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

SAM - A Computer Aided Design Tool for Specifying and Analyzing Modular, Hierarchical Systems

Arturo I Concepcion
Stephen J. Schon
Department of Computer Science
Michigan State University
E. Lansing, MI 48824

ABSTRACT

This paper presents SAM, a computer aided design tool for specifying and analyzing modular, hierarchical systems. SAM is based on Discrete Event System Specification (DEVS) and it uses generic components for specifying coupling relationships among components. The objectives of this design tool are to provide an environment for a user to specify and design systems with ease, and to allow the reuse of previously specified models to build new ones. The later promulgates increased productivity and the applicability of modular approach in the development of complex systems. Furthermore, the design tool analyses a specified system for interface errors which facilitates early testing of the specification before the model is translated into a simulator. The paper discusses how the objectives of SAM are achieved and what are the implementation issues involved. It is envisioned that the specification produced by SAM will be used to map the model onto a network of processors for distributed simulation.

1. INTRODUCTION

Distributed computer systems (DCS) offer several advantages over Von Neuman architecture computer systems. Among the advantages of DCS are sharing of resources, increased reliability and concurrent execution of processes. DCS encompasses many applications, such as the communication subnet, the distributed operating system and the distributed databases. Included in these applications is distributed simulation (Stankovic 1984). Distributed simulation is a form of distributed computation where the simulation process is partitioned and the partitions are run on the DCS. This area of research has generated a lot of interests due to the availabilitv of DCS. Examples of algorithms that implement distributed simulation are the Time Warp Mechanism (Jefferson et. al. 1985), the Asynchronous Distributed Simulation (Chandy and Misra 1981), the Active Logical Processes (Reynolds 1983), and the Hierarchical Abstract Simulator (Concepcion 1985a). Refer to Concepcion and Ziegler (1985) for a complete survey of distributed simulation algorithms and architectures.

However, the writing of distributed programs that carry out simulation instructions is not an easy task. Where different processes communicate and synchronize via message passing, interface errors are common mistakes. Deadlock may occur when there are no restrictions in the waiting and holding of processes. Computer aided design tools can help in making the task of designing distributed programs for distributed simulation easier.

Automated tools for specification of software design has long been used to analyze the consistency and organization of specifications. Examples are SREM (Davis and Vick 1977) and PSL/PSA (Teichroen and Hershey, 1977). The HOS specification language (Hamilton and Zeldin 1976), a methodology based on formal axioms, is used to verify very large software project against interface errors. A commercial version of HOS (Mimno 1982) specifies the software via a graphics interface and after verfication is done by the analyzer, the code is automatically generated from the verified specification. A more recent work is SARA (Estrin et. al. 1986) which provides an environment for modeling, analysis and simulation of concurrent systems. The graphics interface of SARA provides a powerful and helpful interface with a user. Using formal graph model, SARA is able to derive behaviors of concurrent systems in 3 domains: control flow, data flow and interpretation. In modelling and simulation, several software tools have been developed. The use of graphics in simulation is a powerful interface which makes the use of the simulation package easier to interact with and its output more understandable. Simulation languages such as SLAM (Pritsker and Pegden 1979) and SIMAN (Pegden, Miles and Diaz 1985) make use of graphics to write programs in block diagrams and depict output as animations for easier analysis. GIST (Sinclair, Doshi and Madala 1985) is a performance evaluation tool for the specification and simulation of extended queueing network models of computer systems. The software tool provides a graphical interface for specifying a system pictorially and a textual interface which provides the same modelling capabilities but uses menu-driven and window approaches for non-graphic terminals. In another approach, specification languages were developed to aid in analyzing and developing simulation programs for discrete event models (Melman and Livny 1984, Overstreet and Nance 1985).

An area that has generated much interest is in the reuse of software components. This improves software productivity by using previously developed and tested software. An approach was presented in Polster (1986) wherein partial systems were developed by reusing general software for a given application area. Heuristics were used in selecting only those program segments which can be used as building blocks. Another approach is to maintain a database of information of software components which might include the following: code, documentation, specification, requirements and project status (Goguen 1986). A language was developed on how to use the database and interconnect software components.

This paper presents the Specifier and Analyzer Module (SAM), a computer aided design tool, which provides an environment for specifying, designing and analyzing discrete event systems for distributed simulation. SAM is based on Discrete Event System Specification (DEVS) which was defined by Zeigler (1984) and extended to facilitate modular, hierarchical model specification (Zeigler 1985, Concepcion 1985c).

The paper is organized as follows. A review of DEVS and its generic components, which are used for coupling components together under this formalism, is given in section 2. Section 3 discusses the objectives of SAM. Then in section 4, the implementation issues of SAM are presented. This section will contain the inter-relationships between the different components making up SAM. In section 5 conclusions and future work and the directions of the research are provided. Finally, the appendix is included to illustrate system design using SAM.

2. DEVS AND GENERIC COMPONENTS

DEVS provides a fundamental set theoretic framework for representing discrete event models via a composition tree hierarchical specification. The following is a review of DEVS as a formalism for model based distributed simulation; see Zeigler (1985) for more detail.

The system to be specified is viewed as a modular multicomponent DEVS. Each component is affected by external events produced by its environment (other components in the system). In addition, internal events which are generated within a component produce output that constitutes an external event to other components in the system. Thus the components in a modular multicomponent DEVS communicate via message passing. The DEVS fomalism allows each component to be further decomposed into modular multicomponent DEVS where each decomposition represents a level of specification of the component. Therefore, the structure of the specification is a hierarchy where leaf nodes are the atomic (cannot be decomposed anymore) constituents of the system to be specified and each nonleaf nodes are coordinators of subtrees it manages. The subtree represents a non-atomic constituent of the system to be specified. The specification of a system using DEVS has the following advantages:

- the resulting specification can be proven to correctly represent discrete event models.
- systems can be specified using either top-down or bottom-up approach which facilitate modular hierarchical decompositions.
- parallelism can be exploited in the translated simulator.

The multicomponent DEVS has been extended by Concepcion (1985c) to explicitly represent timing and delay. This has been accomplished by introducing generic components which obey the DEVS formalism. These components are atomic and they perform the specific function of specifying timing and delay constraints on the system. The generic components also specify how the set of input events from a component are collected to a single input to another component or how a single output from a component is distributed to each component that needs the output. These definitions of collection and distribution of input/output events establishes a basis for detecting interface errors in the multicomponent DEVS. Generic components were derived in Concepcion (1985c) to perform the functions mentioned above.

The following generic components are used for specifying timing and delay aspects of multicomponent DEVS,

(a) *Synchronizer*, SYNC($X_1,X_2,...,X_n,Y_1,Y_2,...,Y_n$), whose function is to wait for all incoming events, $X_1,X_2,...,X_n$, to arrive. When the last incoming event arrives, it produces exactly the same incoming events on its output and waits for the next set of incoming events.

(b) *Delay*, DELA(X,Y,d), whose function is to delay an incoming event X by an amount of d time units.

The following generic components are used to specify the collection and distribution of input/output events in a multicomponent DEVS,

(a) *Abstractor*, ABST($X_1,X_2,...,X_n,Y$), whose function is to compute the cross product of the input events, $X_1,X_2,...,X_n$, to form a single output event Y.

(b) *Projector*, PROJ($X,x_1,...,x_m,Y_1,...,Y_n,E_1,...,E_n$), whose function is to decompose a given input X into components $Y_1,...,Y_n$. The input X consists of $X_1,...,X_m$ and produces the output, $Y_i = X_{j \in E_i}$ where $E_i \subseteq \{1,2,...,m\}$.

(c) *Selector*, SELE($X_1,X_2,...,X_n,Y$), whose function is to select one input event X_i as its output Y, where X_i is the only one of the events occurring.

(d) *Replicator*, REPL($X,Y_1,...,Y_n,E$), whose function is to duplicate the input event X to each of the output events specified by the parameter E, where $E \subseteq \{1,2,...,n\}$.

The behaviors of these generic components are fully specified by expressing them as models within the DEVS formalism (Concepcion 1985c). Thus a multicomponent DEVS consists of atomic and nonatomic components, generic components, and couplings between these components. Using coupling relationships (via generic components) among components, synchronization and intercommunication can be fully specified. Algorithms for the synchronization and intercommunication between components were implemented on the HEP computer (Concepcion 1985c).

3. OBJECTIVES OF SAM

The objectives of SAM are as follows:

- to provide a good interface for specification and design of discrete event models.
- to provide a facility to reuse previously specified model components.
- to provide a capability to perform interface error checks on the specification.

The first objective is achieved by the implementation of graphics and window interfaces. The specification via pictures provides the user with the ease in performing his tasks. Documentation is included to describe the component together with its input/output specifications.

To provide the reuse of previously specified model components, a file management system is implemented. The file management system stores the specifications of a component and its decomposition, if non-atomic. Then by a retrieval procedure, a stored model component can be integrated into the user's working model. The user, therefore, has the ability to reuse components to build new ones. This facility increases the user's productivity.

Finally, the specified system can be tested for missing parts and for interface errors prior to translating the model to a simulator for execution. The missing parts may represent unspecified components or unspecified connections between components of the system. Interface errors are errors whereby the type and structure of the message sent by a component does not match the type and structure of the message expected by a receiving component. SAM checks the input/output specification of each component to see if these match with the input/output specifications of the component's environment (other components).

4. IMPLEMENTATION ASPECTS OF SAM

SAM consists of three components: the Component Building Unit, the File System Unit and the I/O Specification Unit. Refer to Fig. 1 for an illustration of SAM's architecture. The tool distinguishes between two kinds of components: atomic and nonatomic. Atomic components break down further into generic (built-in) and nongeneric (user supplied) components. It is the job of the tool to allow the specification by the user of nongeneric atomic and all nonatomic components.

The Component Building Unit of the tool was implemented using the suntool/sunwindow environment on the Sun microcomputer; it is responsible for the construction of components graphically and it draws upon the underlying File System Unit for its processing. The File System Unit is

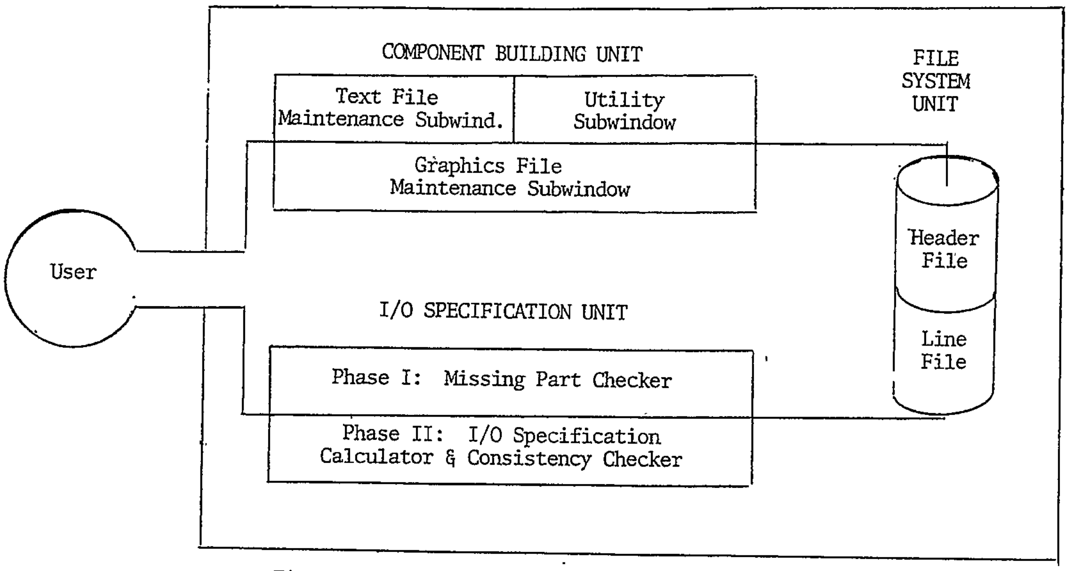
COMPONENT BUILDING UNIT
Text File Maintenance Subwind.
Utility Subwindow
Graphics File Maintenance Subwindow
FILE SYSTEM UNIT
Header File
Line File
User
I/O SPECIFICATION UNIT
Phase I: Missing Part Checker
Phase II: I/O Specification Calculator & Consistency Checker

Figure 1: Architecture of Specifier and Analyzer Module (SAM)

responsible for storing, retrieving, and deleting specified components. The information pertinent to a component that is handled by the File System Unit is keyed on the component name and version number, thus allowing several different versions of the same (generally) component. Also, a subcomponent number is assigned to each of the subcomponents that make up a nonatomic component, thus allowing the same component to be used multiple times in the building of another component. As a separate utility of the tool, the I/O Specification Unit is provided to perform a recursive error checking function, input/output specification calculation and I/O specification consistency check over a system of components.

4.1. THE COMPONENT BUILDING UNIT

The Component Building Unit revolves around three subwindows, the user interface through the keyboard and a mouse and an interface through a pop-down command menu. Through the keyboard the user is able to maintain component subcomponent-independent information by interaction with the Text File Maintenance Subwindow. And through the keyboard, the mouse and through commands offered on a pop-down menu, the user is able to maintain component subcomponent information by interaction with the Graphics File Maintenance Subwindow. The Utility Subwindow serves miscellaneous purposes such as providing verification notices for some commands and allowing "quick" data entry also necessary for interaction with the Graphics File Maintenance Subwindow. Refer to Fig. 2 and the appendix for an illustration of the user environment.

When the user attempts to use the tool to build a new component, the component must be specified as atomic or nonatomic through the Text File Maintenance Subwindow. If the component is atomic then the user is expected to enter the I/O specifications and the name of the object file of a given format that will be used to represent the internal workings of the atomic component. If the component to be built is nonatomic then the user is allowed to graphically build the internals of the component through the placement of its various subcomponent, including line connectors. In the event that the component already exists from previous tool use, The Component Building Unit calls on the File System Unit to retieve all existing component information.

In the process of building the internals of a component graphically, the user can call upon the following group of commands on a pop-down menu: get a subcomponent, remove a subcomponent, decompose a subcomponent, turn line mode on, turn line mode off, delete component information, and exit the Component Building Unit.

4.1.1. Subcomponent Getting and Removing

If the user chooses to "get" a subcomponent then he positions himself with the mouse-controlled cursor at the location he wants the subcomponent placed and gives the proper command. He is then asked through the Utility Subwindow to supply a combination of the following information: subcomponent name, subcomponent version, number of input ports and number of output ports. If the subcomponent is a nongeneric component then the name and version must be given and if it is a generic component then the subcomponent name (SYNC, DELA, ABST, PROJ, SELE, or REPL) and the number of input or output ports, as applicable, must be given. The Component Building Unit leaves the verification of the existence of the subcomponent to the error checking facilities of the I/O Specification Unit and it graphically generates the subcomponent.

If the user chooses to remove a subcomponent then he must simply issue the command and give verification, which is indicated through the Utility Subwindow.

4.1.2. Decomposition of Subcomponents

The user can choose to decompose a nongeneric subcomponent by putting the mouse on that subcomponent and issuing the "decompose" command. The system then generates a new tool instance where that subcomponent becomes the main component of the tool and the user is then allowed to interact with the current set of tools simultaneously.

4.1.3. Line Control

Lines are subcomponents used to connect other subcomponents, they are drawn graphically by the user through the use of mouse buttons, and they are removed through the "remove" subcomponent command discussed previously. Additionally, the "line mode off" command disconnects the line drawing facilities from the mouse buttons and the "line

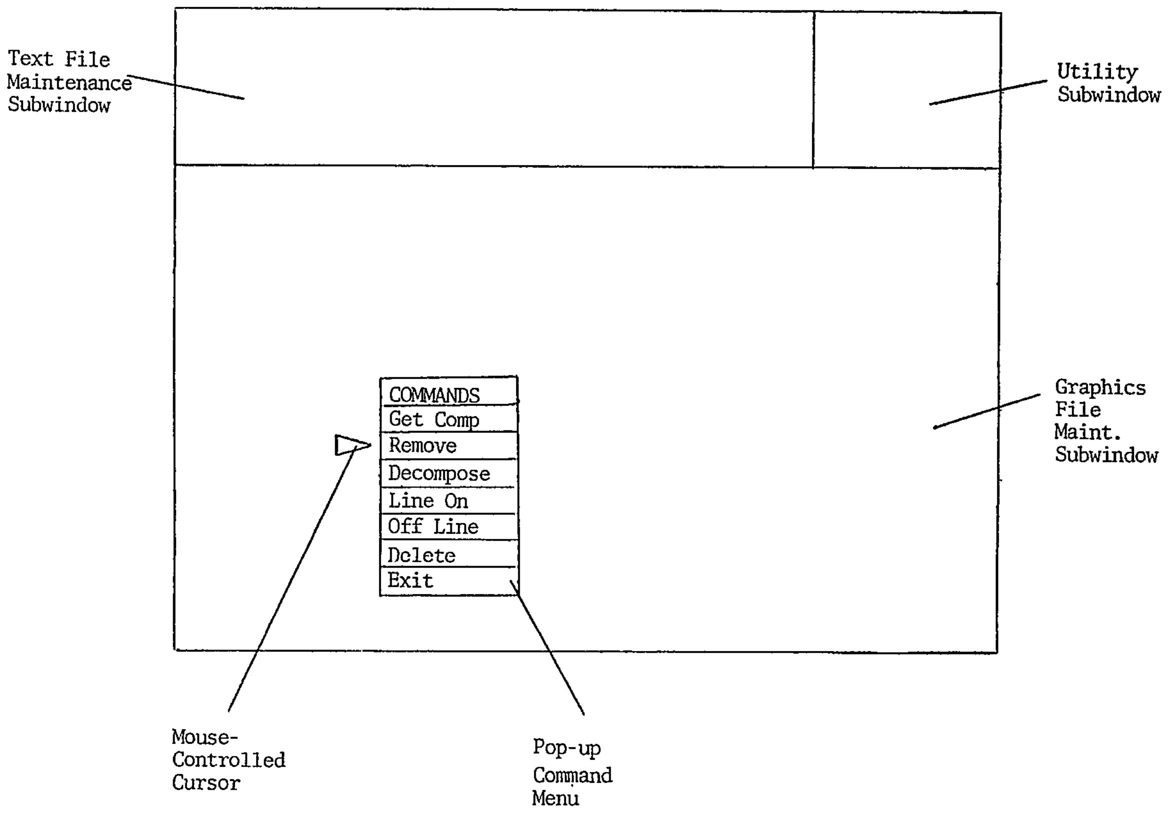

Figure 2: A User's View of SAM

mode on" command connects the line drawing facilities to the mouse buttons. Initially the system is in line mode and a user should have no reason for setting line mode off unless the mouse buttons required for line processing were needed for other tool functions (presently there are no other functions of the tool using those buttons).

When drawing lines, the user many set or unset line beginning markers at any unused subcomponent port. The second line marker set then defines the line. The system requires that the first line marker set must be on an existing port, but the second line marker set does not have to be. Any lines that do not connect two ports and that are not extremely small are drawn extending straight vertically or horizontally, depending on if the largest change between markers occured on the X or the Y axis.

4.1.4. Deleting Component Information

The user may choose to delete the overall component (its textual and graphical information) through the "delete" component command. Upon verification through the Utility Subwindow, the Component Building Unit calls upon the File System Unit to delete all pertinent records and the user is then allowed to exit from the tool.

4.1.5. Exiting a Component

The user may choose to exit the tool and if the Component Building Unit finds no existing decomposed children then it allows the user to verify the action before proceeding. Upon verification, the unit calls upon the File System Unit to save all existing information pertinent to that component and the user is then allowed to exit from the tool.

4.2. THE FILE SYSTEM UNIT

The File System Unit consists of two files: the Header File and the Line File. The Header File contains one record for each nonatomic component and one record for each nongeneric atomic component and each record is made up of the information that is maintained in the Text File Maintenance Subwindow of the Component Building Unit. The information maintained in the Header File is the information pertinent to a component and independent of the specifics of its subcomponents. This information includes the component name and version (which make up the key), a description of the component, input and output type specifications, an indicator of whether or not the component is atomic, and the name of the object module that represents the internal workings of the nongeneric atomic component (if applicable). Fig. 3 lists the information kept in a Header File record.

KEY	Component	Name Version
DATA	Component	Description Input Specification Types Output Specification Types Atomic Indicator (Y/N) Object Module Name

Figure 3: Header File Record Format

The Line File contains one record for each subcomponent of a nonatomic component, where a subcomponent can be another nonatomic subcomponent, a generic component, or a line used for connecting other subcomponents. The information kept in a record specifies the subcomponent number, name, version, center X and Y coordinates, number of input and output ports, and information about the I/O specifications of a port and what it is connected to. Fig. 4 lists the information kept in a Line File record.

```
KEY   Component Name
               Version
      Subcomponent Number

DATA  Subcomponent Name
                   Version
                   Center X Coordinate
                   Center Y Coordinate
                   Number of Input Ports
                   Number of Output Ports

                   For Each Port:
                   Specification Types
                   Connected To Number
                                Port Number
                                Port Type
```

Figure 4: Line File Record Format

4.3. THE I/O SPECIFICATION UNIT

The I/O Specification Unit is a separate utility of the tool that allows the user to check a multilevel system of components (the user specifies the root component in using this unit) for errors in construction. This unit breaks down into Phase I, which is responsible for detecting any missing parts of the system, and Phase II, which is responsible for calculating I/O Specifications for generic and nonatomic components and for checking the consistency of I/O specifications over the entire system of components.

4.3.1. Phase I

Phase I will detect a component subcomponent that has no record in the Header File, a nonatomic component that has no subcomponents (and thus has no records in the Line File), missing I/O Specifications, and an incorrect number of "open" input or output connections. Note that I/O Specifications must be given for the root component, all nongeneric atomic components and for any nonatomic components without subcomponents. All error listed above are fatal errors that prevent the system from continuing on to Phase II before their correction, except the error indicating that a nonatomic component has no subcomponents. If a nonatomic component with no subcomponents has I/O specifications then a warning is given and the system proceeds treating that component as if it were atomic. If no I/O specifications are given for the component then it is a fatal error. Note also that each nonatomic component with subcomponents must have exactly one "open" input port and one "open" output port for connection to other components, or else a fatal error occurs.

Phase I uses work files to proceed through a breadth-first processing of the component tree, eliminating connection lines (in favor of direct connections) and checking components at each level for the mentioned errors.

4.3.2. Phase II

Since the input/output specifications are given by the user for the root component and for all nongeneric atomic components, in Phase II the I/O specifications for generic components and for nonatomic components can be calculated and the consistency of I/O specifications over the entire system of components can be checked.

Once Phase I has been completed without the detection of any fatal errors, Phase II can proceed to use a propagation method of determining the I/O specifications of generic components (through help from their definition) and of nonatomic components. As long as a generic component without determined specifications is not connected on input to another generic component without determined specifications, the I/O specifications of that generic component can be calculated. Since there is a "left most" generic component whose input specifications must be able to be determined (if nothing else, as the I/O specifications of the root component), there is a guarantee of eventual propagation of specifications to all generic components. Similarly, the I/O specifications of nonatomic components can be calculated in a left to right fashion so that eventually all nonatomic component I/O specifications can be determined.

After all I/O specifications for generic and nongeneric components have been calculated, Phase II can simply look at the specification types of a connected input-output pair and mention any mismatches or discrepencies.

5. CONCLUSION AND FUTURE WORK

This paper presents the feasibility of an implementation of a computer aided design tool, the Specifier and Analyzer Module (SAM), for distributed simulation. The advantages gained in using this tool are that it allows the specification of discrete event systems using graphics, tests the specifications for consistency of I/O relationships and provides the storage and retrieval of reusable components. By providing a good environment for the specification of discrete event models, the time for developing models can be greatly reduced. The graphics interface provides a powerful tool for a user to interact with SAM while the file management system gives the user the capability to compose models from previously specified model components. Complex models can therefore be made in a modular and layered manner whereby system specifications can be built on top of previously tested model components. Testing for interface errors can be done easily and pinpointing of errors more accurately determined.

In each of the three components of SAM there is room for future work that would offer the user more freedom. In the Component Building Unit the limit of user workspace and component size (in terms of immediate subcomponents) could be relaxed with a "rolling screen" function and plenty of memory, and new commands could be added that would allow such things as switching from the building of one main component to another, inquiring on component sets in the file system and printing components sets. The File System Unit could allow the faster access of records and the access of records with varying amounts of information in them (in relation to the number of input and output ports of generic components, the input and output specification information, etc.). The I/O Specification Unit could allow more freedom to the user in indicating which parts of the system he would like to specify and analyze. Additionally, the form of the input/output specifications could forever be more elaborate.

A separate research is being undertaken by the authors to link the output from SAM to the Execution Module. This module takes in the specification of the discrete event system from SAM and translates the specification into C programming language code. The Execution Module then maps the different components of the distributed program onto a network of processors. These are then run concurrently. The Execution Module will be implemented on a network of Sun Workstations connected via an Ethernet where processes communicate using UNIX 4.3 BSD interprocess commnunication primitives. With the integration of the SAM and the Execution Module into one system, a user will be able to design, specify, analyze and then run the generated distributed program on the DCS.

APPENDIX

The partial design of a multiple level system component, the P (for "Partial") System component, using SAM is illustrated.

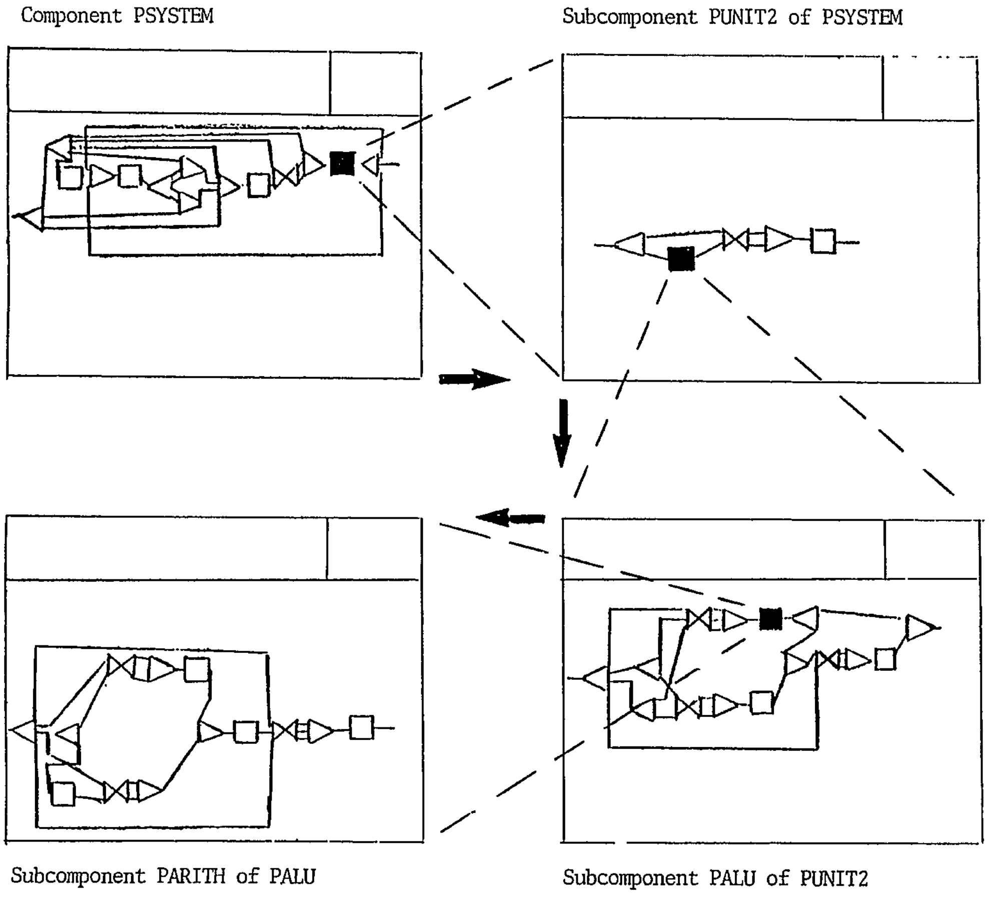
Component PSYSTEM
Subcomponent PUNIT2 of PSYSTEM
Subcomponent PARITH of PALU
Subcomponent PALU of PUNIT2

Figure A: Design of the P System Component Using SAM

REFERENCES

Chandy, K.M. and Misra, J., "Asynchronous Distributed Simulation via A Sequence of Parallel Computations", *Commications of the ACM,* Vol. 24, No. 11, Apr 1981, pp. 198-206.

Concepcion, A. I., "The Implementation of the Hierarchical Abstract Simulator on the HEP Computer", In *Proceedings of the 1985 Winter Simulation Conference,* Dec 1985, pp. 428-434.

Concepcion, A. I. and Zeigler, B. P., "Distributed Modelling and Simulation: A Review", Tech. Rep. *MSU-ENGR-85-027,* Dept. of Computer Science, Michigan State University, E. Lansing, MI.

Concepcion, A. I., "DEVS Formalism: A Framework for Hierarchical Model Development", Tech. Rep. *MSU-ENGR-85-028,* Dept. of Computer Science, Michigan State University, E. Lansing, MI.

Davis, C. and Vick, C., "The Software Development System", *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, Jan 1977, pp. 69-84.

Estrin, G. et. al., "SARA (System Architects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems", *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 2, Feb 1986, pp. 293-311.

Goguen, J. A., "Reusing and Interconnecting Software Components", *IEEE Computer,* Feb 1986, pp. 16-28.

Hamilton, M. and Zeldin, S., "Higher Order Software- A Methodology for Defining Software", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, Mar 1976, pp. 9-32.

Jefferson, D. et. al., "Implementation of the Time Warp on the Caltech Hypercube", In *Proceedings of the Conference on Distributed Simulation*, San Diego, CA, Jan 1985, pp. 70-81.

Melman, M. and Livny, M., "The DISS Methodology of Distributed System Simulation", *Simulation*, Apr 1984, pp. 163-176.

Mimno, P., "A New Technology for Mathematically Provable Software", *Computer World*, Oct 1982.

Overstreet and Nance, "A Specification Language to Assist in Analysis of Discrete Event Simulation Models", *Communications of the ACM*, Vol. 28, No. 2, Feb 1985, pp. 190-201.

Pegden, L. A., Miles, T. I. and Diaz, G. A., "Graphical Interpretation of Output Illustrated by a SIMAN Manufacturing System Simulation", In *Proceedins of the 1985 Winter Simulation Conference*, Dec 1985, pp. 244-251.

Polster, F. J., "Reuse of Software Through Generation of Partial Systems", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 3, Mar 1986, pp. 402-416.

Pritsker, A. A. and Pegden, C. D., "Introduction to Simulation and SLAM", *Halsted Press Book*, New York, 1979.

Reynolds, "Active Logical Processes and Distributed Simulation An Analysis", In *Proceedings of the 1983 Winter Simulation Conference*, Washington, D.C., pp. 263-265.

Sinclair, J. B., Doshi, K. A. and Madala, S., "Computer Performance Evaluation with GIST: A Tool for Specifying Extended Queueing Network Models", In *Proceedings of the 1985 Winter Simulation Conference*, Dec 1985, pp. 290-299.

Stankovic, J. A., "A Perspective on Distributed Computer Systems", *IEEE Transactions on Computers*, Vol. C-33, No. 12, Dec 1984, pp. 1102-1115.

Teichroen, D. and Hershey, E., "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, Jan 1977, pp. 41-48.*

Zeigler, B. P., "Multifacetted Modelling and Discrete Event Simulation", *Academic Press*, New York, 1984.

Zeigler, B. P., "Discrete Event Formalism for Model Based Distributed Simulation", In *Proceedings of Conference on Distributed Simulation*, San Diego, CA, Jan 1985, pp. 3-7.

AUTHORS' BIOGRAPHIES

ARTURO I CONCEPCION received the B.S. degree in Mechanical Engineering from the University of Santo Tomas (Manila), in 1969, the M.S. degree in Computer Science from Washington State University (Pullman) in 1981 and the Ph.D. degree in Computer Science from Wayne State University (Detroit) in 1984. Since 1982 he has been involved with research, funded by the National Science Foundation and Michigan State University's College of Engineering, on the theory, design and implementation of distributed simulation. He is currently an Assistant Professor in the Department of Computer Science at Michigan State University. He is now involved in a research group which studies the distributed control, efficiency and reliability of distributed computer systems. His principal interests are in distributed operating systems, networks, distributed databases, and modeling and simulation. He is a member of the ACM, IEEE-Computer Society and Sigma Xi. (He can be contacted through Stephen J. Schon at the address listed below)

STEPHEN J. SCHON received the Honors Scholar B.S. degree in Computer Science from the University of Michigan-Flint in 1983, the M.S. degree in Computer Science from Michigan State University in 1985 and he is currently working on the Ph.D. degree in Computer Science from Michigan State University. Also, he is president of Burbur LTD., a company that specializes in computer solutions to business problems. His principal interests are in artificial intelligence, knowledge-based and expert systems, distributed systems, and modeling and silmulation.

Stephen J. Schon
Department of Computer Science
Michigan State University
East Lansing, MI 48824