DEVS-C++: A High Performance Modelling and Simulation Environment

Bernard P. Zeigler

AI and Simulation Group
Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721
Email :: zeigler@ece.arizona.edu

Yoonkeon Moon

AI and Simulation Group
Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721
Email :: moon@ece.arizona.edu

Doohwan Kim

AI and Simulation Group
Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721
Email :: dhkim@ece.arizona.edu

Jeong Geun Kim

Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721
Email :: jkkim@ece.arizona.edu

Abstract

Simulation of landscape ecosystems with high realism demands computing power greatly exceeding that of current workstation technology. However, the prospects are excellent that modelling and simulation environments may be implemented on next-generation high performance, heterogeneous distributed computing platforms. Computing technology is becoming powerful enough to support the voluminous amounts of knowledge/information necessary for representing such systems and the speed required of simulations to provide reliable answers in reasonable time. This paper provides an overview of a project to develop a high performance modelling and simulation environment to support modelling of large-scale, high resolution landscape systems.

1 High performance simulation

The paper reports on design and benchmarking of a high performance computing environment supporting simulation of landscape ecosystems at high levels of resolution and encompassing large areas, such as forests and watersheds. We report on experience gained in an NSF-ARPA sponsored Grand Challenge Application Group project whose goals are: 1) constructing a modelling and simulation environment that employs massively parallel processing and Discrete Event System Specification(DEVS) formalized models to simulate interactions of ecosystem processes at selectable scales of space and time, 2) integrating, as intrinsic to the environment, Geographical Information System(GIS) data bases to provided realistic descriptions of 3-dimensional landscapes, and 3) supporting experimentation and interpretation through scientific visualization and automated optimization.

Simulation of landscape ecosystems with high realism demands computing power greatly exceeding that of current workstation technology. However, the prospects are excellent that modelling and simulation environments may be implemented on next-generation high performance, heterogeneous distributed computing platforms. These powerful platforms are necessary to address challenging computing problems using high-resolution large scale representations of systems composed of natural and artificial elements. High performance simulation-based design environments are characterized by two levels of intensive knowledge/information processing; at the decision-making level, searches are conducted through vast problem spaces of alternative design configurations and associated model structures; at the execution level, simulations generate and evaluate complex candidate model behaviors, possibly interacting with human actors in real time.

Computing technology is becoming powerful enough to support the voluminous

amounts of knowledge/information necessary for representing such systems and the speed required of simulations to provide reliable answers in reasonable time.

2 Layered architecture

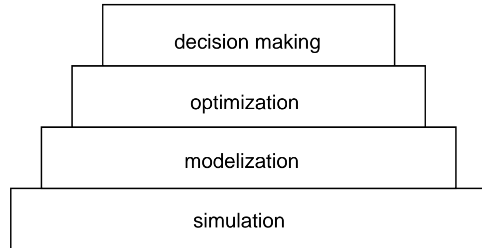

Figure 1: Layered Representation of Simulation-based Decision Making

Figure 1 depicts simulation-based decision making in terms of a layered system of functions. In this paradigm, decision makers, for example, forest managers, base their decisions on experiments with alternative strategies (e.g., for reducing the risk of wild fires) where the best strategies (according to some criteria) are put into practice. For a variety of reasons, experiments on models are preferred to those carried out in reality. For realistic models (e.g., of forest fire spread), such experiments can not be worked out analytically and require direct simulation. The design of our environment to support all these activities is based on the layered collection of services shown in Figure 1, where each layer uses the services of lower layers to implement its functionality as just explained.

In the next section, we discuss why we are basing the simulation environment on a system- theoretically based formalism, DEVS.

2.1 The DEVS formalism

DEVS falls within the formalisms identified by Ho[1] for Discrete Event Dynamical Systems (DEDS). Work on a mathematical foundation of discrete event dynamic modelling and simulation began in the 70s[2, 3, 4] when DEVS was introduced as

an abstract formalism for discrete event modelling. Because of its system theoretic basis, DEVS is a universal formalism for DEDS[5]. Indeed, DEVS is properly viewed as a short-hand to specify systems whose input, state and output trajectories are piecewise constant[5]. The step-like transitions in the trajectories are identified as discrete events.

Discrete event modelling has its origin in job shop scheduling simulations in operations research. More recently control theorists developed their own discrete event formalisms[1]. The universality claims of the DEVS are justified by characterizing the class of dynamical systems which can be represented by DEVS models. Praehofer and Zeigler[6] showed that any causal dynamical system which has piecewise constant input and output segments can be represented by DEVS. We call this class of systems *DEVS-Representable*[5]. In particular, Differential Equation Specified Systems (DESS) are usually used to represent a natural system controlled by high-level, symbolic, event-driven control schemes. Sensing and actuation in such systems are event-like and they have piecewise constant input and output trajectories when viewed within the frame of their sensing/actuator interface. Consequently, within such an interface, the plant is representable by a DEVS. Likewise, the controller has natural DEVS representation.

Closure under coupling[3, 7] is a desirable property for subclasses of dynamical systems since it guarantees that coupling of class instances results in a system in the same class. The class of DEVS-representable dynamical systems is closed under coupling. This justifies hierarchical, modular construction of both DEVS models and the (continuous or discrete) counterpart systems they represent.

The DEVS formalism focuses on the changes of variable values and generates time segments that are piecewise constant. Thus an event is a change in a variable value which occurs instantaneously. In essence the formalism defines how to generate new

values for variables and the times the new values should take effect. An important aspect of the DEVS formalism is that the time intervals between event occurrences are variable in contrast to discrete time where the time step is a fixed number.

Independence from a fixed time step affords important advantages for modelling and simulation. Models where many processes operating on different time scales are present are difficult to describe when a common time granularity must be chosen to represent them all. Moreover, simulation is inherently inefficient since the states of all processes must be updated in step with this smallest time increment – such rapid updating is wasteful when applied to the slower processes. In contrast, in a discrete event model every component has its own control over the time of its next internal event. Thus, components demand processing resources only to the extent that their own intrinsic speeds or their response to external events dictate.

2.2 Parallel DEVS

Recently the DEVS formalism was revised to remove all vestiges of sequential processing to enable full exploitation of parallel execution[8]. We now review this structure.

A DEVS basic model is a structure:

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

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,

$\delta_{ext}(s, e, \phi) = (s, e)$.

$\lambda : S \rightarrow Y^b$: output function.

$ta : S \rightarrow R_{0+\rightarrow\infty}$: time advance function,

where $Q = \{(s,e)|s \in S, 0 < e < ta(s)\}$,

e is the elapsed time since last state transition.

DEVS models are constructed in hierarchical fashion by interconnecting components (which are DEVS models). The specification of interconnection, or coupling, is provided in the form of a coupled model. The structure of a *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 component names.

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,

$M_i =< X_i, S_i, Y_i, \delta_{inti}, \delta_{exti}, ta_i >$ is a DEVS basic 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.

The behavior of a coupled model is constructed from the behaviors of its compo-

nents and the coupling specification. The *resultant* of a coupled model is the formal expression of such behavior. Closure of the formalism under coupling is demonstrated by constructing the resultant and showing it to be a well defined DEVS. Such closure ensures that hierarchical construction is well defined since a coupled model (as represented by its resultant) is a DEVS model that can be coupled with other components in a larger model. Details of closure proof are given in [8]

3 Fire spread: DEVS modelling and simulation example

An example of fire growth in mountain environments will illustrate DEVS modelling and simulation. Wildland fires are a common phenomenon in forests and shrublands in mountainous terrain. Vegetation varies greatly in its combustibility. Topographic factors, particularly slope, greatly influence the direction and rate of fire spread. Topography also provides both barriers to fire spread (e.g. lakes) and channels facilitating its spread (e.g., dried river beds). Weather, particularly wind, is also a major driver of fire spread. The complexity of the fire propagation in mountain forests calls for powerful modelling methodologies able to handle spatial interaction over a heterogeneous landscape as well as temporal dynamics introduced by varying weather conditions. GIS can provide the spatially referenced data necessary to represent topography, weather, and vegetation state distributions. Spatial dynamic models are needed to project such states forward in time. However conventional differential equation formulations entail an enormous computational burden that greatly limits their applicability. By combining GIS, for state characterization, and DEVS, for dynamic state projection, we derive an approach that can achieve realism within feasible computational constraints, albeit in high performance environments.

Demonstration studies led by Vasconcelos [9, 10] employed the DEVS-Scheme Modelling and Simulation language linked to a GIS data base for Ivins Canyon in

the White Mountains of Arizona. The digitized vector files were rasterized to a grid of 75 rows by 76 columns, with a cell size of 1 acre. Weather data gathered at the fire camp were used to generate a set of weather-related map overlays. DEVS-Scheme is an object-oriented simulation environment that enables directly expressing models in the (original) DEVS formalism [4]. Atomic models are standalone modular objects that contain state variables and parameters, internal transition, external transition, output and time advance functions. Two state variables are standard in *atomic models*: *phase* and *sigma*. In the absence of external events the model remains in the current *phase* for the time given by *sigma* at which time an internal transition occurs. If an external event occurs, the external transition place the model in a new *phase* and *sigma* thus scheduling it for the next internal transition. The state entered upon an external (input) event depends not only upon the input and current state (including *phase* and *sigma*) but also upon the time that has elapsed in this state. The latter is critical to the ability of DEVS to faithfully represent the behavior of continuous systems though discrete events [11, 5].

For a set of component models, a *coupled model* can be created by specifying how the input and output ports of the components will be connected to each other and to the input and output ports of the *coupled model*. The new coupled model is itself a modular model and thus can be used as a component in a yet larger, hierarchically higher level model. For the simulation of fire growth in a cellular space one can envision the placement of an atomic model at each cell location. Thus there is an array of spatially-referenced models that correspond to distributed processors in the raster map lattice.

An atomic model embodies the behavior of an ignited cell and the propagation of this behavior to its spatial neighbors. Once a cell is ignited by contagion it burns until the fire has traversed the distance corresponding to that cell. The rate of spread

can be computed using conventional equations that take into account topography, vegetation fuel type, and weather as suggested above. Given this rate one can easily compute the traversal time in the direction of spread, Thus when an atomic model, initially in the unburned state, receives an ignition signal from a neighbor, the external transition function places it in the burning *phase* with a *sigma* set to the predicted burning time.

Construction of this model illustrates the general concepts of DEVS representation of continuous systems. Think of a continuously burning cell as base model from which a DEVS abstraction will be derived. The base model's state space is partitioned into output equivalent blocks — while the state trajectory remains in a block, the model's output remains constant. Internal events of the DEVS correspond to boundary crossings in the base model space. Given a state on a boundary the DEVS predicts the state that will be reached on the next boundary crossing and the time (*sigma*) to reach it(time to spread to neighboring cell). A change in the input received by the base model (such as ignition or weather shift) is mapped to an external event in the DEVS representation. To capture its effect on the base model state, the DEVS is able to utilize its elapsed time information(e.g., update the amount burned so far).

The area burned after 4 simulated hours matches quite closely that of the actual fire. However, as is to be expected, there are some significant differences. Many more validation experiments are needed before definitive model structures may emerge. Questions such as the dependence on resolution (cell size and number of phases considered) and the adequacy of propagation rules (e.g., can a head cell re-ignite a partially burned cell) must be addressed. Insight into these issues can be had by seeking conditions under which a homomorphism holds between base (high resolution) and lumped (low resolution) models[6].

Ultimately, the validity of a lumped model, and the methodology that generates

it, hinges on the agreement between model predictions and real world measurements. Linking DEVS with data rich GIS sources greatly facilitates this comparison. Moreover, the fact that the DEVS formalism can homomorphically map any system dynamic observed through a quantized output(partitioned state space) assures us that a error free model exists. The critical question with DEVS, as with any modelling methodology, is to how good an approximation to this ideal can we construct within the computational resources available. DEVS's focus on events is inherently efficient in both space and time on this score.

4 DEVS on high performance computers

Powerful workstations are becoming increasingly available at the same time that high costs and small commercial markets continue to limit general access to massively parallel supercomputers. Therefore, there is much interest in heterogeneous, distributed computing environments which exploit the full panoply of computing resources. Distributed computing refers to the coordinated activity of a network of processors. Heterogeneity refers to the variety of platforms that such processor nodes could assume ranging from general purpose desktop workstations of various powers, to special purpose hardware such as neural net simulators, to internet linked massively parallel computers. Thus a heterogeneous, distributed environment is a collection of modular, autonomous, and not necessarily homogeneous, processors that communicate with one another via message passing [12, 13, 14].

Distributed simulation schemes in the literature[15] typically require that each processor be able to execute autonomously with relatively infrequent message-passing. Because of this, almost all implementations of such algorithms have been on coarse grained, MIMD machines.

In designing a DEVS-based high performance simulation environment, our goal was

portability of models across platforms at a high level of abstraction. A DEVS model should not have to be rewritten to run on serial, parallel or distributed environment. Ideally, this invariance should apply at the high level of abstraction (set-theory) in which DEVS is formulated. However, a computational equivalent of this level does yet not exist (although efforts are beginning in that direction). Falling short of this ideal, but still significant, is the ability to port DEVS models written in the same computer language across platforms. There are numerous advantages to such portability. To name several that are especially important in this context: 1) models developed on a serial workstation, with all its comfortable development support, can be easily ported after verification to a parallel system for high performance production runs, 2) in a parallel/distributed environment, the same form of model description can be used for the interaction of model components executing within the (serial) nodes as for the (parallel) interaction of components executing on different nodes (more of this later).

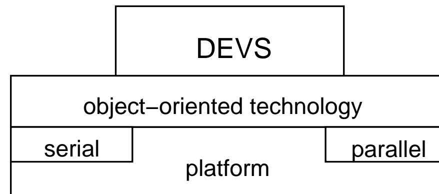

Figure 2: Object-Oriented Implementation of DEVS on Various Platforms

Object-oriented technology is the key to achieving DEVS portability objectives while retaining the flexibility to mitigate concomitant performance costs. Perhaps the most characteristic attribute of this technology is its ability to separate behavior from implementation, enabling distinct implementations of the same behavior to coexist[16]. As shown in Figure 2, DEVS is implemented in an object-oriented form which enables it to executed on serial or parallel platforms. The DEVS formalism is expressed as objects and their interactions with the details of the implementation (serial or parallel) hidden within the objects. The user interacts with only those in-

terfaces that manifest the DEVS constructs while being shielded from the ultimate execution environment.

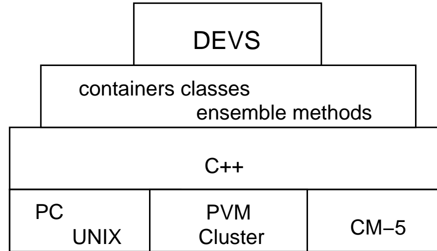

Figure 3: Implementation of DEVS using Containers Classes with C++

The approach is illustrated in greater detail in Figure 3. Due to its rapidly growing availability, C++ was employed as the target object-oriented language. As shown, DEVS is implemented in terms of a collection of classes, called *containers*. In their usual serial guise, such classes provide well-known means for defining list data structures and their manipulation. However, a more abstract and useful characterization of their functionality is that containers provide services to group objects into collections and coordinate the activity within such groups.

5 Specification and implementation of containers

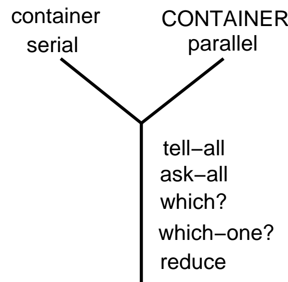

Figure 4: Five Primitives of Containers Classes

This is illustrated in Figure 4 which enumerates five basic primitives for coordinating behavior of objects in a container[16]. In outline:

- *tell-all* sends the same command to each object in a container.

- *ask-all* sends the same query to each object and returns a container holding the responses (which are also objects).
- *which?* returns the subcontainer of all objects whose response to a boolean query is TRUE.
- *which-one?* returns one of the objects in the container whose response to a boolean query is TRUE.
- *reduce* aggregates the responses of the objects in a container to a single response (e.g., taking the sum).

While these so-called *ensemble methods* may seem more parallel than sequential in nature, they have abstract specifications that are independent of how one chooses to implement them. Thus, using the polymorphism properties of C++ we define two classes for each abstract container class; one (lower-case) implementing the ensemble methods in serial form, the other (upper-case), implementing them in parallel form(Figure 4). The serial implementations run on any architecture that has a C++ compiler. In particular, if the nodes of a parallel or distributed system run C++, then the serial containers will work on them. However, the implementation of parallel CONTAINERS involves physical (as opposed to virtual) message passing among objects residing on different nodes. Such message passing must be implemented within the communications primitives afforded by the parallel/distributed system in question. For example, massively parallel CM-5 implementation employs CMMD(CM-5 message passing library). Likewise, a network of workstations linked together under PVM [17] offers the communication primitives supplied by PVM.

6 DEVS implementation over containers classes

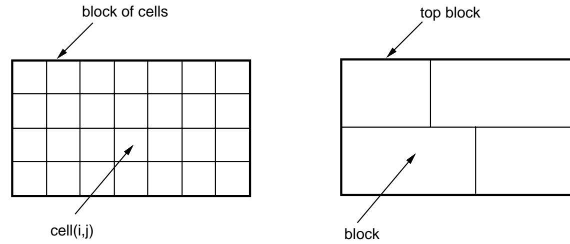

Figure 5: Hierarchical Construction of Block Models from Atomic Cell Models

To illustrate, consider a two-dimensional grid of cells as shown in Figure 5. The cells could be the atomic components in a landscape model of fire spread in Section 3. They are grouped into blocks and each block is assigned to a processor node. The closure property of DEVS guarantees that each block can itself be regarded as a DEVS model which can now be considered as a component model. These components are then grouped together to form a new DEVS model(shown as "top block") which is equivalent in behavior to the original.

Blocks are effectively containers whether they contain cells or (in the case of the top block) lower level blocks. Figure 6 illustrates the mapping of top block and the nodal blocks on CM-5 processing nodes. Since several coupled models exist, they are synchronized by message-passing primitives running in hostless mode. Each block mapped in each processing node can have variable number of cells. The arrow lines indicate direction of the container ensemble messages sent by top block or imminents contianer(see below).

In terms of ensemble methods, a cycle in a DEVS simulation can be outlined as follows:

1. *Ask all components for their addresses*: this stores the addresses in a container. Normally, this request need only be done once. However, one motivation for the containers-based formulation is that cells can be easily shifted from one node to another to balance computational load. An updating of components addresses

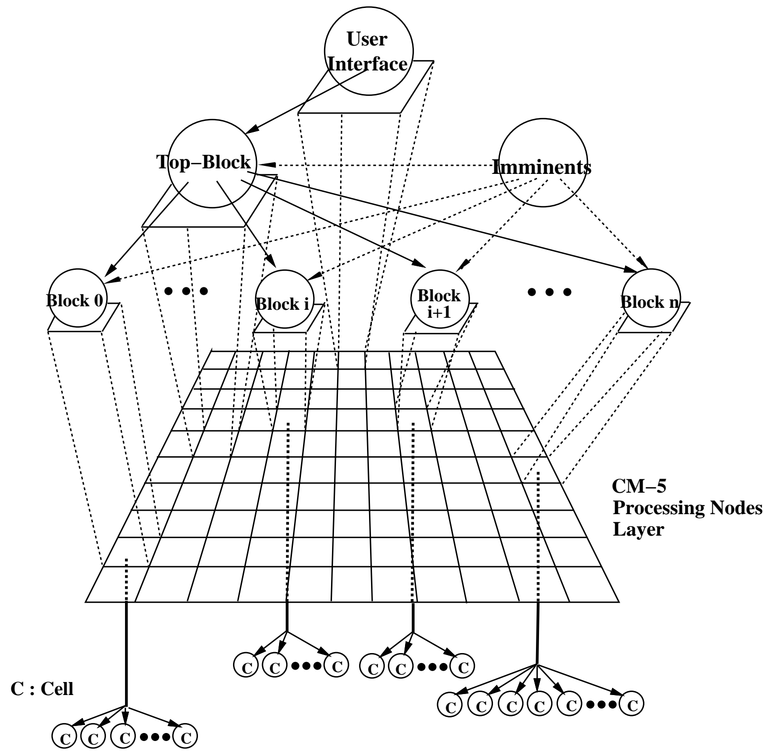

Figure 6: Mapping of Block Models to CM-5 Processing Nodes

is needed after each such load balancing operation.

2. *Compute the next event time*: this is done by a reduction which gets the minimum of the component times to next event.

3. *Determine the imminent components*: these are the components whose next event times are minimal. They are identified by using the *which?* ensemble method.

4. *Tell all the imminent components to sort and distribute their output messages called the mail.*

5. *Tell all the imminent components to execute their internal transition functions.*

6. *Tell all components with incoming mail to execute their external transition functions with their mail as input.*

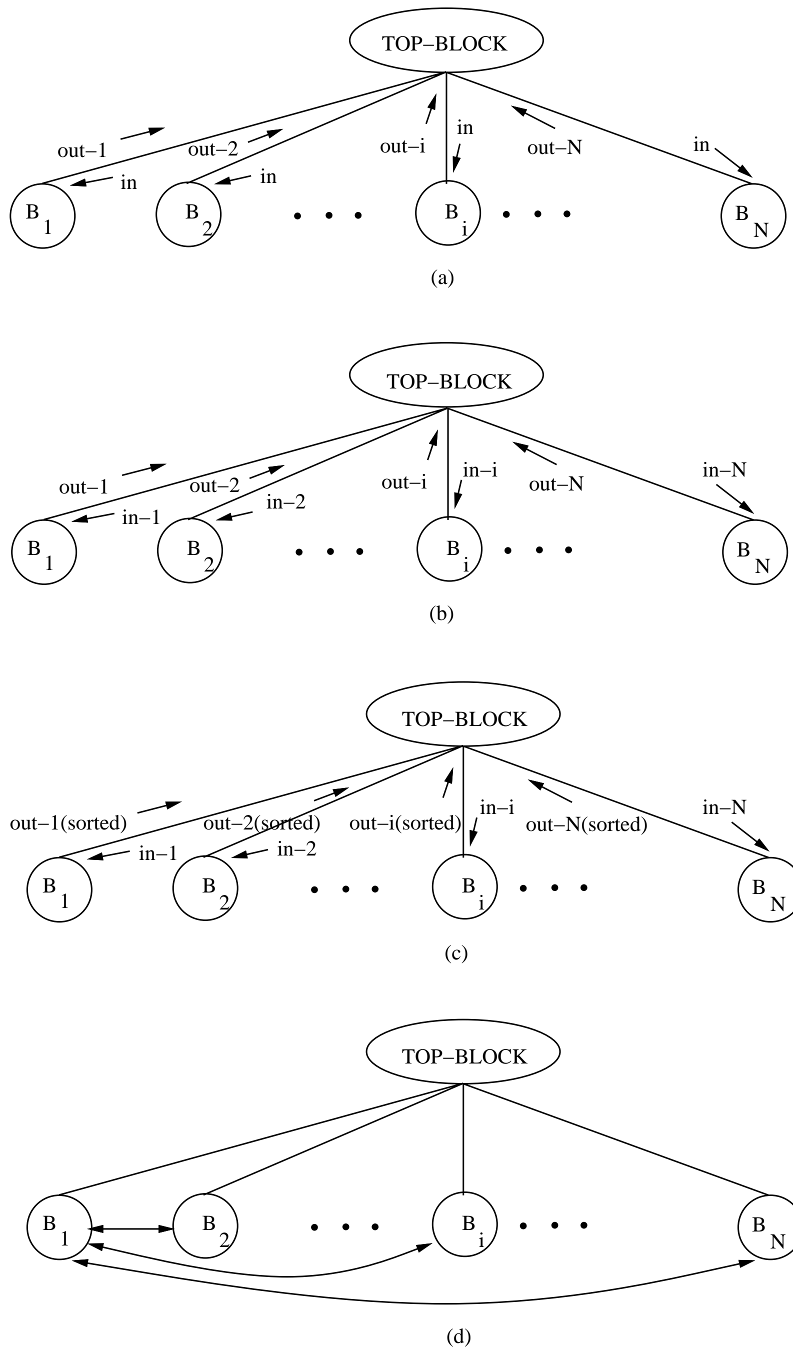

Figure 7: Four Different Mail Handling Schemes: (a) Centralized distribution and localized sorting in destination blocks, (b) Centralized sorting and centralized distribution, (c) Localized sorting in source blocks and centralized distribution, and (d) Localized sorting in source blocks and localized distribution

Figure 7 shows four different mail sorting and distribution schemes that we are considering for step 4.

1. *Centralized distribution and localized sorting in destination blocks*: top-block gathers the output messages from nodal blocks and send them to nodal blocks as one message, then each block selects the messages for its own cells.
2. *Centralized sorting and centralized distribution*: top-block gathers the output messages from nodal blocks and sorts them so that each block is sent only the mail designated for its cells.
3. *Localized sorting in source blocks and centralized distribution*: each source block(nodal) presorts the output mail so that the top-block can distribute the messages for nodal blocks with reduced sorting effort.
4. *Localized sorting in source blocks and localized distribution*: each source block sorts the output mail and directly sends them to destination blocks.

The efficiency of the above schemes will depend on the number of nodal blocks, the size of messages to be passed between blocks and the speed and topology of the communication network.

7 Watershed example

Besides distributed simulation, direct access to GIS data bases is a major requirement for computing environment capable of addressing complex ecological questions. The linkage was achieved by implementing a data handler object in C++ which processes data requests from models. The data object allows the model to be developed without knowledge of how the data is stored. The models can request information at a specific geographic location and data is returned in the form that the model requires.

Hierarchical GIS-based models expressed in DEVS can now be developed on a serial workstation. After verification, they can be mapped to a parallel/distributed environment by recompiling with the parallel version of containers. We have demonstrated this methodology in the development and execution of a GIS-based watershed model. The GIS databases that drive the hydrology simulation contain thematic layers for elevation, slope, aspect, vegetation, soils and other essential parameters. The databases come from a variety of sources, the primary watershed being the Walnut Gulch Experimental Watershed near Tombstone, Arizona. This is a heavily gauged and monitored watershed with over 40 years of precipitation and runoff data. The watershed covers over 100 square kilometers and is being modelled at 40 meter and 20 meter resolutions. This translates to 250,000 cells at the higher resolution.

We have developed a DEVS cell model for surface infiltration and runoff[18]. It has been tested with a database for Brown's Pond with a 180 by 180 rasterized grid, yielding a total of 32,400 cells. As far as we can tell, this is by far the largest number of cells that has been attempted in the hydrologic literature[19, 20, 21, 22]. In contrast to other models where channels are "built in" using distinct functional components, in our high resolution model, channel flows "emerge" from the underlying water dynamics and topography of the landscape. Figure 8 is an elevation map of Brown's Pond. Figure 9 shows the distribution of water depth at some time after a uniform rainfall. One can see that channels have formed that are clearly correlated with the topography.

8 Some benchmarking tests

Figure 10 compares execution times of sequential DEVS simulations on a single processing node of the CM-5 and a Sparc-1000 workstation. The CM-5 employs a Sparc-2 processor with 32 Mega Bytes of memory. The figure shows that not only is the Sparc-

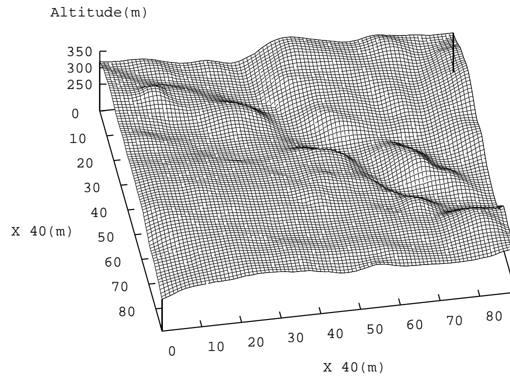

Figure 8: Brown's Pond Elevation Map

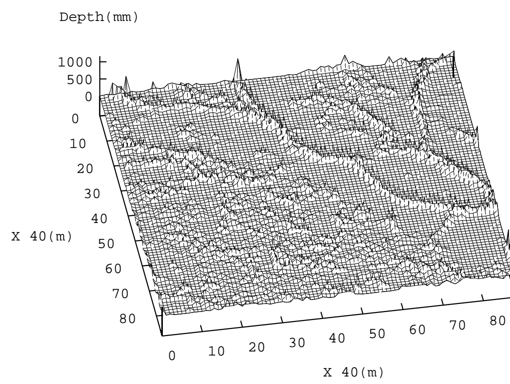

Figure 9: Brown's Pond Water Depth after 20 simulated hours(15 hours after end of 5 hour long rainfall)

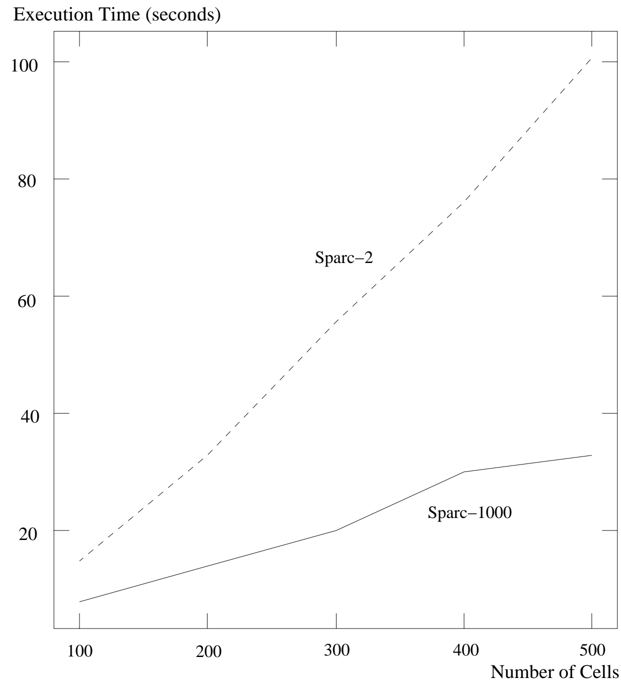

Figure 10: Sequential DEVS Execution Times on Single Processing Node of Sparc-2 and Sparc-1000

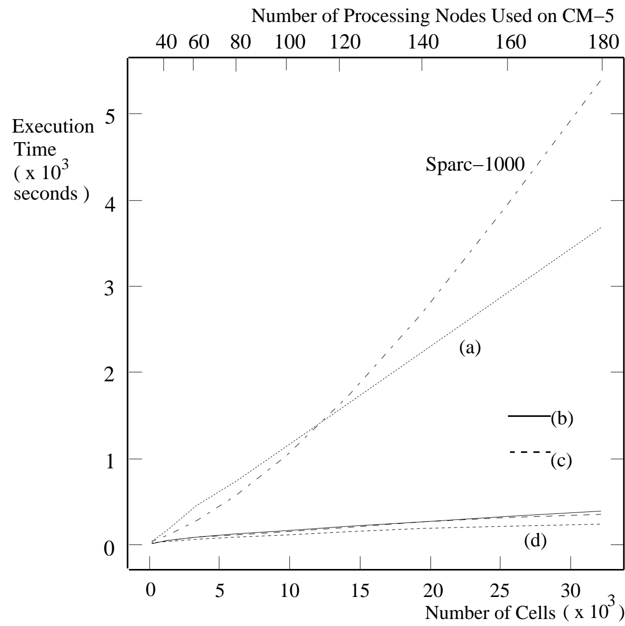

Figure 11: Execution Times of different algorithms on CM-5 and Sparc-1000 workstation with the Brown's Pond data

2 slower but also the ratio of execution times becomes greater as the number of cells increases due to the memory management and other factors.

Figure 11 shows the parallel execution times of four mailing schemes shown in Figure 7 on the CM-5. For comparison, also shown is sequential execution time on a Sparc-1000 machine. The mailing scheme (d), localized mail distribution, significantly outperforms other schemes. The number of processing nodes used on CM-5 increase in proportion to data set size.

A larger GIS data set, with 512x512 cells, was also simulated on CM-5. Figure 12 shows that simulation of a quarter million cells takes slightly more than half an hour, using scheme (d). For comparison we ran the same simulations on a PVM network of 4 workstations using the DEVS portability (section 4). Figure 12 shows that although, small simulations can be managed on such networks, simulation becomes impractical for models having more than 100,000 cells.

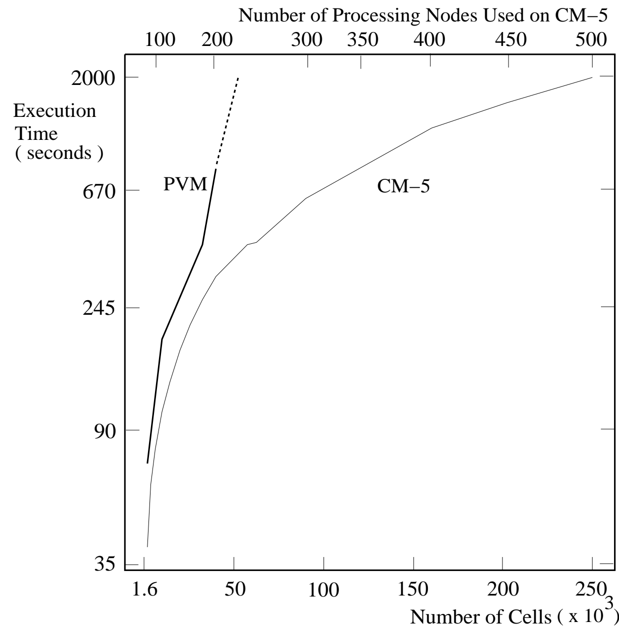

Figure 12: Experimental Result for large data size on CM-5

9 Conclusions

Although our efforts have been at the level of massively parallel computers and heterogeneous, distributed computing at the LAN level, the DEVS/containers architecture could be applied at the WAN level where the Army ADS (Advanced Distributed Simulation) is concentrated. It would be interesting to contrast this with approach with alternatives being developed. Mowbray et al[23] provide C++ classes to encapsulate interprocess communication mechanisms for discrete event simulation without the higher levels of containers and DEVS. We believe that more flexible, portable and theory-justified modelling can be obtained by adding in the latter levels.

References

[1] Y. C. Ho, "Special issue on discrete event dynamic systems", Proceedings of the IEEE, 77(1), 1989.

[2] B. P. Zeigler, *Theory of Modelling and Simulation*, John Wiley, New York, 1976.

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

[4] B. P. Zeigler, *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic press, San Diego, CA, 1990.

[5] B. P. Zeigler and W. H. Sanders, "Preface to special issue on environments for discrete event dynamic systems", Discrete Event Dynamic Systems: Theory and Application, 3(2):110–119, 1993.

[6] H. Praehofer and B. P. Zeigler, "Automatic abstraction of event-based control models from continuous base models", submitted to IEEE Trans. Systems, Man and Cybernetics, 1995.

[7] H. Praehofer, *System Theoretic Foundations for Combined Discrete-Continuous System Simulation*, PhD thesis, Johannes Kepler University of Linz, Linz, Austria, 1991.

[8] A. Chow and B. P. Zeigler, "Revised DEVS: A Parallel, Hierarchical, Modular Modeling Formalism", Proc. Winter Simulation Conf., 1994.

[9] M. J. Vasconcelos, J. M. C. Pereira and B. P. Zeigler (1992), "Simulation of Fire Growth in Mountain Environments", in: Mountain Environments and GIS, eds: M. F. Price and D. I. Heywood, Taylor Francis, pp. 167-186, 1994.

[10] M. J. Vasconcelos and B. P. Zeigler (1993), "Simulation of forest landscape response to fire disturbances", Ecological Modelling, 65, pp. 177-198, 1993.

[11] B. P. Zeigler, "DEVS representation of dynamical systems: Event-based intelligent control", Proceedings of the IEEE, 77(1):72–80, 1989.

[12] J. N. Gary, "An Approach to decentralized computer systems", IEEE trans. on Software Engineering, vol. 12, no. 2, pp. 135-149, 1990.

[13] L. Gasser, "Distributed Artificial Intelligence", AI Expert, pp. 26-33, July, 1989.

[14] V. V. Dixit and D. I. Moldova, "The allocation problem in parallel production systems", Journal of Parallel Distributed Computing, vol. 8, no. 1, pp. 20-29, 1990.

[15] R. M. Fujimoto, "Parallel discrete event simulation", CACM, vol. 33, pp. 30-53, 1990.

[16] Y. K. Cho, *Parallel Implementation of Container using Parallel Virtual Machine*, Master thesis, The University of Arizona, Tucson, AZ, 1995.

[17] V. S. Sunderam and *et. al.*, "The PVM Concurrent Computing System: Evolution, Experience, and Trends", Parallel Computing, vol. 20, no. 4, pp. 531-545, 1994.

[18] B. P. Zeigler, Y. Moon, V. L. Lopes and J. Kim, "DEVS Approximation of Infiltration Using Genetic Algorithm Optimization of a Fuzzy System", submitted to Mathematical and Computer Modeling, 1995.

[19] M. B. Abbot, J. C. Bathurst, J. A. Cunge, P.E. O'Connell and J. Rasmussen, "An Introduction to the European Hydrological System - Systeme Hydrologique European, 'SHE'. 1. History and phylosopy of a physically-based, distributed modeling system", Journal of Hydrology, 87:45-59, 1986.

[20] C. V. Alonso and D. G. De Coursey, "Small watershed model", In: D. G. De Coursey(editor), Proc. of the Natural Resources Modeling Symposium, PingreePark, CO, USDA-ARS-30, pp. 40-46, 1985.

[21] L. E. Band and E. F. Wood, "Strategies for large scale distributed hydrologic simulation", Applied Mathematics and Computation, 27(1) pp. 23-38, 1988.

[22] K. J. Beven, "Spatially distributed modeling: conceptual approach to runoff prediction", In: P. E. O'Connel and D. S. Bowles(eds), Recent advances in the modeling of hydrologic systems, NATO ASI Series, Kluwer Academic Publishers, 373-387, 1991.

[23] D. W. Mowbray, J. W. Wallace, A. L. Herman, and E. S. Hirschorn, "An Architecture for Advanced Distributed Simulation", Phalanx, June 15, pp. 12-15, 1995.

Contents

1 High performance simulation **2**

2 Layered architecture **3**

2.1 The DEVS formalism 3

2.2 Parallel DEVS 5

3 Fire spread: DEVS modelling and simulation example **7**

4 DEVS on high performance computers **10**

5 Specification and implementation of containers **12**

6 DEVS implementation over containers classes **13**

7 Watershed example **17**

8 Some benchmarking tests **18**

9 Conclusions **21**

List of Figures

1 Layered Representation of Simulation-based Decision Making 3

2 Object-Oriented Implementation of DEVS on Various Platforms . . . 11

3 Implementation of DEVS using Containers Classes with C++ 12

4 Five Primitives of Containers Classes 12

5 Hierarchical Construction of Block Models from Atomic Cell Models . 14

6 Mapping of Block Models to CM-5 Processing Nodes 15

7 Four Different Mail Handling Schemes: (a) Centralized distribution and localized sorting in destination blocks, (b) Centralized sorting and centralized distribution, (c) Localized sorting in source blocks and centralized distribution, and (d) Localized sorting in source blocks and localized distribution 16

8 Brown's Pond Elevation Map 19

9 Brown's Pond Water Depth after 20 simulated hours(15 hours after end of 5 hour long rainfall) 19

10 Sequential DEVS Execution Times on Single Processing Node of Sparc-2 and Sparc-1000 20

11 Execution Times of different algorithms on CM-5 and Sparc-1000 workstation with the Brown's Pond data 20

12 Experimental Result for large data size on CM-5 21