Software Architecture for Circuit Simulation

Stephen Maas

Abstract — **Engineering design and analysis (EDA) software circuit simulators has traditionally focussed virtually exclusively on the scientific calculations performed by the software. Software technology, however, has matured to the point where it can provide significant improvements in the ability of EDA software to address the full requirement of the engineering design flow. In this paper we describe some of those technologies.**

Keywords — **Circuit Simulation, EDA, Software.**

I. INTRODUCTION

Engineering design and analysis (EDA) software has traditionally addressed only the matter of "crunching numbers," the technical calculations that the software performs. The needs of the RF and microwave industries go far beyond this, however; they have evolved to the point where technical capability, by itself, no longer meets the complex needs of circuit and system designers. In order to achieve a high level of design productivity, the architecture of the software system—not just the analytical capabilities—must be addressed. At the same time, software technology can enhance the analytical capabilities of a circuit simulator. In this paper, we show ways in which that can be accomplished.

Our long-term goal has been to make use of modern software technology to enhance the ability of RF and microwave engineers to design circuits and systems. We have implemented many of the ideas expressed in this paper, in a commercial software product [1]. It differs from other tools in that it is designed to address design- flow issues as well as technical capabilities. We believe that the use of modern software technology, which has largely been ignored by both the research and commercial communities, can do much to provide great efficiencies in the design process.

II. DESIGN FLOW

By the term *design flow*, we mean the overall process of designing a circuit, from conception through electrical design and layout, to tape-out (in the case of monolithic circuits) or fabrication (of hybrids). Design flows that exist commonly throughout industry have significant, well-known, and commonly-encountered bottlenecks.

For example, consider a simple function: electromagnetic (EM) analysis of a part of a microstrip circuit. In traditional simulators, the structure to be analyzed must be drawn in the EM simulator's drawing tool, analyzed, and the results, in the form of scattering (S) parameters, returned to the circuit simulator. If the circuit does not work, the process must be repeated, and at each iteration, there is a genuine chance that errors will occur. Throughout the process, there is no way to view the results and ascertain that they are indeed valid. In effect, the designer depends on human infallibility to guarantee that the results are correct, and that the circuit simulated in the EM simulator is precisely the one that finally is laid out. This is a dangerous thing to do, especially when a large number of circuit components must be simulated. The same problem occurs in layout. Layout is frequently performed by a technician, based on a sketch provided by the design engineer. The circuit is copied at least twice, once by the engineer and once by the technician, and ample opportunities exist for errors to occur and for changes to be made that are not reflected in the circuit description in the simulator. Usually, considerable modification of the layout takes place, much of which may be outside the control of the design engineer.

Design-flow issues such as these can be addressed though software technology, as opposed to simulation technology. There are two aspects to this. The first is to basic architecture of the EDA software itself, designed as a complete system. The second is the use of modern software technologies that are available to the software designer. Many technologies that have been developed in the past decade can ease the task of designing high-frequency circuits and systems, and smooth the design flow. The results are improved designs, reduced error, and decreased cost.

III. SOFTWARE INTEGRATION

Initially, a circuit simulator consisted of an analytical "engine," plus modules to provide and process data (Fig. 1).

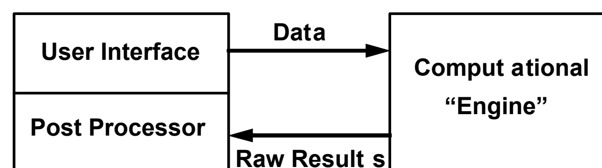

Fig.1. Architecture of a simple circuit simulator

This arrangement is fairly rigid in its ability to handle data, and it quickly became obvious that much more versatile methods were needed. It is axiomatic that tight integration

Stephen Maas is Chief Scientist of Applied Wave Research, Inc.,
1960 E. Grand Ave., Suite 430, El Segundo, California, USA 90245

between simulators can do much to smooth design flow and to eliminate sources of human error. Attempts at such integration began around 1990, and involved the use of supervisory software to control the flow of data between dissimilar tools, such as simulators, layout tools, and display modules (Fig. 2). Such software had mixed success. It happens that the supervisory function is surprisingly complex, often requiring millions of lines of code, an, as a result, d software configured in this manner often has been unreliable. However successful or unsuccessful in terms of functionality, such software introduces a layer of complexity that is not fundamentally necessary. It is necessitated only by the need to interconnect existing, dissimilar products.

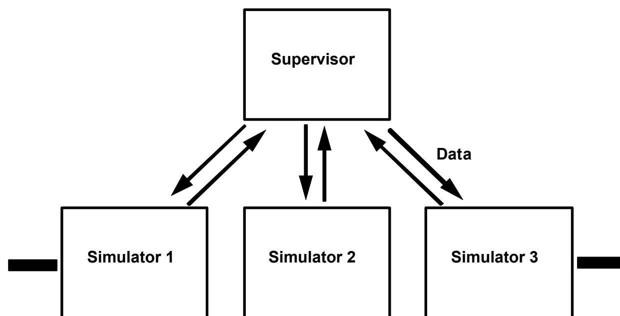

Fig. 2. Use of supervisory software to control data flow for integration of simulator components

Modern software technology offers ways seamlessly to interconnect software developed by independent parties without the need for supervisory software. One such technology is *component architecture*, which we describe shortly. Of course, the basic design of the software system, in the absence of such special technologies, can also do much to provide seamless integration between functions.

IV. Architectural Characteristics

A. Object Oriented Design

Object-oriented design (OOD) involves the use of software *objects*, sometimes called *classes*, which contain data and the functions to manipulate those data. Those objects initialize themselves on coming into scope and release allocated memory when no longer needed. Objects can inherit other objects, allowing for substantial code reuse. Object-oriented design allows functions within the software system to be self contained, so modifying one part has minimal effect on other parts of the system.

OOD exists for dealing with the complexity of large software systems, which otherwise might become unmanageable. Without the use of such techniques, large software systems become so complex, with so many interactions between parts, that it is virtually impossible to modify one part and make certain that the modifications have no unforeseen effect on other parts.

It is important to emphasize that OOD is an *architectural* method, not a programming language. Programming in C++ or some other "object-oriented" language does not automatically achieve these benefits; the underlying architecture, which is an engineering design, must be successful. Conversely, it is possible to create object-oriented architectures in programming languages that are not designed to be object oriented; for example, the original version of APLAC, an object-oriented circuit simulator developed at Helsinki University of Technology, was written in C.

B. Single-Database Architecture

In our architecture, all data are contained in a single database. Circuit elements in different simulation functions are simply different views of the *same* data. For example, a microstrip line is a symbol in a schematic window, with a certain length and width, but in the layout it is a GDSII cell showing the line. If the length, for example, is changed in the schematic, the length changes instantaneously in the layout, since both views access the *same* data item. It is literally impossible for the layout to become inconsistent with the circuit description. In this way, human errors in the layout process are substantially reduced.

C. Component Architecture

Component architecture allows the integration of dissimilar software at the object-code level. In Microsoft Windows, the implementation is called *COM*, for *Component Object Model.* COM works by defining interfaces and statistically unique identifiers that allow one software object to access the functionality of another. COM objects need not know anything about their clients beyond the interfaces. Much of the MS Windows operating system, and many Windows programs, are implemented in COM. For example, the various parts of Microsoft Office are COM clients.

COM is a binary standard, so COM objects can be written in any programming. language. They are implemented in dynamic link libraries (DLLs). COM objects need not even be located on the host computer; they can be relocated on a network.

COM allows great versatility in linking third-party simulators to a common application program interface (API). It is also valuable in allowing users to write programs that operate the API and its clients directly.

D. Component-Based Simulation Systems

Extensive use of COM interfaces allows a simulation systems to be created from a user-selected mix of simulators and model sets. We foresee the time, in the near future, when a user will be able to create a custom simulation system, addressing his particular needs, from a broad range of third-party simulators and models. Already, users of our design environment can employ various third-party electromagnetic simulators, which integrate seamlessly with the rest of the system. This capability is currently being extended to include circuit simulator interfaces.

A. Models

Models in our software do not reside within the simulator code. They reside in DLLs separate from the main executable and link to the executable through COM interfaces. This creates many advantages. It eases the problem of model updates, since only a DLL need be replaced, and it eases the management of proprietary models, as they need only be distributed to users who actually need them. In most simulators, proprietary models must be collected, compiled into a single executable, and that executable made available to all potential users. This creates a substantial management task.

User-defined models are created in precisely the same manner as our own models. A separate program, implemented as a COM "wizard," accepts the model description (names, parameters, Y parameters or I/V equations, and so on) and creates the C++ source code for the model. The code is then compiled into a DLL and loaded into the appropriate directory. The new model then loads, when the system is started, just as any other model.

B. Direct Formulation Of Circuit Equations

In the past, circuit simulators used *netlists* to enter data. A netlist is simply a list of components, excitations, and their nodal connections. Later, schematic-capture modules were included. Those modules allowed users to enter the circuit graphically, then created a netlist, which was delivered to the simulator. The use of schematic capture modules decreased the probability of error in the design of large circuits, but also created a data-flow bottleneck between the circuit description and the simulator, because the simulator could not send data directly to the schematic.

To avoid this problem we eliminate the netlist. Circuit equations are formulated directly from the component database, and the database is maintained in memory during this process. This makes the formulation process very fast, and allows for bidirectional data transfer between the simulator and the database.

C. Dependency Hierarchy

All objects maintain a dependency hierarchy. When an object (e.g., an EM structure) is modified, the information about the modification ripples through the hierarchy and all objects (specifically, subcircuits) are marked for reanalysis. Objects that do not depend on that structure are not reanalyzed. This prevents unnecessary computation, eliminating unnecessary computational effort. This is especially important for structures that require long analyses, such as electromagnetic simulations.

Figure 3 illustrates the dependency hierarchy for a circuit. suppose, for example, that we want the S parameters of object A, which might be a part of a larger circuit. Object B, another subcircuit, has been modified. Because of the dependency pointers, the system knows that object A, object B, and the two objects between B and A, which depend on B, must be reanalyzed. However, the objects to the left of A are unaffected by the changes to B, and thus need not be recalculated.

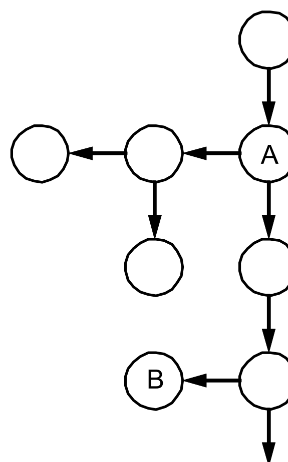

Fig.3. Illustration of the dependency hierarchy.
The circles sepresent objects, which could be S parameter blocks, subcircuits, EM structures, or similar parts of a design. The arrows show the dependency.

When an analysis is performed, the simulation begins with the user's desired measurements. Subcircuits (or other objects) to which the measurement applies are marked for analysis, and finally their dependencies are similarly marked. The analysis then proceeds in the logical manner, computing only the elements that have been marked and ignoring the rest.

D. Caching and the Speed-Memory Trade-Off

We cache all data, and delete it only when it has become invalid. This minimizes reanalysis, and is essential for such functions as real-time tuning, described below.

There is a fundamental trade-off between speed and memory use in any simulator. To minimize the use of memory, it is essential to delete data as soon as it is not immediately needed, and to reuse the memory space. Unfortunately, the recreation of these data, which is frequently necessary, requires extra computation time. If the data are saved, however, computation time is reduced, but more memory space is used.

Because of the high cost of memory, simulators developed before the mid 1990s invariably minimized memory use at the expense of speed. Today, memory is cheap, so it makes much more sense to use memory and minimize computation time. Unfortunately, the decision to minimize memory of computation is a fundamental one, and it is difficult to convert a simulator designed to minimize memory into one that minimizes computation.

E. Real-Time Tuning

It is possible to tune any linear circuit in our simulators in real time, and many nonlinear circuits as well. This is accomplished by reducing the untuned part of a circuit to a single admittance matrix, at each analysis frequency. The untuned part is then reduced and the results cached. This reduced admittance matrix is then connected to the tuned elements and analyzed. The dimension of the reduced Y matrix is simply the number nodes of the tuned elements plus the number of ports and

measurement points. The speed of tuning then depends on the number of circuit elements in the optimization, not on the size of the original circuit. Thus, arbitrarily large circuits can be tuned in real time. Figure 4 illustrates this process.

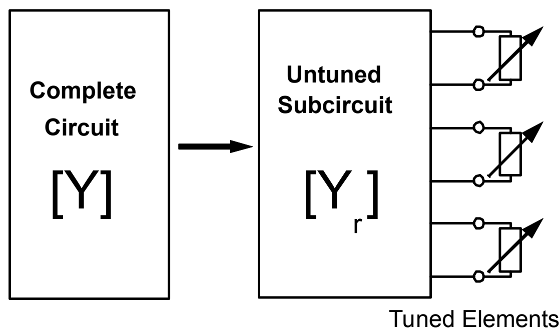

Fig.4. To provide real-time tuning, the untuned part of the circuit is reduced to a single admittance matrix, analyzed, and the Y parameters are cashed. This is performed automatically at the beginning of the analysis. Then, only a reduced matrix needs to ne analyzed when element values are modified by tuning.

F. Simulators as Objects

In our simulation architecture, simulators are viewed simply as objects, like other objects, not as the "focal point" of the system. Simulator objects can be used in a variety of ways throughout the system. For example, suppose a nonlinear device model requires S parameter data as part of its set-up process. The model can invoke the linear simulator, have it read a file of data, convert it to Y parameters, and store it. As another example, suppose that a user wants to use an EM model that does not have data for the desired dimensions in its database. The model invokes the EM simulator, creates the necessary data, and stores it in the database.

This creates an interesting and apparently backward situation in which the model uses the simulator; normally, we expect the opposite. The advantages of this arrangement, however, are increased versatility and code reuse.

G. Use of Frequency-Domain Data in Time-Domain Simulators

A perennial problem has been the need to use frequency-domain data in time-domain simulation. A practical approach is to create a LaPlace representation of an impedance or admittance function. A number of technologies have been developed to include LaPlace data in time- domain simulation, including asymptotic waveform expansion [2] and numerical LaPlace inversion [3]. Given the network function, it is also possible to realize it as an RLC network. Whatever method is used, it is necessary somehow to create a pole-zero representation of the required data (e.g., a set of measured S parameters or the results of and EM simulation).

We use a practical method in which we create a network function of the form

$$F(s) = \sum_{n=1}^{N} \frac{c_n}{s - p_n} + d + sh \qquad (1)$$

from calculated frequency-domain data. This process can be used with S parameter blocks, transmission lines, EM simulations, and similar frequency-domain data. The terms c_n and p_n are residues and poles, respectively, and d and h are real. This is easily converted to a time-domain response. Iterative and recursive numerical techniques are used for finding c_n, p_n, d, h.

V. Conclusions

We have shown that attention to software architecture, in contrast to analytical capability, can do much to enhance the design flow of an engineering organization. This is accomplished by tight integration of simulator functions, through the use of modern software design techniques. The result is improvements in engineering productivity, cost, and time to market of engineered products.

Acknowledgement

The author would like to thank his colleagues at Applied Wave Research and other institutions, too many to list individually, whose work and insights contributed to the information in this paper.

References

[1] *Microwave Office*, Applied Wave Research, Inc. 1960 E. Grand Ave., El Segundo, CA 90245, USA.

[2] V. Raghavan, J. E. Bracken, and R. A. Rohrer, "AWESpice: A General Tool for the Accurate and Efficient Simulation of Interconnect Problems," Proc. 29th ACM/IEEE Design Automation Conf., 1992, p. 87.

[3] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, 2nd. ed., Chapman & Hall, New York, 1994.