

ENGINEERING SOFTWARE

January 23, 2010

by: W.C. Cave, R.E. Wassmer, K.T. Irvine, & E.S. Slatt

INTRODUCTION

This paper is motivated by the following:

- Software is becoming the most important industry in the world;
- Software is one of the very few industries where productivity is actually declining;
- A new technology is available that dramatically increases software productivity.

It does not take much investigation to confirm the growing importance of the software industry. Despite huge investments, the industry lags far behind demand. The fact that productivity in this industry is declining is also well documented, [AM], [AN1], [AN2], [CA1], [GL], and [GR]. This has been occurring in spite of the fact that project sizes have been decreasing over the past ten to fifteen years because of management concerns about risk of failure, [SG1], [SG2]. This is also in contrast to the computer chip industry, the world leader in productivity increases, resulting from its sophisticated Computer-Aided Design (CAD) tools and disciplined engineering approach.

This paper provides an overview of a new technology that brings an engineering discipline to software and supports a CAD approach. We start with measures of software productivity. We then describe how a new technology has evolved from CAD engineering (see Appendix A) for dramatically improving software productivity. This technology had evolved originally from a requirement for fast simulations to run on parallel processors, so it naturally provides improved productivity for the application of parallel processing as well.

IMPROVING PRODUCTIVITY

The paper by Anselmo and Ledgard, [AN1], describes the properties required of a software product that enhance the ability to produce it, and more importantly support future upgrades. The two key properties - *independence* and *understandability* - help to ensure reusability and scalability. These properties must be ensured by the tools used to build that software. We agree that these key properties must be embedded in the software, and that the tools used to build and support a software product must ensure that these properties exist.

Our objective is to minimize the time to build and support a software product while meeting constraints on functional requirements, quality, run-time speed, accuracy of results, and budgets. We leave selection of these constraints to marketing, user groups, and top management. Our goal is to minimize the time to create a new module or modify an existing module. This is the software technology issue, the subject of our focus.

DEALING WITH INCREASING COMPLEXITY

Recent papers describe the need for an engineering approach to overcome the barriers in dealing with increased software complexity, see [AM], [BR], [GL], [GR], and [PO]. To accomplish this goal, we must draw on engineering principles that can be used to build software. The principles suggested in these papers call for techniques such as CAD tools for software, and an approach to modeling the architecture of software systems similar to that used in hardware design. Such a technology already exists, being developed and refined since 1982.

This technology was developed as the General Simulation System (GSS), used for building complex engineering design and planning tools that require embedded discrete event simulations. This technology is based upon engineering principles and CAD tools used to produce complex electronic circuit designs, reference Appendix A. The remainder of this paper provides an overview of the underlying principles, implementation, and use of that approach. It encompasses a new concept - *software architecture*, and the use of engineering drawings to describe that architecture. This architecture has no relation to flow charts or other symbolic approaches that merely represent code. This technology provides for the design of modular system architectures using measures of independence and understandability.

Automating The Representation Process

In the early 1960's, electronic circuit designers developed automated tools for solving complex systems of nonlinear differential equations representing digital waveforms in the time domain. These CAD tools allowed engineers to describe large networks topologically using engineering drawings. Programming skills were no longer required to build large simulations. This afforded a huge leap in design productivity, leading to integrated circuit design.

For large networks, the number of state variables may be in the thousands. Solving worst case design problems involves multiple optimization runs that may require thousands of simulations each. Each simulation may involve thousands of nonlinear differential equations. Speed and accuracy are the driving forces in designing these simulations. If it takes a computer days to do a design, only one or two test points are produced in a week - not very productive.

Capitalizing Upon General Principles

The *state space* framework has been shown to be the most general mathematical representation of a dynamic system, see [GE], [SC], and [ZA]. Providing a framework for describing problems is not the only benefit of state space. It also affords general rules for discovery of much faster solutions to problems that could otherwise run for days on a computer.

State space is used by CAD system developers because it provides the most convenient framework for solving any type of dynamic problem in the time domain. The general form of solution holds for any set of independent state variables. This allows for the development of generalized methods, e.g., optimal sparse matrix inversion and describing functions to solve nonlinear problems fast while ensuring algorithm convergence. The end result is to solve huge problems N times faster than conventional approaches, where N is the size of the transformation matrix. However, this approach requires formulating problems in a mathematical framework.

As models become very nonlinear and therefore complex, it is difficult to describe their dynamics in a strictly numeric framework. For many models, restriction to conventional mathematical representations is awkward, independent of size. In these cases, mathematical abstractions lead to difficulties in understanding how a model relates to the real system. From an economic standpoint, the use of distributed sets of generalized *data* vectors, [CA4], is superior in most simulation (and software) applications. This is because generalized data vectors can contain natural language descriptions of the state of a model, rendering the transformations much easier to understand by a subject area expert with little or no programming experience.

The General Simulation System (GSS), is a discrete event simulation environment, see [GO] and [GSS]. However, the underlying structure of GSS is different, being based upon a *generalized state vector* that defines the state of the simulation at any instant of time. Based on this concept, GSS models move from state to state as transformations are invoked in a simulation. The generalized state vector and corresponding transformations are explained in the following sections.

A MORE GENERALIZED PROBLEM FORMULATION

The motivation to extend the concept of mathematical modeling into discrete event simulation stemmed from the requirement to insert nonlinear decision algorithms within sets of differential equations. In the late 1970s, modelers wanted to break up systems of equations and embed natural language conditions and rules, e.g.,

```
IF MESSAGE_TYPE IS CONTROL, THEN ... ,  
ELSE IF MESSAGE_TYPE IS DATA, THEN ... .
```

In discrete event simulation, state transitions are determined by the modeler in terms of scheduled events. Thus it was not apparent how to derive a mathematical framework to support this new requirement. In 1982, further study led to the development of a complete and consistent state space definition of GSS. An overview of this is provided in the Section below entitled *Concept Of A Generalized State Vector* and in Appendix A - *State Space Definition Of A GSS Model*. A comparison of mathematical and rule oriented formulations is provided in *Simulation Of Complex Systems*, [CA4].

Achieving Run-Time Speed - *The Property Of Independence*

Because of the excessive running times of some critical simulations (5 to 7 days to run a 2 hour scenario), the most important GSS design requirement was to ensure that simulations could run on a parallel machine, [PS]. The need for parallel processing imposed the requirement that two or more processes (sets of instructions) must be able to run concurrently on separate processors. This implies that concurrent processes must be *independent*. The *property of independence* implies that those processes share no data. This led to the decision to separate data from instructions - *the separation principle* - so that the independence property could be tracked. The GSS design called for a connectivity matrix to determine what processes shared what data. Then when allocating processes to processors, the connectivity matrix is used to determine if a process could run concurrently with those already running.

Benefits Of Separating Data From Instructions

The separation of data from instructions provides significant benefits. First, it allows one to capitalize directly on the property of independence. By limiting access to specified data structures, models can be *designed* to be independent. This leads to a decomposition of the simulation or system database into hierarchical data structures - defined as *resources* in GSS. Instructions are grouped into hierarchical sets of rules - defined as *processes*. Resources and processes are grouped into elementary models. Elementary models are grouped into hierarchical models. This is illustrated in Figure 2, which contains a model of a local telephone system with PBXs connected to a local switch. The resources (data) are contained in the ovals, and the processes (instructions) are contained in the small rectangles. These can be edited directly on a drawing as shown in the boxes designated by the arrows.

In GSS, the interconnection of processes and resources is done graphically using icons and lines. This provides the ability to produce an engineering drawing of the *architecture* of a model, where lines connecting processes to resources determine what processes have access to which resources. Models (blue boxes) may be connected to each other by connecting a process in one model to a resource in another. *Independence of models can be inspected, visually, simply by looking at the number of lines connecting them.*

For parallel processing, one needs a map to determine if two processes are independent and therefore can run concurrently. This is accomplished using the Connectivity Matrix shown in Figure 3. For each resource, one can see the processes that are connected to it (denoted by an X), and therefore share data. The more sparse the matrix, the greater the independence, and the more simple the transformations, a concept taught to engineers in linear system theory.

The connectivity matrix in Figure 3 has additional information. Where there is an I, the resource is shared during initialization only, not during run time. The P denotes that the resource is shared when outputting performance data. The blue boxes indicate the model boundaries for elementary as well as hierarchical models. In this example, the hierarchical model is instanced. During run time, it only shares two resources with the rest of the system, except when producing performance data, a one-way output.

Concept Of A Generalized State Vector

Separating data from instructions clarifies the meaning of the state of a model or simulation. It is defined by the state of all resources in that model or simulation. This leads to the *generalized state vector* concept, refer to Appendix A. The state of a simulation is defined by the state vector comprised of all the resources in that simulation. A simulation is partitioned into a set of sub-states corresponding to the resources or subvectors.

This approach allows one to apply many of the concepts from the state space framework. For example, the simulation state vector in GSS is considered to represent a *generalized coordinate system*. It is up to the modeler to come up with the best set of states to make the problem easy to solve. This implies selecting the set of resources (substates) that simplify the transformations of state representing the dynamics of the system. These transformations are embodied in the processes. When a process runs, it starts with the initial state of the attached resources and takes them to the next state.

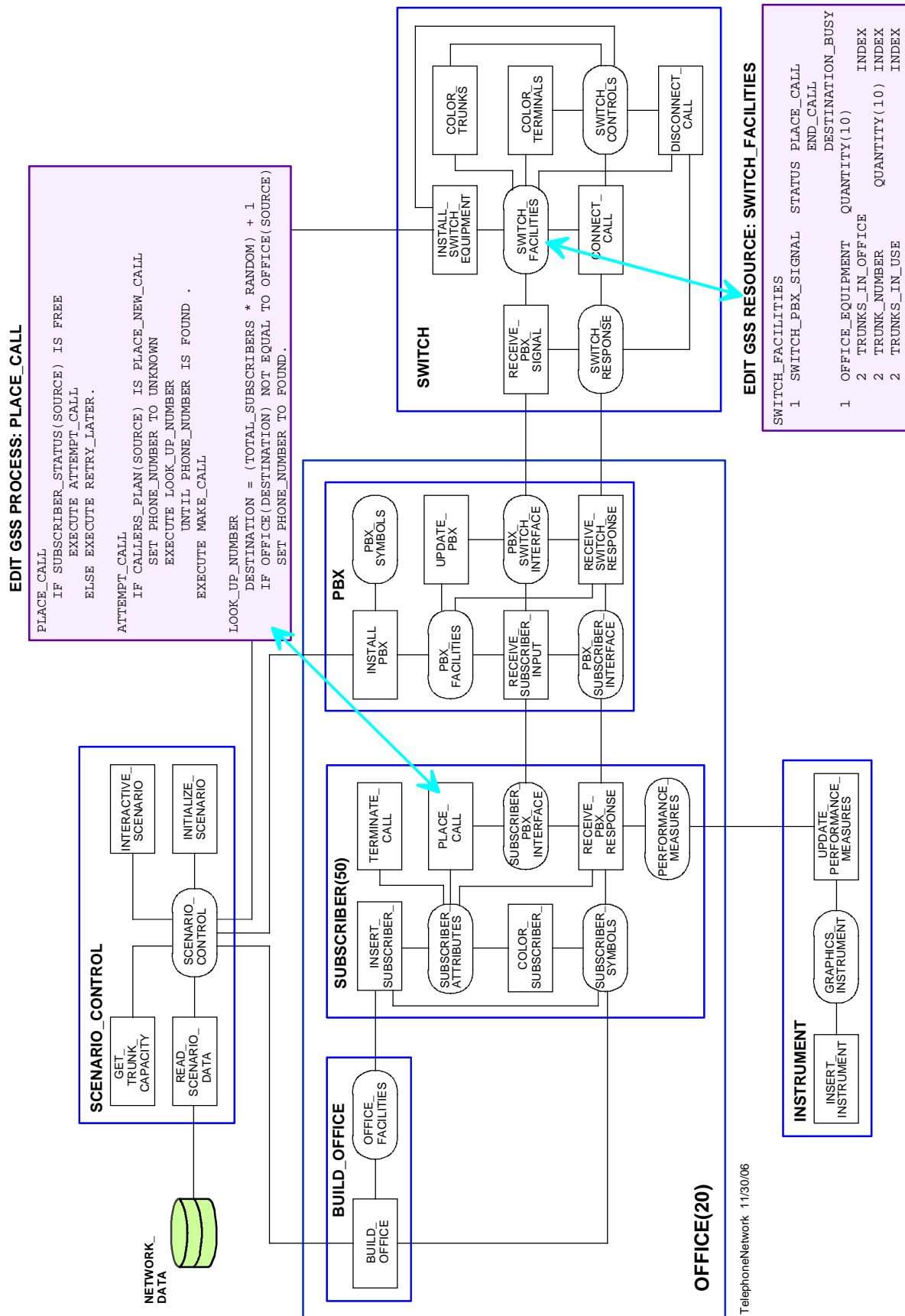


Figure 2. Telephone network models in GSS.

CONNECTIVITY_MATRIX 11/14/06

MODELS	PROCESSES	RESOURCES													
		SCENARIO_CONTROL	OFFICE_FACILITIES	SUBSCRIBER_SYMBOLS	PERFORMANCE_MEASURES	SUBSCRIBER_ATTRIBUTES	SUBSCRIBER_PBX_INTERFACE	PBX_SUBSCRIBER_INTERFACE	PBX_FACILITIES	PBX_SYMBOLS	PBX_SWITCH_INTERFACE	SWITCH_RESPONSE	SWITCH_FACILITIES	SWITCH_SYMBOLS	GRAPHICS_INSTRUMENT
READ_SCENARIO_DATA		X													
INITIALIZE_SCENARIO		X													
INTERACTIVE_SCENARIO		X													
GET_TRUNK_CAPACITY		X													
BUILD_OFFICE	I	X	X												
INSERT_SUBSCRIBER		X	X												
PLACE_CALL					X	X									
TERMINATE_CALL			X	X											
COLOR_SUBSCRIBER			X	X											
RECEIVE_PBX_RESPONSE		X	X	X	X	X									
RECEIVE_SUBSCRIBER_INPUT						X	X	X		X					
INSTALL_PBX	I							X	X						
UPDATE_PBX								X	X						
RECEIVE_SWITCH_RESPONSE							X	X		X	X				
RECEIVE_PBX_SIGNAL										X	X	X			
INSTALL_SWITCH_EQUIPMENT	I												X	X	
COLOR_TRUNKS													X	X	
COLOR_TERMINALS													X	X	
CONNECT_CALL											X	X	X		
DISCONNECT_CALL											X	X	X		
UPDATE_PERFORMANCE_MEAS					P										X
INSERT_INSTRUMENT															X

Figure 3. Resource, Process, and Model Connectivity Matrix.

This is no different from the problem of picking the best set of variables or coordinate system to simplify a set of partial differential equations. As indicated above, courses that cover problem solving in the applied sciences, e.g., physics and engineering, stress that choice of a coordinate system is key to making a problem easier to solve. In GSS, one selects the best breakout of resources (state subvectors) to simplify the processes (transformations of state).

In GSS, this concept is expanded to the generalized state vector, one that consists of general information, not just variables that take on numeric values. In particular, we consider that a GSS resource is equivalent to a data vector containing states such as RED, YELLOW, and GREEN. The data can be natural language words or character strings, as well as numbers. A generalized state vector may consist of many subvectors, i.e., GSS resources.

With the above in mind, consider that a GSS simulation consists of a very complex *simulation state vector* (the simulation's data base) that changes as processes are invoked. At any instant of time, the simulation state vector is represented by the state of all resources in the simulation.

Relation To Software

The Visual Software Environment (VSE), a general software environment, is a subset of GSS. There are only two statements that differentiate GSS from VSE. These are the SCHEDULE and CANCEL statements that pertain to scheduling and removing processes from the GSS event queue. On a parallel processing system, the SCHEDULE statement effectively starts a thread that, if independent from those that are running, can be run concurrently. Since GSS is written in VSE, anything that can be done in GSS can also be done in VSE.

ARCHITECTURAL DESIGN FOR REUSABILITY

In a production software environment, many opportunities arise to enhance a module in its original form, or reuse a module in another system. Anyone who has completed a successful project for one client should be aware of the desire to reuse as much of the prior design with new or existing clients. This typically requires changes or additions to the prior design to incorporate new requirements.

We want to highlight those factors that are major contributors to reusability when building a task of the size shown in Figure 4 or larger. When producing a large software architecture, there are design principles to be followed (as in hardware) to maximize module understandability and independence. Those described below have been developed from practical experience.

To improve productivity across the total life cycle, successful developers know that their products are in the support mode typically more than 80% of the time. Support includes adding new facilities as well as modifying a growing set of existing facilities. Not only do these facilities grow in number, but they also grow in complexity as more options are added. Like adding more rooms to a building, one must consider modifications to the architecture, else the old architecture becomes too weak or burdened to support the new facilities. Sticking with an old architecture makes adding new facilities more difficult as a system grows. Thus, good software architectures must be designed to evolve and allow generation of new reliable releases quickly. So how does one develop a good architecture?

APPROACH TO SOFTWARE ARCHITECTURE

Figure 5 depicts the system life cycle that includes a Detailed Architectural Design box (blue). Our objective is to define this process in sufficient detail to ensure the design of a good architecture.

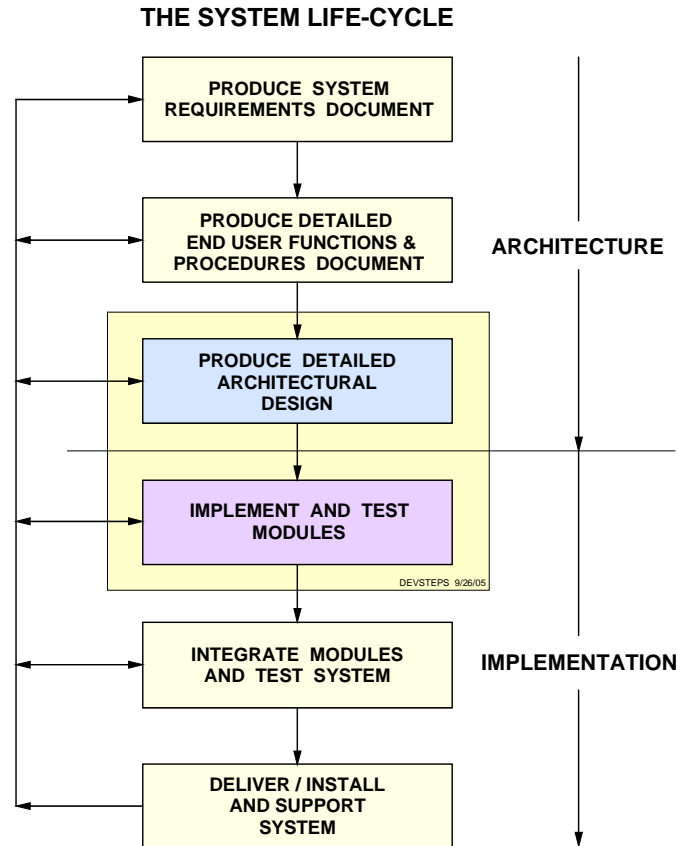


Figure 5. Role of detailed architectural design within the system life cycle.

A Framework For Describing The Architectural Design Process

To accomplish our objective, we will borrow from the mathematics of control theory, using our generalized state-space framework. We must decompose the user's functional requirements into a detailed software architecture using hierarchical modules, resources and processes from the GSS/VSE version of this framework.

This will be approached as transformations on different substate spaces as well as transformations between substate spaces. Our goal is to produce a detailed hierarchical module architecture, down to the resource and process level, such that there are minimal architectural changes during the code implementation phase. However, functional changes should be easily accommodated, including those requiring changes to the architecture.

Application Of The State-Space Framework

GSS is a discrete event simulation environment, where flow of control is dependent upon a huge set of event strings that in turn depends upon a huge state space. Although sequences of events may be deterministic, they are virtually unpredictable in a large simulation. When selecting frameworks for solving such problems, one must ensure completeness and consistency so that, depending upon their inputs, solutions converge to the expected answer unambiguously. When developing GSS, the State Space framework was used to ensure these properties.

The direct mapping of GSS resources into state subvectors and processes into state space transformations made this analogy trivial, refer to Appendix A. The only difference is that GSS state vectors contain non-numeric data in the form of character strings, and the transformations on these vectors permit other than mathematical operations. Since they are all resolved at the bit level inside a computer, one can show that these operations can be reduced to the equivalent of mathematical operators on numbers. Therefore, properties of transformations using the state space framework can be applied to those used in GSS.

The state space framework has been used to derive fast and efficient approaches to solving circuit design problems, see [HA], [HA1], and [CA2]. These approaches were implemented in CAD software packages used by engineers for simulation and design optimization of complex electrical networks. When designing this type of software - to achieve speed in the multiple simulation optimization process - one is concerned with selecting the best set of state variables for solving the problem, and grouping these state variables to minimize the complexity of the transformation.

Selection of the “best” state variables equates to choosing the best coordinate system for doing transformations. For example, if one is investigating the dynamics of motion on a sphere, problems are likely solved with much less algebra, and therefore faster, in spherical coordinates. This can be measured using the sum of the products of operations and instruction speeds.

Given the best state variables, one may further reduce operation counts by effectively diagonalizing the large matrices that implement the transformations required to solve the problem. This can be done by interchanging rows and columns of the matrix and corresponding vectors to produce submatrices that are *maximally independent*, as described in [HA1]. This is known as the optimal ordering problem.

Mapping this into the software architecture problem, selection of the best set of state variables is equivalent to choosing the attributes used to represent the states of a system so as to maximize simplicity of the resulting transformation. Optimal ordering is equivalent to grouping these states in a way that maximizes independence of the modules and further simplifies the transformations. This is illustrated in Figure 3.

To translate these rules into those for software architectures, a complex transformation may require multiple resources and processes. Selection of the states that represent the software functions, and grouping them into different resources can serve to simplify the processes used to do the transformations. Simplification may be considered from two standpoints: (1) speed of operations, and (2) module understandability and independence. The approach to circuit design achieved both. In the case of software, this is an architectural problem, requiring learned skills.

As designs become more complex, one must look to break up the architecture. This requires decisions on the use of more components (modules, resources, and processes) with smaller amounts of code in each. This represents a trade-off between architecture (graphical symbology) and language (text). An algebraic equation or a complex conditional statement is best done in a language. However, as the size of resources and processes in a model grow and span many pages of code, the architecture may be decomposed into separate modules as illustrated in Figure 4. This makes the resources and processes smaller and more understandable. Breaking up modules into submodules makes the modules more understandable. Without a visualization of the architecture that has a one-to-one mapping into the code, this cannot be discussed - let alone achieved.

Software Architecture Development Steps

We want to translate these concepts into steps for designing software architectures. Figure 6 is an expansion of Figure 5, where the software architecture development process (blue box) has been broken into 5 steps. These steps are described below.

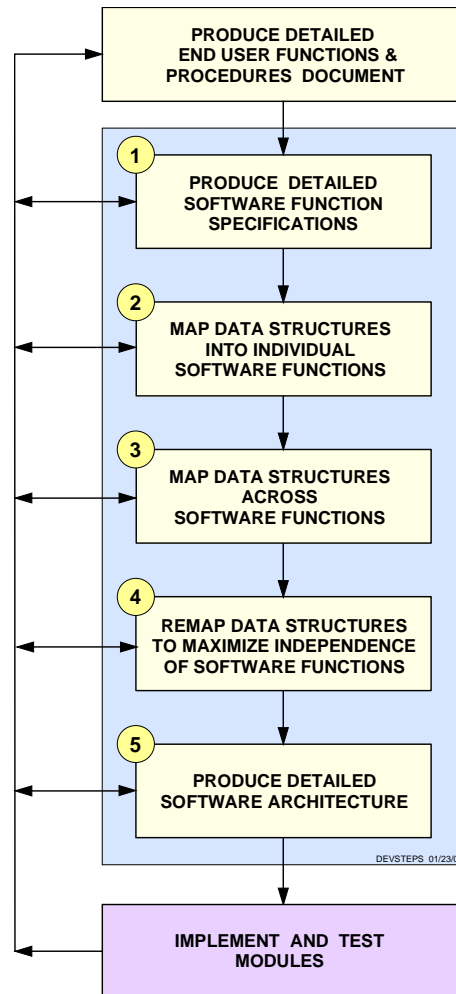


Figure 6. Breakout of the detailed architectural design process.

Step 1 - Produce Detailed Software Function Specifications

Given the end user functional requirements, determine the generic software functions that are required to perform them. These may be described as transformations of state. This implies describing the functional states of the system, and the transformations on those states. It is important to note that when dealing with complex systems, the dichotomy of software functions (e.g., mouse handler, keyboard handler, etc.) may be totally different from the dichotomy of end user functional requirements (e.g., enter specified data, produce a specified report, etc.). In this step we are describing the software transformation requirements, and grouping them into an initial set of modules.

Step 2 - Map Data Structures Into Individual Software Functions

Select coordinate systems and states for implementing the software transformations for each function. Map these coordinate systems and states into data structures that support the functional transformations. Assign these data structures to resources and the transformations to processes within modules for each software function.

Step 3 - Map Data Structures Across Software Functions

On a module-by-module basis, determine the coordinate transformations required between functions and map the states required to support these coordinate transformations into data structures within resources. Map these transformations into processes. More complex coordinate transformations may be best put into separate modules.

Step 4 - Re-Map Data Structures To Maximize Software Module Independence

On a hierarchical module basis, review the combination of coordinate systems and states (resources), and the transformations (processes) and determine the best breakout of coordinate systems and states into resources for implementing the transformations. This requires minimizing the resources shared between modules to maximize module independence. This implies shifting data structures from one resource to another, possibly removing some resources and creating new ones. This also requires corresponding changes to the transformations assigned to processes. This implies shifting rules from one process to another, possibly removing some processes and creating new ones, or creating new modules.

Step 5 - Produce Detailed Software Architecture

Complete the detailed software architecture by connecting all of the modules. This implies connecting those processes in a module to the resources they share between modules. One may iterate between steps 4 and 5 as the architecture becomes more transparent.

IMPROVING PRODUCTIVITY

The approach outlined above borrows concepts from other engineering disciplines to improve the productivity of software development and support. The underlying concepts are enumerated below, where “GSS models” may be replaced with “VSE modules”.

Separation of Attributes from Rules

The separation of attribute structures from rule structures provides a direct mapping of the generalized state vector and transformation concept into GSS resources and processes. It provides the means for decomposition of a model into submodels that can be mapped directly back to a physical system. It also allows representation of the primitive elements - attribute structures (resources) and rule structures (processes) - using graphic symbols (icons).

Generalized Hierarchical Data Structures

Just as we can deal much more economically with generalized data vectors (as opposed to numbers) we can deal much better with generalized hierarchical data structures (as opposed to the standard "arrays" in programming languages). Real system attributes are generally defined as hierarchical structures in an engineering description. It is a natural way to organize systems with a high degree of complexity. It is also natural to follow the same hierarchical breakout of attributes in the model as are described in the real system's engineering documentation. These generalized hierarchical data structures support the direct representation of a physical system's natural hierarchy.

Reusability Analogy

In the case of electrical circuit modeling, a transistor model may require a significant effort to build and validate. Once completed, that model can be shared in many different simulations, as well as hundreds of instances used in a given simulation. Similarly, for models built using GSS. Development and validation may require significant effort, whereupon a given model can be shared in many different simulations by different organizations, as well as appear in hundreds of instances in a given simulation.

Complex models of electrical elements, e.g., transistors and transmission lines, may be composed of primitive elements and represented by higher order symbols. One can *push down* on these symbols and bring up the primitive representations that show all the detail underlying the model. More complex networks, such as groups of digital circuits in the form of gates and flip-flops can be represented using another level of hierarchy. In this manner, complexity is pushed down to the level that one wants to see it, and removed from view when it only serves to cloud the picture. This aids in both the scalability and reusability of a model.

Similarly, one can represent complex models in GSS using a hierarchy of models, wherein higher level icons are used to represent the highest level of a model, and one can push down as many times as needed to get to the primitive layer. In GSS, the primitive layer consists of resources and processes. This also aids in model understandability and independence, implying scalability and reusability.

SUMMARY

The desire to use existing engineering disciplines to improve productivity in software development and support is a growing topic in the literature. This paper exposes a technology that has evolved from CAD concepts used in electronic circuit engineering where they dramatically improved design productivity. It describes the graphical depiction of software architecture that maps one-to-one into the code, providing direct access to the code from the architectural drawing interface.

It is clear from this technology that *architecture* is the key to controlling complex software, and that architecture cannot be defined without the *separation principle* (separation of data from instructions) derived to support this technology. By focusing on architecture, and borrowing from the electronic CAD concepts, approaches to building easily *scalable* and *reusable* modules become a reality. This is the result of ensuring that the properties of *understandability* and *independence* are critical foundations of any software architecture. The ability to visualize these properties, as embedded in this new technology, supports a new approach for designing, building and supporting changes to complex software that dramatically improves productivity.

REFERENCES

- [AM] Armour, Phillip G., *Software: Hard Data*, Communications of the ACM, Vol. 49. No.9, Sept. 2006.
- [AN1] Anselmo, Donald and Henry Ledgard, *Measuring Productivity in The Software Industry*, Communications of the ACM, Vol. 46. No.11, Nov 2003.
- [AN2] Anselmo, Donald, *Tools and Software Productivity*, Software Summit, Washington, D.C., May 2004.
- [BR] Broy, Manfred, *The "Grand Challenge" in Informatics: Engineering Software-Intensive Systems*, Computer, IEEE Computer Society, 2006.
- [CA1] Cave, W. & H. Ledgard, **The Software Survivors**, Visual Software International, Spring Lake, NJ, 2006.
- [CA2] Cave, W., "The Constrained Optimal Design System," Proceedings IEEE WESCON, San Francisco, CA, 1971.
- [CA4] Cave, W.C., *Simulation of Complex Systems*, Prediction Systems, Inc., Spring Lake, NJ, June 2001.
- [GE] Gelb, A., Editor, *Applied Optimal Estimation*, MIT Press, Cambridge, MA, 1974.

- [GL] Glass, R.L., *Looking into the Challenges of Complex IT Projects*, Communications of the ACM, Nov 2006.
- [GO] Gordon, J., **System Simulation**, Prentice-Hall, Englewood Cliffs, NJ, 1978.
- [GR] Groth, R., *Is the Software Industry's Productivity Declining?*, IEEE Software, Nov/Dec 2004.
- [GSS] *GSS User's Reference Manual*, Version 10.4, Visual Software International, Spring Lake, NJ, 2005.
- [HA] Hactel, G.D. and Roher, R.A., *Techniques For The Optimal Design And Synthesis Of Switching Circuits*, Proceedings of the IEEE Special Issue on CAD, Nov 1967, pp 1864.
- [HA1] Hactel, G.D. et al, *The Sparse Tableau Approach To Network Analysis And Design*, IEEE Transactions on Circuit Theory, Jan 1971, pp 101.
- [PO] Poore, Jesse H., *A Tale of Three Disciplines and a Revolution*, IEEE Computer Society, Jan 2004.
- [PS] *High Efficiency, Scalable, Parallel Processing*, DARPA Contract SF022-035, Final Report, Prediction Systems, Inc., Spring Lake, NJ, June 2003.
- [SC] Schweppe, F., *Uncertain Dynamic Systems*, Prentice Hall, Englewood Cliffs, NJ, 1978.
- [SG1] "Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%," Press Release, The Standish Group International Inc., West Yarmouth, MA, 2003. <http://www.standishgroup.com/press/article.php?id=2>
- [SG2] "Standish: Project Success Rates Improved Over 10 Years" Software Magazine, January 2004. <http://www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish>
- [ZA] Zadeh, L.A. and Desoer, C.A., *Linear System Theory: The State Space Approach*, McGraw-Hill, NY 1963.

APPENDIX A - State Space Definition Of A GSS Model

ELECTRICAL NETWORK ANALOGIES

Electrical engineers have evolved a graphical representation for complex networks of electrical elements using interconnected icons of basic element types: resistors, capacitors, inductors, generators, etc. As illustrated in Figure A-1, these elements can be used to build up hierarchical models of higher order elements, e.g., transistors, transmission lines, etc. Such a drawing defines the differential equations that describe the dynamic changes in electrical voltages and currents in the circuit. A mathematical representation is shown in Figure A-2, where X represents a vector of state variables and U represents a vector of external driving forces. Given the initial conditions, the state of the circuit is defined for all time thereafter. In other words, the dynamic behavior of the network is defined by the hierarchical symbolic network.

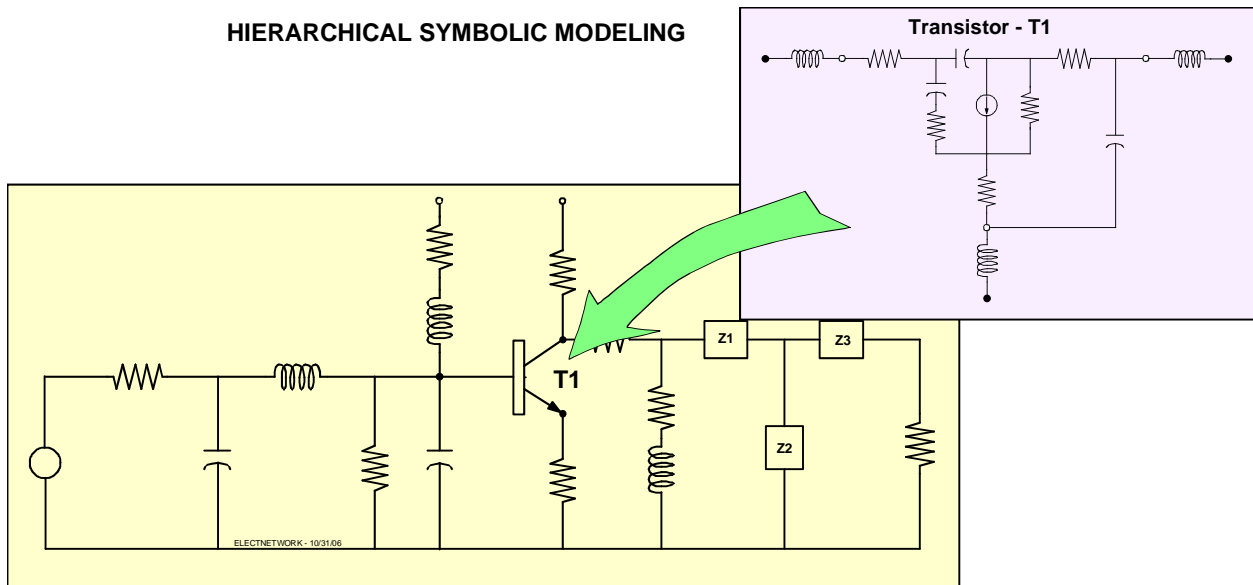


Figure A-1. Iconic representation of an electrical network.

$$\begin{bmatrix} X(T+\tau) \end{bmatrix} = \begin{bmatrix} A(T) \end{bmatrix} \cdot \begin{bmatrix} X(T) \end{bmatrix} + \begin{bmatrix} B(T) \end{bmatrix} \cdot \begin{bmatrix} U(T) \end{bmatrix}$$

Figure A-2. State Space Representation of an electrical network.

Requirement Specification And Design Specification

Designers of electronic circuits start with functional requirement specification documents and produce drawings and design specification documents, see Figure A-3, describing the network to be fabricated in production. Negotiation of changes in functional requirements and testing of proposed fabrication processes for reliability are a significant part of the engineering design process. Engineering judgment supported by mathematical calculations provide a critical part of this process. Testing is used to validate models of the nominal behavior of a network as well as variations in component values due to production and environmental changes. Ensuring reliability requirements are met in anticipated “worst case” environments is a hard constraint.

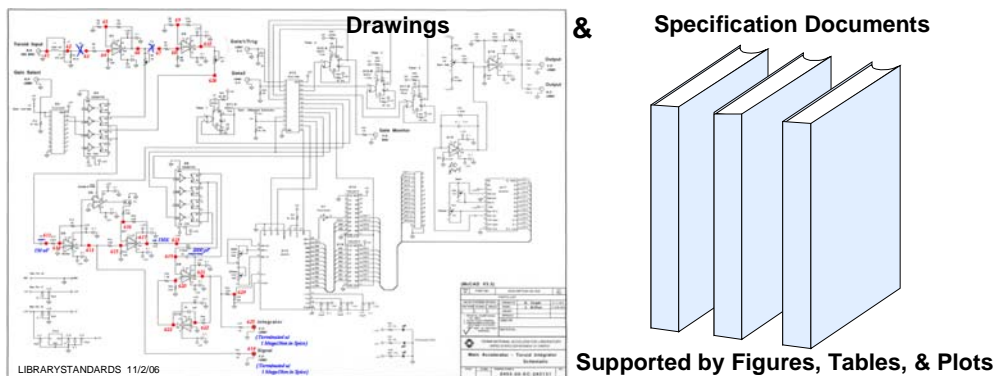


Figure A-3. Requirements and design specification documents.

Use Of Electrical Analogies For Enhanced Productivity

Because of the development of a complete and consistent theory for electrical network design, electrical analogies are used in many other fields, such as those involving mechanical design, fluid flow, etc. Additionally, characterization of nonlinear components and development of fast and accurate solutions of the differential equations representing these networks provides a highly productive facility, minimizing the burden of analysis of different designs. These facilities have been incorporated into numerous Computer-Aided Design (CAD) tools that have reduced the time and resources required to complete a complex design by orders of magnitude. Obtaining fast and accurate solutions to posed design problems provides huge improvements in productivity in the fabrication stage as well as in production of the design specifications.

APPLICATION TO DISCRETE EVENT SIMULATION AND SOFTWARE

There are four reasons why *discrete event* simulation is a good software analogy starting point. First, an electrical network analogy for this type of simulation is easy to derive. Second, it has been shown, [CA4], that discrete event simulation can be used to solve nonlinear problems in the time domain that are otherwise intractable using differential or discrete time equations. Third, this same reference provides examples of more easily understood solutions even for linear problems using a discrete event CAD system versus using differential or discrete time equations. This is because of the higher degrees of abstraction imposed when using standard mathematical frameworks. After reviewing sufficient examples, it becomes clear that such abstractions make it much more difficult to model realistic behavior in a way that is easily understood.

Finally, software is a subset of discrete event simulation when using the General Simulation System (GSS). We will start with the electrical analogy of discrete event simulation using GSS and the State Space framework used to represent electrical network equations.

State Space Representation Of GSS

Within a GSS simulation, a model only has access to a subset of the simulation state vector when a process in that model is running. We will call this the model state vector. A subset of the *model state vector* contains those resources that are contained within the model. Because this GSS state vector may contain character as well as numeric data, it is called a *generalized state vector*. The state space representation of a GSS model is shown in Figure A-4. The state vector that a model has access to consists of the following items and their corresponding information elements:

- ACCESSIBLE RESOURCES - The information contained in resources to which processes within the model are attached. Note: Shared resources may or may not reside within the model.
- SIMULATION QUEUE - The entries in the queue, including the indices it uses when it's processes are scheduled.
- SIMULATION CLOCK - The time of the simulation clock, including priority, if and when it schedules another process.
- REAL TIME CLOCK - The value of the real-time clock if it is used.
- RANDOM NUMBER GENERATOR - The current value of the random number generator seed if it is used.

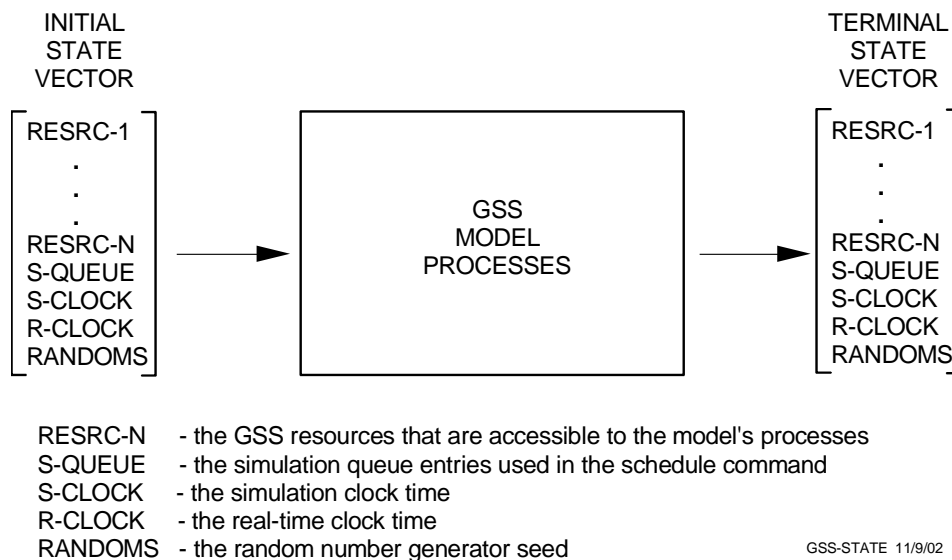


Figure A-4. State space representation of GSS.

GSS processes are scheduled based upon the logic within itself or other processes. When a GSS process *runs*, it may schedule itself or other processes at specified times in the future, or at the current time. GSS processes run in zero *simulated* time. At any time, the state of a model depends solely upon its state vector. When a process in a model runs, its *terminal state*, i.e., the value of its substate vector - when it passes control back to GSS - depends solely upon its *initial state*, i.e., the initial value of its substate vector, and the rules within the process.

When processes in another model share a part of the state vector of a given model, then any future state of the given model is, in general, dependent upon the rules in the other model, since they can change the given model's state vector.

ANALOGY TO SYMBOLIC MODELS USING STATE SPACE

The state space representation of a GSS model, Figure A-3, is analogous to a set of differential equations that represent the state of a dynamic system at any instant in time. All future states are represented by the *equations of motion* in state space notation, and the initial conditions, reference Schweppe, [SC].

In GSS, the interconnection of resources and processes, see Figures 2 and 4 (main text), is analogous to the electrical circuit drawing in Figure A-1. Each has its corresponding *rules* and *storage* underlying each primitive element. In the case of electrical circuits, there are constituent equations that describe the changes in energy storage in differential form for each primitive icon. Representation of any system element must conform to this form of change.

In the case of GSS, sets of rules operate on sets of attributes (contained in data structures) to define the elementary change relationships in a model. Using GSS, the engineering drawing shown in Figure 4, and the underlying rule and data structures, define the total state of the simulation at any point in time after the initial conditions. We call this the *generalized state space framework*.

Choosing the Most Convenient Reference Frame

As implemented in GSS and described above, the generalized state space framework supports the representation of discrete event systems as well as discrete and continuous time systems. Figure A-5 illustrates the generalized state space as providing an underlying framework for representing dynamic systems.

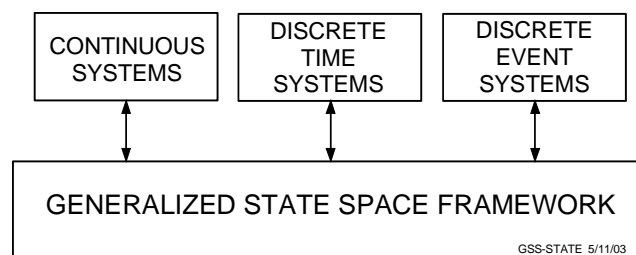


Figure A-5. Generalized State Space:
- an underlying framework for representing dynamic systems.

The difference between representations of a system's dynamics may translate into huge differences in productivity. A particular representation can be selected to make it easier to analyze or predict specific system behavior. If a system is conveniently represented by a set of differential or difference equations, then one of those representations may be best. If the system is more easily described by sets of rules operating on sets of attributes that may contain nonnumeric elements, then that representation should be chosen.

Since the advent of the digital computer, people have moved from analytical methods for integrating differential equations to heuristic algorithmic methods, especially when the systems represented are either nonlinear or nonstationary. Fast numerical algorithms for solving stiff nonlinear systems typically use complex heuristic approaches. All of these approaches can be implemented easily using GSS rule and attribute structures. As computers provide significantly greater memory and speed advantages, the space for solving problems is growing, alleviating restrictions to abstract numerical methods for solution, and allowing rapid movement toward heuristic rule-oriented approaches using complex data structures. These approaches are compared in *Simulation Of Complex Systems*, [CA4].

Having selected GSS as the overall framework, the analogy then becomes one of selecting the best set of information vectors (GSS Resources) to represent the system attributes. Depending upon how the resources are selected and structured, the rules (GSS Processes) may be much more simple to understand, build, and modify. This is determined by the *independence properties of the architecture*, i.e. the interconnection of resources and processes - *not the code!*

Mapping Into Software

Since there are only two language statements used by GSS that are specific to simulation, the rest of the language is the same as any software language. It is a very rich language that is used for the GSS counterpart, the Visual Software Environment (VSE). All of the architectural properties available to GSS users also reside in VSE. The fact that GSS is written in VSE implies that VSE can also be used to build simulations.