assigned to simulators and the coupling of these resultants to a coordinator. The transformation of the base model into an equivalent coupling of blocks is called "deepening". The entire process can be defined formally and implemented nicely in the object-oriented paradigm. Partitioning of cells into blocks is not constrained and indeed, can be performed dynamically during execution to balance the processor loads as the locus of cellular activity migrates about.

5 Conclusions

In the *Parallel DEVS* formalism, a modeler is explicitly enabled to supply the confluent transition function that captures the collision behavior. This function allows the coupling construction to follow the semantics of a collision down to the atomic level and obviates any behavioral difference between a model and its deepened and flattened restructurings.

The abstract simulator concept leads to many possible implementations. The well isolated transition groups add to the existing possibilities to exploit the parallelism of the hierarchical *DEVS* models. Since the abstract simulation engine is based on the assumption of a parallel environment, the implementation on parallel machines is straightforward. Moreover, closure under coupling supports flexible restructuring for more effective mappings, both static and dynamic, to particular platforms.

Acknowledgements

Some of this research is supported by NSF HPCC Grand Challenge Application Group Grant ASC-9318169 with ARPA participation and employs the CM-5 at NCSA under grant MCA94P02.

References

[1] Alex C. Chow and Bernard P. Zeigler. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In *Winter Simulation Conference Proceedings*, Orlando, Florida, 1994. SCS.

[2] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[3] J. Hu. Cedes: Object-oriented hardware modeling & simulation.

[4] Tag Gon Kim. *DEVSIM++ User's Manual*. Taejon, Korea, 1994.

[5] B. Lubachevesky, A. Weiss, and A. Schwartz. An analysis of rollback-based simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(2), 1991.

[6] C. D. Pegden and D. A. Davis. Arena^{TM} : A SIMAN/CINEMA based hierarchical modeling system. In *Winter Simulation Conference Proceedings*, Phoenix, AZ, 1992.

[7] H. Praehofer. An environment for DEVS-based multiformalism simulation in common Lisp/CLOS. *Discrete Event Dynamic Systems*, 3, 1993.

[8] B. Preiss. The Yaddes distributed discrete event simulation specification language and execution environment. In *Distributed Simulation 89*. SCS Press, 1989.

[9] J. W. Rozenblit and P. Janknski. An integrated framework for knowledge-based modeling and simulation of natural systems. *Simulation*, 57(3), 1990.

[10] Robert Sargent. Hierarchical modeling for discrete event simulation (panel). In *Winter Simulation Conference Proceedings*, page 569, Los Angeles, CA, 1993.

[11] Yung-Hsin Wang and Bernard P. Zeigler. Extending the DEVS Formalism for Massively Parallel Simulation. *Discrete Event Dynamic Systems: Theory and Applications*, 3:193–218, 1993.

[12] T.G. Kim Y.R. Seong and K.H. Park. Mapping modular, hierarchical discrete event models in a hypercube multicomputer. In *Proceedings of International Conference on Massively Parallel Proc. Application and Development*, 1994.

[13] T.G. Kim Y.R. Seong, S.H. Jung and K.H. Park. Parallel simulation of hierarchical modular devs models: A modified time warp approach. *International Journal of Computer Simulation*, (to appear).

[14] Bernard P. Zeigler. *Theory of Modelling and Simulation*. Wiley-Interscience, New York, 1976.

[15] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, London, 1984.

[16] Bernard P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models*. Academic Press, San Diego, California, 1990.

[17] B.P. Zeigler and G. Zhang. Mapping hierarchical discrete event models to multiprocessor systems: Concepts, algorithm and simulation. *Parallel & Distributed Computing*, 10:271–281, 7 1990.