Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

TOWARDS A COMPUTER AIDED SIMULATION MODEL ENGINEERING (CASME) ENVIRONMENT

Joel J. Luna

Dynamics Research Corporation
60 Frontage Road
Andover, Massachusetts 01810, U.S.A.

ABSTRACT

There is increasing focus on the model development aspect of systems simulation, termed "Model Engineering". Current software development tools consist primarily of CASE tools with simulation extensions or simulation tools with model-building extensions. The need for a Computer-Aided Simulation Model Engineering (CASME) environment is identified and discussed. In particular, a CASME environment should support a model building process based on the interconnection of components which are either primitives or collections of primitives (model modules) to form other model modules. The features and tools comprising a CASME environment which support such a process and provide model specifications for use in simulation execution environments are also identified and discussed. The conclusion is that such a tool would be useful, but only insofar as simulation environments provide for implementation of model and measurement (experimental frame) specifications.

1. INTRODUCTION

To simulate a system requires a model of the structure and behavior of the system, whether it is a detailed model intended to capture as much of the actual structure and behavior as possible, or a reasonably simplified model capturing what is proposed to be the most pertinent structure and behavior. Most systems, and thus system models, can be decomposed into separable parts as part of a natural mental process in understanding and defining the system structure and behavior. This way of thinking yields approaches to system modeling that identify the system parts and their interactions in succeeding layers of detail. This approach of decomposition can be clearly seen in formalized approaches to system modeling such as Structured Analysis and Design Techniques (Marca & McGowan 1988), object-oriented modeling and design (Booch 1991, Rumbaugh, et al 1991), and in software engineering based on the concepts of modularity (Wirth 1985). In fact, it is only natural to apply software engineering techniques to simulation model development, since simulation models are ultimately software products. In order to apply those techniques to simulation modeling, however, software tools (similar to CASE tools for software engineering but oriented to simulation modeling) are needed to facilitate the decomposition and "re"composition of system simulation models. Often these tools are CASE tools with simulation extensions (e.g., Cadre) or are modeling extensions of simulation environments (e.g., SLAMSYSTEM, GPSS/VI, SIMAN IV, and SES/Workbench just to name a few).

If the simulation model development tool is separate and distinct from the execution tool, then the model can potentially be executed on different tools and in different paradigms (e.g., levels and flows of a Systems Dynamics approach versus queues, customers, and servers in a discrete-event queueing approach). Experience, however, dictates that model development is a highly iterative process between model building and execution. As a result, the developer wants to instantaneously transition between the two, which casts a strong vote for both model development and execution residing in the same tool. In any case, the point can still be made that model development and execution, though highly interrelated, are two separate efforts. Issues for model development, such as decomposition, system structure, and system behavior, are separate from issues for execution, such as timing mechanisms, collection of measures (frequently statistical), random number generation, and output analysis. The focus of this paper is on issues of the former, simulation model development, and the division between it and executable model implementation.

The emphasis on model development has been growing in conferences and literature, receiving its own label - modeling engineering - first coined by Fishwick (1989). Model engineering, analogous to software engineering, refers to the methodology and tools for

building system models which are less focused on a particular simulation method and more focused on the underlying and unifying framework. One naturally would ask: "What software tools would support model engineering ? What would be the features and tools of a Computer Aided Simulation Model Engineering (CASME) environment? What would enhance the process of modeling?" In answer to these questions, the following approach is proposed: *a CASME tool will provide for the "code-free" construction of model modules (which are collections of primitives) from model primitives and other model modules already defined.* Both model modules and primitives will have defined interfaces which are used to interconnect them to form other model modules. The rest of the paper will first define model primitives and modules, describe the model building process with an emphasis on reuse, and then identify CASME environment features and tools which address different aspects of model building.

2. MODEL PRIMITIVES

The essential starting point of the model building process is with a set of model primitives which correspond to pre-coded implementation-specific software modules. The primitives are used much like higher-level language constructs in software programming to build model modules. Many available commercial simulation environments utilize a library of primitives, such as in environments based on SIMAN, GPSS and SLAM. These approaches are based on transaction flow which defines the processes a series of transactions experience. The interfaces between primitives in these environments are predefined; the user specifies the attributes of the transactions and the flows through which they go. While transaction flow modeling environments are both useful and popular, a CASME tool should adopt a more general view to encompass a wider range of models. A more general approach is taken in SmartSim (Ulgen & Thomasma 1990), in which the interfaces (in terms of messages) between simulation model objects can also be defined. DeNet (Livny 1991) also provides a more general approach, in which connections are defined between precoded modules. Each connection can represent any number of events between modules (events in DeNet are analogous to messages in object-oriented languages). While in DeNet the ports are implemented in code , in a different, object-oriented approach (Luna 1992) the ports are notional, in that they define the set of messages which can be sent to and from an object. In effect, ports are sets of protocol which a sender or receiver object must satisfy in order to make a valid connection with another object. In order for the model primitives to be used for model building, they should identify the corresponding implemented code's interface both in terms of the types of messages (or function calls) and arguments. It is important to note that only the interface specification is needed for model building (such as the definition module in Modula-2); while the implementation is only needed for execution.

2.1 Model Primitive Interface

The interface specification of a simulation model primitive is comprised of external events (which are analogous to messages) which are exchanged with other primitives. These events are separated into input and output events, in which external input events are received and external output events are sent. In DEVS (Zeigler 1984, Concepcion & Zeigler 1988), which is based on a system theoretic approach, these external events correspond to the DEVS input and output ports. In OMT (Rumbaugh et al 1991), an object-oriented modeling methodology, events correspond to messages received and sent by objects. An example of a DEVS implementation is found in DeNet, which provides the event calling mechanism by sending an event to an output port which in turn causes the input port of the receiving module to execute the corresponding event. An example of an object-oriented approach is seen in SmartSim and other approaches (Luna 1991) utilizing the object-oriented language Smalltalk (Goldberg & Robson 1983), in which an event sent by one object as a message causes the receiving object to implement a method corresponding to the message. Zeigler (1990) discusses the similarities and differences of the DEVS and object-oriented approaches.

2.2 Declarative Model of Primitives

While the interface is all that is needed for interconnection of primitives, it does not provide any information on the behavior of the primitive. For some primitives, the name and interface provide enough indication for the user to guess its behavior. If there is documentation, the user can read a definition of the behavior of the primitive. If the user has access to the primitive's source code, the user can also try to decipher its behavior from the listing. But none of these approaches is particularly satisfying since primitives can exhibit complex behavior which is not obvious from its interface. In addition, documentation on primitive behavior can be inadequate or lag the current state of the software, and source listings of primitives (if they can be obtained) are often difficult to comprehend. The issue of understanding the behavior of implemented software objects is particularly germane to software reuse (Stacy et al 1992). Object-oriented modeling methodology addresses this issue through the use of state/transition diagrams, which are declarative models of the behavior of objects. While such an approach has merit, several improvements must be made. First, a state/transition diagram approach must be able to handle the complexity of large systems

(fairly complex primitives should not result in unusable state/transition diagrams). Second, the diagram should consistently handle events (external and internal) which effect transitions between states. Third, the diagram should be kept consistent with the actual implemented code it represents.

Reducing state/transition diagram complexity through the use of formal notation and state orthogonality has been introduced by Harel (1987). In particular, his approach using "statecharts" provides for hierarchical state diagrams so that a state/transition diagram can be divided into levels in order to be more manageable (state substitution). An alternative approach is with Augmented Transition Networks (Bole 1983), which provides for hierarchy by substituting lower level diagrams for a transition in a higher level (transition substitution). In any case, these approaches are a start in addressing the issue of dealing with state/transition diagram complexity.

The purpose of the state/transition diagram is to represent the behavior of the primitive, and as such it must specifically identify the external input and output and internal transitions corresponding to external and internal events. The time advance must also be represented for those events which occur after a specified period in a particular state (such as a delay). It is important that processes which occur over time be identified as states with corresponding transitions such as "start process" and "end process". An example of this is provided by Fishwick's FSA for a jug (1992a), in which to fill a jug, the jug transitions from the "empty" state by either transition "fill" or "transfer" to an intermediate state, "filling", and then when it is complete, to state "full". The elements of external and internal transitions with time advance are all organized with a strong theoretical basis in the DEVS methodology (Zeigler 1984). The DEVS formalism would prove the ideal basis for developing a state/transition diagram approach which relates the type of event, condition and delay to transitions between states.

In order for a state/transition representation of a model primitive's behavior to be valuable in understanding and correctly implementing the model primitive, it must be consistent with the model primitive code. If the model primitive is in a state of change due to further development and modification, then the state/transition diagram must be continually updated. There are two possible approaches: update the state/transition diagram manually from the model primitive implementation, or automatically generate the diagram each time the model primitive is changed. The latter seems most desirable, although there do not seem to be any such tools currently in existence. If the needed information is contained in the primitive source code (such as one based on a DEVS or object-oriented approach), then it seems plausible that a state/transition diagram could be automatically generated. An added wrinkle in an object-oriented approach is with inheritance - can an object's state/transition diagram also be defined incrementally? This issue is raised by Aksit and Bergmans (1992). It seems that using a hierarchical approach could be promising, but this requires further thought and work.

In summary, a model primitive defines the interface of a corresponding software implementation in terms of ports or connections which each have a specified protocol of events or messages. The model primitive's behavior is represented by a state/transition diagram tailored to simulation modeling. A group of primitives together form a library for use in the model building process of a CASME tool.

3. MODEL MODULES

Model modules are a set of interconnected components in which each connection is defined by a set of protocol and each component is either another model module or a model primitive. The model module external interface is the set of unconnected interfaces of the module's components. Model modules are analogous to the "coupled component" in DEVS (Zeigler 1984), while the model primitive is analogous to the "atomic component".

3.1 Module Levels of Abstraction

Model modules can be defined in successive layers of abstraction (Figure 1). At the highest level, the module *topology* view is defined by a digraph composed of nodes interconnected by unidirectional arcs. The next level, the *protocol* view, identifies the types of internode communication. Each arc has a corresponding protocol of events or messages identified for it. As a result, a node's interface is defined by the total protocol of its input and output connections. Since the node's behavior itself is not identified, any number of nodes which satisfy the interface can be "plugged" in. This essentially results in a module whose component types are parameters which can be varied. Such an approach based on processes is implemented in AI3 (Benjamin, Mayer & Blinn 1992), and is also implemented using substitution transitions in a Petri-net implementation in Design/CPN (Meta Software 1992). When each node is labeled as an object, the *protocol* view corresponds to the object model in OMT. The next level, the *component* view, identifies the node types as either model primitives or model modules. At this level the components have been parameterized. The result of a hierarchy of modules which are realized at the *component* view is a hierarchical tree of interconnected modules whose 'leaves' correspond to actual software code and whose internal nodes are connection specifications (Figure 2). At the lowest level, the *data* view, the variables of each primitive are assigned with a particular set of values (i.e. from a data base or

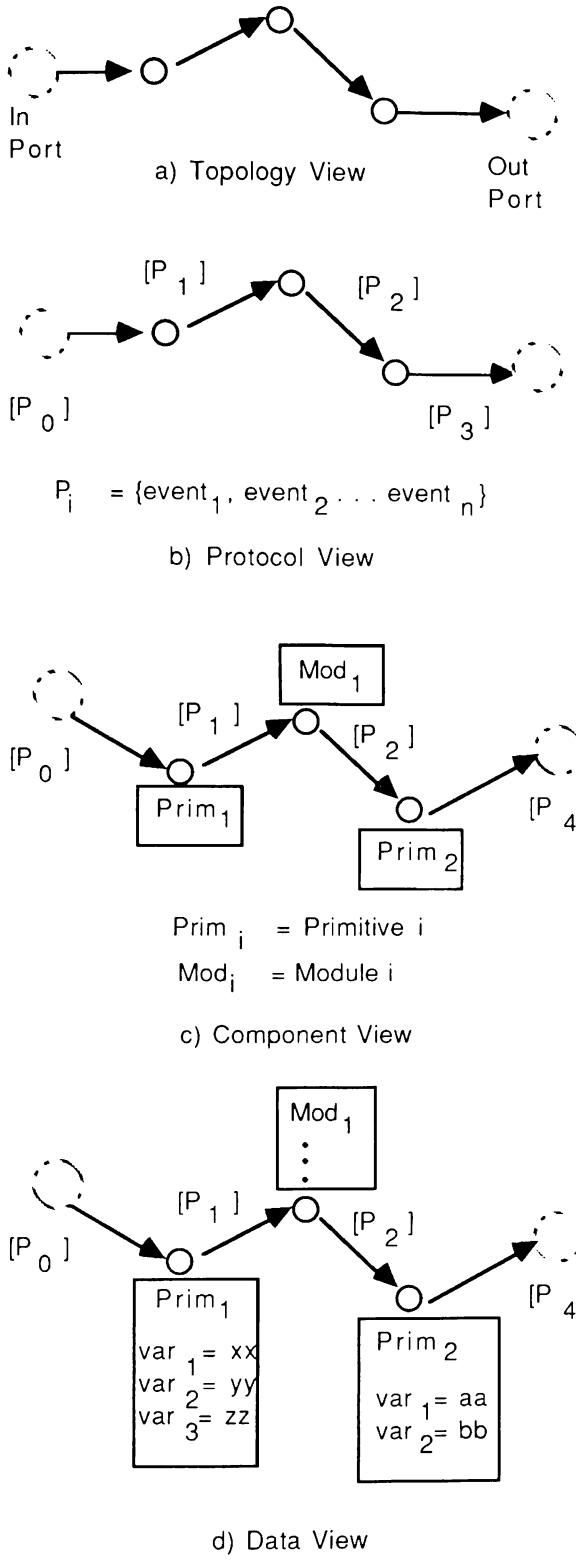

Figure 1. Model Module Views by Abstraction Level

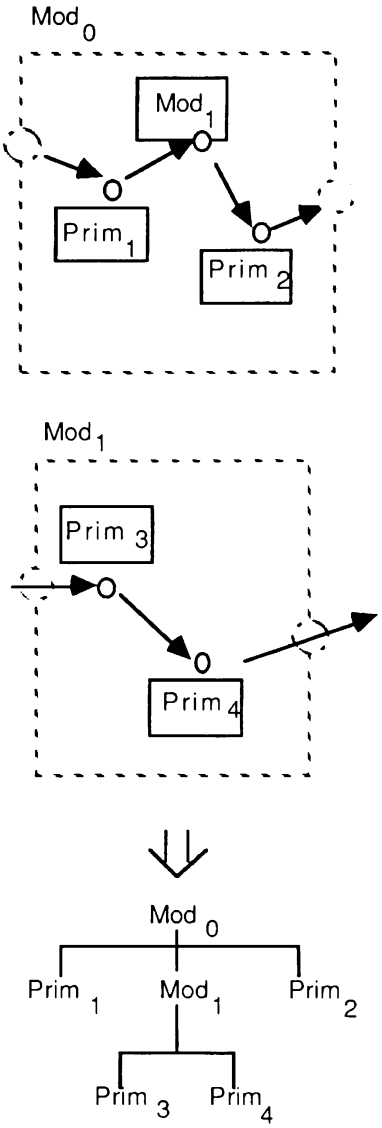

Figure 2. Hierarchy of Modules at Component View

parameter file). The module at this level corresponds to an object instance and provides a full definition of the module for execution. The different levels of abstraction for a model module are important when considering issues of model repositories and model reuse.

3.2 Module Declarative Models

Model modules as described thus far are functional rather than declarative models; they identify the different components and their relationships. The behavior of the module is essentially the aggregate of the interactive behavior of its components. Is a declarative model appropriate for the module? It seems the answer is yes

if the declarative models of the components can be abstracted to form a single overall declarative model which provides some overview of the module behavior, perhaps using hierarchical state/transition techniques as discussed earlier. Whether this actually provides more insight, rather than just a jumble of states or a small but nonintuitive set of states, remains to be seen. But what if the user desires for the module to have a behavior of its own? The answer is simple, another primitive needs to be added to the primitive library (with corresponding software code) which implements the module-specific behavior (beyond the components' aggregate behavior). This 'meta' primitive corresponds to the immutable composition of an object as described by Odell (1992).

4. "CODE-FREE" MODEL BUILDING

Model building is the process of building modules from model primitives and other modules (Figure 3). "Code-free" refers to the pluggable approach this allows by utilizing pre-coded software objects. In a typical session in which a user builds a model "bottom-up", a selection of primitives and pre-defined modules is made available to the user via a menu or a palette of icons. The user defines the different components by selecting one of the primitives or modules. The user then connects the components (either via menu-prompt or by graphically drawing an arc), which establishes the protocol of the connection based on the protocol of the interface ports selected. If the port protocols are incompatible, then the connection is invalid and thus disallowed. This ensures compatibility at run-time. Once all the components are connected, the user specifies the input and output ports of the module interface and connects the remaining component ports to them. The resultant module can then be used as is or reused in a higher level module.

As an alternative, the user can adopt a "top-down" approach, in which the user defines a module consisting of primitives. As the behavior of each component becomes better defined, then a corresponding module can be created and substituted for the primitives in the higher level module. As a result, the model becomes increasingly decomposed into greater depth.

4.1 Reusability Issues

This approach to model building involves a heavy reliance on the adequacy of the library primitives to meet the user's needs, analogous to the language constructs the software programmer has available to meet his or her programming needs. While many simulation environments (i.e. those based on transaction flow) have a pre-defined set of primitives, these are usually kept in inaccessible libraries. In addition, much fewer allow for the capability to combine primitives into subsystems which themselves can be reused, which does exist in such tools as Arena (Pegden & Davis 1992), SES/Workbench, and RESQME (Gordon et al 1993). SmartSim allows for more flexible primitives and subsystems, although subsystems are themselves objects requiring some programming. A CASME user should be allowed to program when necessary to build new primitives, and be allowed to save modules which are built as reusable model components. This is analogous to the X-Window approach, which separates the widget programmers from the application programmers (Young 1989).

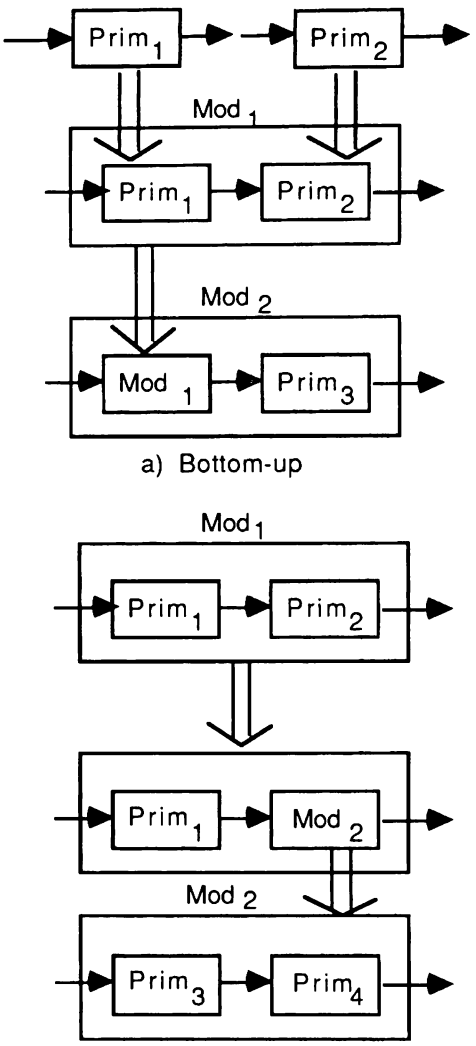

Figure 3. Model Building Using Primitives and Modules

4.2 Incremental Model Building

Model building by modifying existing modules raises the issue of managing the variation of modules, in

particular, an incremental programming approach to building modules. Just as inheritance in object-oriented programming allows subclassing by defining only the changes to the superclass, so new modules ought to be able to be defined as variations to an existing module. For example, if a change in component type is considered, only the new component type, protocol, and connection need to be saved as changes to the existing model module. The variations of a given module can then be displayed to the user hierarchically (similar to the Smalltalk class hierarchy browser) rather than as a flat collection of modules from which the user has no indication as to how they are related.

A question remains as to whether the user should be able to view a module variation as a variation or as a fully reconstructed module with the variations included. In object-oriented code, typically only the variations are displayed to the programmer for each object class. For large and complex modules, displaying only the variations would be valuable. If the changed components, arcs, and protocol are displayed with the corresponding connected components, then only the pertinent parts of the module would be viewed by the user. Sometimes, however, having the whole picture is also valuable, so that a user would want the option of either an incremental or full display of the module (Figure 4).

5. CASME ENVIRONMENT FEATURES AND TOOLS

There are several features and tools which a CASME environment should provide which were highlighted from previous discussion: a model builder tool, repositories for primitives and modules, and an executable builder tool.

5.1 Model Builder Tool

The model builder tool consists of a user interface, access to the repositories, and a means for checking connection protocol compatibility. The user interface can be menu-prompt or graphical or some combination, and must provide a means for the user to create, select, delete, and modify model modules. The user should also be able to view declarative models of primitives and modules. Browsers which display the primitives and modules according to hierarchical relationships would greatly simplify the user's selection of them. In order to support this, the model builder tool should provide saving and retrieving facilities for access to the primitive and module repositories. The user should also be able to create or delete connections between components. This involves another tool, a protocol checker, which compares the events or messages originating from the sender with the events or messages which the receiver expects to get. This could be further enhanced by a protocol suggester, which given a component identified as either a source or destination component, will find other components (primitives and modules) which satisfy the protocol constraints.

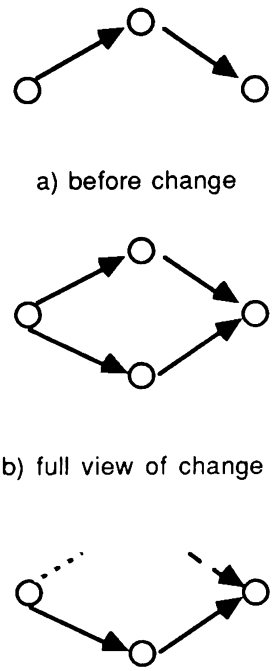

Figure 4. Display of Incremental Changes

5.2 Repositories

The repositories for primitives and modules should hold both model and data information. The primitive repository holds the interface specification and associated state/transition diagram (corresponding declarative model). The primitives should be hierarchically related by class as defined using object-oriented analysis and design techniques whether or not they are implemented in an object-oriented language. This allows for the user to focus on fewer selections at a time; it is easier to pick 1 out of 10 twice rather than pick 1 out of 100 once. Data sets for each primitive should also be saved in a corresponding database. Module repositories, however, can be more complex. Abstraction, compositional, and incremental (specialization) hierarchical relations all exist for each module (Luna 1993). Abstraction hierarchical relations relate the module abstraction levels (*topology*, *protocol*, *component*, and *data* views). A single trace down the abstraction hierarchy fully defines a model module (Figure 5). Compositional hierarchical relations relate the module to its components, which constitutes a hierarchical tree in the *component* view. Incremental

(specialization) relations relate a module to the 'base' module from which it is a variation. Each of these relations should be displayable to the user. In addition, the repositories should allow for import and export in some kind of standard form (e.g. flat ASCII).

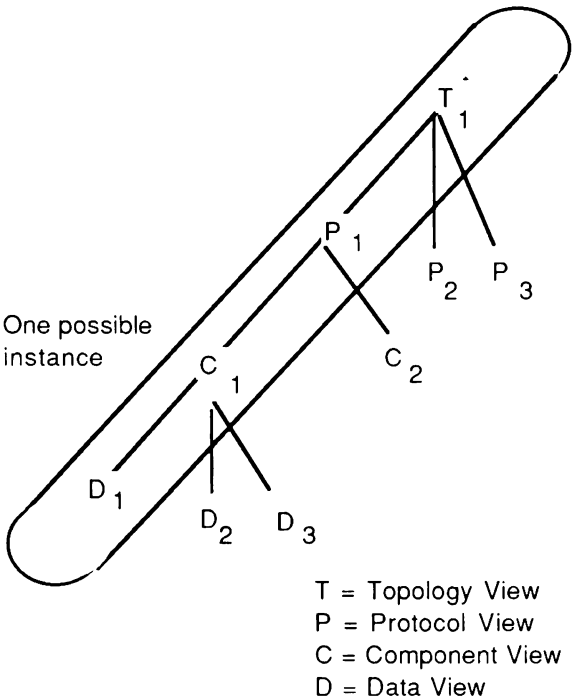

Figure 5. Fully Defined Module Hierarchy

5.3 Executable Builder Tool

Lastly, the executable builder tool would access the primitive and module repositories to generate simulation model specifications for existing simulation execution environments. In DeNet, for example, the topology file is the simulation model specification. The DeNet environment uses this file to build and execute the simulation model. DYMOLA (Cellier 1991) similarly generates model specification files to run in ACSL, a continuous simulation environment. SmartSim similarly can generate executable SIMAN code.

So far, little attention has been paid to the specification of measures. This is an important consideration, since measures are typically included in model code and can significantly complicate it. Measures should be defined externally to the model code itself. The executable builder tool should also provide facilities for defining measures based on component events, similar to the approach described by Luna for hierarchical modular modeling (1992). These measures should then be defined in a simulation environment file which can be used by simulation execution environments. The only problem is that many environments rely on the reporting facilities for measures to be directly implemented in the modules as described, so that modifying the measures taken requires modifying model primitive code. This is unsatisfactory. Approaches similar to that of SIMAN, which utilize Zeigler's experimental frame based on separation of model behavior and measurement (Zeigler 1984), should be encouraged. This would allow CASME environments to specify measures as well to be implemented by the target simulation environment.

6. CONCLUSION

A concluding question well might be, "Is there a need for a CASME environment? Why not use a CASE tool instead?" The answer lies in how well existing tools provide the capabilities needed for model engineering as defined in this paper and elsewhere (Fishwick 1992a). Clearly parts of the capabilities are implemented in currently available tools, but others are still lacking. It is believed that a hybrid approach, utilizing object-oriented and system theoretic methodologies provides the greatest promise. Ultimately, however, a CASME environment will be ineffective if it is developed without regard to simulation executable environments. A good start would be to develop an open simulation executable environment as the target for a CASME, based on open libraries such as SIMPACK (Fishwick, 1992b). An open CASME approach much like that of UNIX or X Windows would enable contributions and enhancements to occur over time. It seems that it may be time for industry standards regarding model and measurement specification interfaces between a CASME environment and executable environments. With such standards, it would be possible to develop models for implementation in a variety of environments utilizing a variety of paradigms. This would also make it much more likely that independently developed primitives and modules could be reused.

REFERENCES

Aksit, M., and Bergmans, L. Obstacles in Object-Oriented Software Development. In Proceedings of OOPSLA '92, 341-358.

Benjamin, P.C., Mayer, R.J., and Blinn, T.M. 1992. Intelligent Support for Simulation Model Design from IDEF3 Descriptions. College Station, TX: Knowledge Based Systems Inc.

Bole, L. (Ed.) 1983. The Design of Interpreters, Compilers, and Editors for Augmented Transition Networks. Berlin: Springer-Verlag.

Booch, G. 1991. Object-Oriented Design with Applications.

Cellier. F.E. 1991. Continuous System Modeling. Borlin Springer-Verlag.

Concepcion, A.I., and Zeigler, B.P. 1988. DEVS

Formalism: A Framework for Hierarchical Model Development. IEEE Transactions on Software Engineering, 14 (2), 228-241.

Fishwick, P.A. 1989. Qualitative Methodology in Simulation Model Engineering. Simulation Journal 52, 3 (March), 95-101.

Fishwick, P.A. 1992a. An Integrated Approach to System Modeling Using a Synthesis of Artificial Intelligence, Software Engineering and Simulation Methodologies. ACM Transactions on Modeling and Computer Simulation, 2(4).

Fishwick, P.A. 1992b. SIMPACK: Getting Started with Simulation Programming in C and C++. In Proceedings of the 1992 Winter Simulation Conference, 145-162.

Goldberg, A. and D. Robson, 1983. Smalltalk-80: The Language and Its Implementation. Reading, MA: Addison-Wesley.

Gordon, R.F., Loewner, P.G., MacNair, E.A., and Chang, K.C. 1993. The RESearch Queueing Package Modeling Environment (RESQME) Version 4.6 Guide for OS/2 and RS/6000. Yorktown Heights, NY: IBM Research Division.

Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8, 231-274.

Livny, M. 1991. DeNet: An Overview. Computer Sciences Department, University of Wisconsin - Madison.

Luna, J. 1991. Application of Hierarchical Modeling Concepts to a Multi-Analysis Environment. In the Proceedings of the 1991 Winter Simulation Conference 1165-1172.

Luna, J. 1992. Hierarchical, Modular Concepts Applied to an Object-Oriented Simulation Model Development Environment. In Proceedings of the 1992 Winter Simulation Conference, 694-699.

Luna, J. 1993. Hierarchical Relations in Simulation Models. In the Proceedings of the 1993 Winter Simulation Conference. To be presented.

Marca, D.A., and McGowan, C.L. 1988. SADT. New York: McGraw-Hill.

Meta Software Corp. 1992. Design/CPN Version 1.9: A Reference Manual. Cambridge, MA: Meta Software Corp.

Odell, J. 1992. Managing Object Complexity, Part II: Composition. Journal of Object-Oriented Programming, 5(6), 17-20.

Pegden, C.D., and Davis, D.a. 1992. Arena: A SIMAN/Cinema-Based Hierarchical Modeling System. In Proceedings of the 1992 Winter Simulation Conference, 390-399.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W. 1991. Object-Oriented Modeling and Design. Englewood Cliffs, NJ: Prentice-Hall.

Stacy, W., Helm, R., Kaiser, G., and Meyer, B. 1992. Ensuring Semantic Integrity of Reusable Objects. In Proceedings of OOPSLA '92, 298-302.

Ulgen, O.M. and Thomasma, T. 1990. SmartSim: An Object-Oriented Simulation Program Generator for Manufacturing Systems. International Journal of Production Research, 28 (9), 1713-1730.

Wirth, N. 1985. Programming in Modula-2. 3d ed. New York: Springer-Verlag.

Young, D.A. 1989. XWindow Systems Programming and Applications with Xt. Englewood Cliffs, NJ: Prentice-Hall.

Zeigler, B.P. 1984. Multifaceted Modelling and Discrete Event Simulation. Orlando FL: Academic Press Inc.

Zeigler, B.P. 1990. Object-Oriented simulation with Hierarchical, Modular Models. Orlando, FL: Academic Press.

AUTHOR BIOGRAPHY

JOEL J. LUNA is a Senior Analyst at Dynamics Research Corporation, He is primarily involved in application of systems analysis techniques, especially modeling and simulation, to a variety of projects. His current interests are in the areas of simulation, modeling, and object-oriented programming.