Mixed-Signal Simulation by way of the Simbus Backplane*

Dale E. Martin and Philip A. Wilsey
Clifton Labs, Cincinnati, OH
dmartin@cliftonlabs.com

Robert J. Hoekstra, Eric R. Keiter,
Scott A. Hutchinson, and Thomas V. Russo
Sandia National Labs, Albuquerque, NM

Abstract

1 Introduction

Electronic circuit simulation has emerged as a technology of fundamental importance to electrical engineers. Simulations can have relevance throughout the entire life-cycle of an electronic system, providing a mechanism for exploring the design space early in a system's life and enabling the exploration of "what if" scenarios after systems have been deployed.

As computer technology has advanced the size and scope of systems that researchers have attempted to simulate has grown. Typically the modeling needs for complex systems have been adequately satisfied using discrete event-driven simulation models. However, in many cases an interest in more detailed modeling and simulation analysis activities is satisfied only by the use of continuous system models (commonly expressed with ordinary or partial differential equations).

In addition many systems being produced today have elements that span multiple design domains. For example, mobile phones contain digital circuitry, analog cicuitry, and software, all of which must function properly for the phone to work correctly. Designing such systems can be a challenge as the tools available for simulation often focus on only one of the domains of interest.

Consequently, an important optimization technique is to build *mixed-domain* simulation models of the system where portions of the system are represented using models from different domains; for example some pieces may be represented using discrete event models and other portions are represented with a continuous model. This can reduce the total simulation runtime cost while preserving the detailed analysis results for those regions of a system model where they are necessary.

This paper will discuss a simulation backplane called "Simbus", which allows analog and digital simulators to be connected and utilized together to perform mixed-signal simulation. Our approach allows elements of the each domain to be expressed in their native format, simulate on their native simulators, and be coupled into an aggregate mixed-signal simulation. Currently supported simulators include the Savant VHDL simulator [6] and Sandia National Lab's Xyce [3] simulator — a SPICE-like circuit simulator. Simbus is available under the open source LGPL license with source code freely available.

Section 2 will describe in more detail the background of Simbus and why it is applicable. Section 3 presents Simbus' design in more detail. Section 4 discusses our experiences with Simbus and the current state of the project. Section 5 discusses the generality of Simbus; that is can it be applied to other simulators or problem domains? Lastly, Section 6 draws come conclusions on our work.

2 Background

As previously described, the design and implementation of mixed-signal designs can be very challenging. Consider simulation of a mobile phone; the analog electronics must be simulated independently from the digital electronics, and the software tested in some other fashion. This is suboptimal for a variety of reasons, but perhaps the most important is that design problems often occur at the domain boundaries and independent simulations will not always expose them.

One alternative is that designers can move to the domain that is the "least common denominator", in our example, this might mean simulating everything in the analog domain on some SPICE-like simulator. This is inefficient at best; the phone's embedded microcontroller will be simulated at the transistor level even if a behavioral simulation would have been sufficient. In general this approach will use far more computational resources than necessary.

Another issue with this approach is that one must be able to describe the digital circuits in terms of SPICE

*This work was supported by Sandia National Laboratories, Albuquerque, NM under Contract Number 29727 .The authors thank them for their support.

netlists. This often is cumbersome at best. Digital design is often done in languages like VHDL [1] or Verilog [5] which do not easily convert to netlists without completion of synthesis and/or layout, which happens very late in the design cycle.

Sandia National Labs has interest in efficient simulation of circuits at all levels of abstraction. In supporting their simulation efforts, personnel at Sandia have developed a variety of internal tools in addition to using commercial tools. One such team at Sandia has been working on a SPICE-like simulator called "Xyce" for some time. They have several goals: (i) allow parallel simulation of SPICE-compatible models on platforms ranging from clusters of PCs to supercomputer class machines, (ii) use novel numeric techniques to achieve greater accuracy and more efficient simulation than SPICE, and (iii) allow development of advanced capabilities beyond what traditional SPICE provides, such as analysis of thermal effects, aging, radiation, and so on.

Clifton Labs has been working on a project implementing an LGPL parallel VHDL simulator for some time. This system is composed of several components: *Savant* is the VHDL analysis front-end and code generator; *tyvis* implements a VHDL simulation runtime support environment; and WARPED which is a parallel discrete event simulator. When used together, these systems can be used to execute VHDL in parallel. For the sake of simplicity, in this paper we will refer to this system as *Savant*.

Sandia personnel were interested in combining Savant and Xyce into a system that would allow efficient parallel execution of mixed signal designs. In our initial discussions into how to best provide mixed-signal simulation we talked with users about migrating their mixed-signal designs to VHDL-AMS [4] and supporting execution of VHDL-AMS designs using a Savant/Xyce hybrid. They had several concerns with this approach. Chief among these concerns were: (i) there was a feeling that the adoption of VHDL-AMS has been relatively slow and therefore that commercial tool support was limited; and (ii) Sandia strongly desired to preserve the considerable investment in time and money that they had extended in validating their SPICE models. They felt that they wanted to be able to use the intellectual property of their SPICE models with as few modifications as possible.

In summary, the goals of the system to be developed were:

Support mixed-signal simulation: Enabling this functionality was the primary goal in developing the system.

Leverage parallel infrastructure: Since Sandia has a large investment in parallel computational platforms, this was a secondary goal.

Use existing models: Given the large investment that Sandia has in models, the ability to use them relatively unmodified was considered important.

Leverage existing software: Software development is expensive in both terms of time and manpower required, so utilizing existing software where possible was considered an important goal. An additional element of the this goal is that we did not want to preclude the use of commercial software with the backplane.

Keeping these goals in mind, section 3 will describe Simbus in more detail.

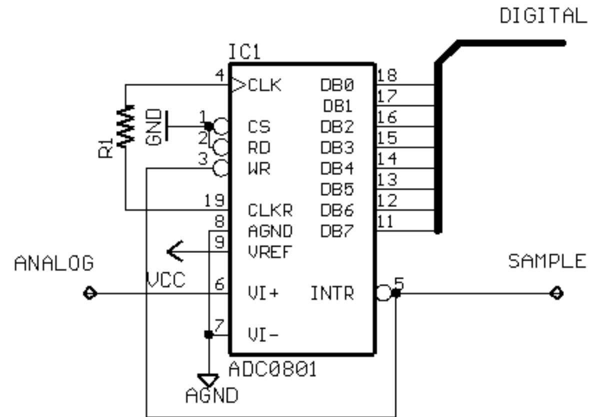

Figure 1: *Domain conversion from analog to digital through an ADC device.*

3 The Simbus Backplane

In designing the Simbus Backplane, one of the first questions that arose was how the analog and digital simulators will interact with each other. To answer this question we looked to real mixed-signal systems and see how they interact across domains. Typical interactions occur through analog to digital (A/D) or digital to analog (D/A) converters, or through other circuitry that is acting as one of these devices.

Figures 1, 2, and 3 show representative examples of each of these types of converters. Figure 1 illustrates a conversion from the analog to digital domains using an ADC device. The analog signal varies in the continuous time domain, but the digital domain only sees updates when `sample` is asserted, limiting information flow across the domains. (The ADC device in this example is set up in a self-clocking mode, hence the connection from `CLKR` to `CLK`).

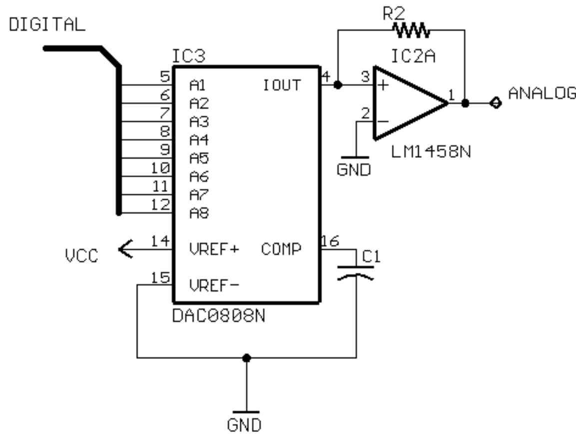

Figure 2: *Domain conversion from digital to analog through a DAC device.*

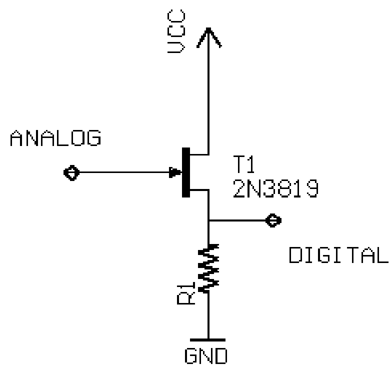

Figure 3: *Domain conversion from analog to digital through a n-channel mosfet acting as a switch.*

In Figure 2, we show a digital to analog conversion using a DAC device. The particular DAC device shown in this figure continuously generates analog outputs as the digital inputs vary — other devices often have clock inputs used to latch in the digital data and assert the new analog output. The amplifier serves to convert the current output into a voltage output with enough current driving capability to allow easy connection to other circuitry.

Finally, Figure 3 depicts domain conversion from analog to digital using an n-channel MOSFET acting as a switch. This case presents a more interesting domain conversion, because it represents a case where the analog domain can "push" a value into the digital domain with no request flowing from the digital domain. As the input to the FET varies, it will convert the analog signal into a digital one or zero.

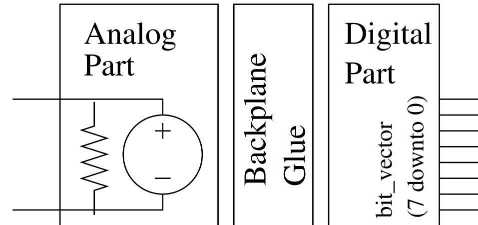

Figure 5: *Modeled D/A converter.*

Extending this methodology to our simulation system was a natural fit; explicit devices would be utilized to allow interactions across the boundaries. An illustration of the organization of a SAVANT/Xyce simulation running on the Simbus backplane can be seen in Figure 4. In this figure we see that all system interaction occurs through domain conversion devices, specifically A/D and D/A devices. Each system operates in its "normal" mode, but with connections to the backplane through additional models representing A/D and D/A devices.

Each modeled converter requires three elements: (i) the analog representation, (ii) the digital representation, and (iii) the backplane glue. Figure 5 illustrates a modeled D/A converter. Note that the analog part contains elements of a physical device including the notion of a voltage source and output impedance; these elements are required by the continuous simulator to accurately simulate the device, as the driving capability and output impedance will define its interaction with the rest of the model. The digital part of the model simply defines the VHDL interface into the model; in this D/A converter the VHDL entity description might look like the following:

```
entity DAC is
 generic (device_name : string;
          width : natural );
 port
  (input: bit_vector(width-1 downto 0))
end DAC;
```

The last part of the model is the part we have referred to as the "backplane glue". This part of the model is responsible for transporting events coming from the digital simulator to the analog part of the model, which then converts the events into voltage/time tuples for the voltage source, which acts much like a piecewise linear source in a traditional SPICE simulation. The linkage between the digital part of the device and the analog part of the device occurs by the device naming; the netlist and the VHDL the domain converters must have matching names.

Both the analog and digital simulators are free to define a variety of models for domain crossing devices. These

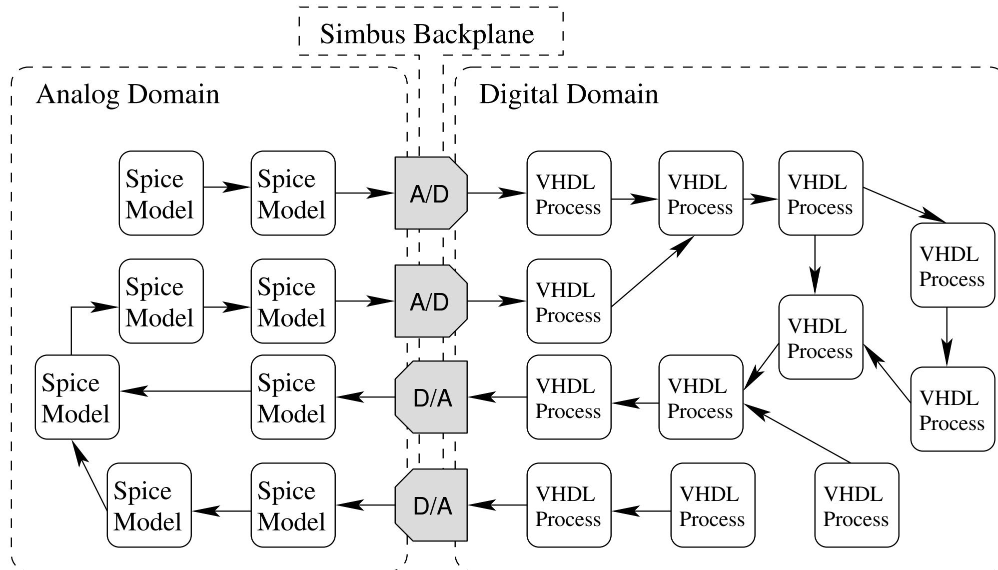

Figure 4: *The organization of a Simbus simulation.*

models can be simple like that shown in figure 5, or could be more complex and representative of a specific device used in a real design, with varying output impedance and non-linear conversion times.

At the most basic level the backplane exchanges quantity/time tuples and bit_vector/time tuples and does not concern itself with the semantics of the values being passed. Simbus defines an interface for domain crossing devices which the models implement. When the simulators start up these devices register with the backplane and it connects the pieces of the device together.

3.1 Startup and Signal Interconnect

While we have access to the source code for both Xyce and Savant, we were concerned about designing a backplane that required extensive modification to the simulators that would interact with it — and this would violate one of the goals we had determined for the system. Therefore the design had to allow for the possibility of interacting with simulators for which we did not have the source code and that we could not link against. In order to meet these criteria we devised the following design:

***Simulator startup uses the factory pattern*:** The factory pattern [2] is a widely used design pattern for allowing construction of objects through an abstracted interface. This allows the backplane to make calls to a known factory interface rather than calling constructors directly.

***Runtime linkage of factory classes*:** The backplane is not compiled or linked against the simulator factory classes; instead, these factories are dynamically loaded at runtime when the backplane starts up. This is achieved through the shared library loading mechanism found on modern operating systems.

***Deferred configuration of simulators*:** The backplane configuration is read through a file. The file contains configuration information for the simulators as well. This is achieved by defining a structure to the configuration file but not defining the semantics within the simulator-specific sections. The backplane passes the simulator's configuration to the simulator at startup, and the simulator itself handles its own configuration.

This architecture allows for a large degree of flexibility, and even the possibility to connect to commercial/proprietary simulators as is discussed in section 5.

3.2 Simulator Scheduling

Scheduling is another area of interest in a simulation backplane such as Simbus. The current implementation of Simbus does "ping pong" scheduling, where control is passed from simulator to simulator for specific timeslices. Our original implementation used a fixed timeslice chosen as the minimum propagation delay across the backplane. This approach can be very expensive in terms of simulation performance, in particular if any devices such as

those found in figure 3 are employed in a simulation. In this case the time slice could be as small as the switching time of a single transistor.

The second scheduling algorithm that has been implemented still uses a “ping pong” approach but is capable of expanding the timeslices for which we can schedule. This is achieved by querying the discrete event simulators for their next activation time. The simulator that is furthest behind is scheduled for the duration between “now” and the next scheduled activation time; if the executing simulator generates an event for the backplane it relinquishes control to the backplane which does a new time step calculation. This imposes a limit on the backplane that at most one continuous simulator may be connected at a time, although this limitation can be largely mitigated by combining multiple netlists into one.

Ultimately one would like to allow all simulators to execute concurrently to achieve maximum performance on parallel platforms, however causality issues can make this problematic. We have plans to look into this possibility in the future as both Xyce and Savant have rollback capabilities and could potentially recover from causality violations using rollback.

Each scheduling approach has impact on performance and implications with regards to the APIs of the connected simulators. This is an area which will continue to be studied.

4 Experiences

The initial implementation of Simbus is realized in 4200 lines of C++ code[1] including the interfaces for Savant and Xyce. The system has been developed in a GNU/Linux environment and a port to FreeBSD is underway.

Initially we used hand-rolled analog models for testing purposes while developing the Savant interfaces. Our initial examples model communication from from the analog simulator to the Savant simulation, from the Savant simulation to the analog simulator, and finally in round-trip fashion.

After developing these examples we were able to hand them off to developers at Sandia who developed the Xyce interfaces. In the example circuits they replaced our hand-rolled analog simulators with real netlists executing on Xyce, and this system is currently operational. Our current work is to expand the systems we are modeling beyond small examples to real, relevant circuits.

Our experiences with the system so far have been encouraging and given us confidence that the approach will work and that it will scale. As we work with larger designs we will be able to carefully study performance, including scheduling, communication overhead, and other issues that become apparent.

[1] generated using 'SLOCCount' by David A. Wheeler

5 The Generality of Simbus

While the immediate goal of Simbus was to bring together Xyce and Savant to enable mixed-signal simulation, our intention has been to allow other tools to be connected to Simbus as well. As was mentioned previously, Simbus' API design utilizes thin interfaces defined in terms of pure virtual classes. This precisely defines the functionality that Simbus requires to interact with a simulator. Also, using the dynamic linking of shared libraries means that Simbus can operate with simulators that it has not been linked against directly. We chose the LGPL licensing specifically to allow integration of proprietary systems with Simbus.

One of the first commercial tools that we have examined in terms of integration is Mentor Graphics' ModelSim VHDL simulator. ModelSim is very popular in the EDA marketplace so it is of a high degree of interest. Some versions ModelSim provide a VHDL “Foreign Language Interface” (or VHDL-FLI). Our initial investigations into the VHDL-FLI lead us to believe it would be possible to integrate ModelSim with Simbus and use it as a digital simulator in a mixed-signal simulation. The VHDL-FLI clearly has the capabilities to read and write signals that would be required; the issue that requires further study is if the API provides the necessary functionality required for efficient scheduling.

Another area of interest is integrating with an open source analog simulator such as SPICE 3. A variety of issues keep Xyce from being released under an open source license and without an analog simulator Simbus' usefulness is limited to couple together multiple digital simulators.

6 Conclusions

A significant amount of progress has been made towards the goal of enabling parallel mixed-signal simulation using Savant and Xyce. The capability has been demonstrated and large strides have been made in demonstrating the applicibility of the of the system to real problems.

Future work will focus on the following issues:

***Performance*:** As experience is gained with larger models, performance will become a key area of investigation. Scheduling will continue to be of interest,

along with reduction of communication and other overheads. The ultimate level of performance would be for the aggregate system to be bound by the performance of the slowest simulator (which could vary on a model-by-model basis.) This would imply zero overhead which is unrealistic, but a benchmark to bear in mind.

***Integration with Additional Simulators*:** As was mentioned in section 5 integration with additional simulators is of interest. We would like to see integration with both proprietary and open source simulators. In addition to ModelSim and SPICE 3 which were mentioned previously, we can envision integration with Icarus Verilog, the irsim switch-level simulator, bindings for various programming languages beyond C/C++ like Java and perl, and a variety of commercial tools.

***Investigation into User-Interface Designs*:** The current inputs into a mixed-signal simulation using Simbus are a VHDL model, a netlist, a Simbus configuration file, and any inputs supported by Savant and Xyce. The current outputs are those that are possible using Xyce and Savant — text file output.

Today's engineers are accustomed to using schematic capture programs for input, and waveform viewers for output. How to achieve these capabilities "cleanly" in a backplane system is an interesting question and one that warrants investigation. The popularity of open source may be of benefit here as several excellent programs such as "gschem"and "gtkwave" might be candidates for integration in some form.

Simbus has demonstrated the feasibility of the backplane approach for integrating Xyce and Savant, and we are confident that it has application to other tools as well. By releasing Simbus as free software, we hope to make a contribution that can be used to solve a variety of real problems faced by today's electrical engineers.

References

[1] ASHENDEN, P. *The Designers Guide to VHDL*. Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1996.

[2] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Reading, MA, 1994.

[3] HUTCHINSON, S. A., KEITER, E. R., HOEKSTRA, R. J., WATERS, L. J., RUSSO, T. V., RANKIN, E. L., AND WIX, S. D. Xyce parallel electronic simulator. Sand Report SAND2002–3790, Sandia National Laboratories, Nov. 2002.

[4] *IEEE Standard VHDL Analog and Mixed-Signal Extensions*. New York, NY, 1999.

[5] THOMAS, D. E., AND MOORBY, P. R. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, MA, 1991.

[6] WILSEY, P. A., MARTIN, D. E., AND SUBRAMANI, K. SAVANT/TyVIS/WARPED: Components for the analysis and simulation of VHDL. In *VHDL Users' Group Spring 1998 Conference* (Mar. 1998), pp. 195–201.