THE ARCHITECTURE OF A HIGHLY INTEGRATED SIMULATION SYSTEM

Michel Heydemann Alain Plaignaud Daniel Dure

EUROPEAN SILICON STRUCTURES
72-78 Grande Rue - 92310 SEVRES - FRANCE
tel : (33-1) 4626-4495

Abstract

A mechanism for integrating simulators into CAD systems has been implemented in order to provide high-performance interaction during the course of a design. This mechanism replaces slow text-based interfaces by a persistent programming technique where the database is viewed as an extension of dynamically allocated memory. Organization of simulation data and implementation of the interface mechanism are described.

1 - Introduction

Much effort has recently been spent on developing simulation systems, either to design high-speed simulation algorithms at the various simulation levels [1,2], or to design multi-level and mixed-mode simulators that embed level specific algorithms into the same program [3,4]. However, implementing high-performance simulation systems also involves other aspects that have not been subject to much research and development effort, in particular high-speed communication between simulation and the other components of the CAD system, e.g. design capture, pattern description, result editing, back-annotation of capacitances from physical layout, and back-annotation of simulation results into the design capture system.

For instance, virtually all existing CAD systems use slow textual interfaces to their simulators, spending significant amounts of computing time first to translate database information into text input files for the incorporated simulator, and then to parse these files in order to create data structures used at runtime by these simulators.

Section 2 discusses how simulation is usually integrated with the other components of CAD systems. Section 3 discusses in more detail the data used internally by a simulator, while section 4 presents the persistent programming technique used in our system. Section 5 describes the actual implementation.

2 - Conventional integration of simulators within CAD systems

As shown in Figure 1, a design capture subsystem like a schematics editor, or even a text editor, is initially used to enter design data into the CAD database. This tool may also annotate the database in order to let the simulator know what particular nets are observed during a given simulation and what simulation levels should be used for each hierarchical cell. Back-annotation from the physical design subsystem is performed later on, in order to take net capacitances into account for precise timing information.

A simulator has to read design data from the database in order to create its own specific data structures that will be used by the simulation algorithms. Results produced during simulation may either be stored into files that contain waveforms for the whole simulation interval, or back-annotated into the database at a given simulation instant for display by the design capture subsystem. Finally, the simulator will produce checkpoint files that contain a complete image of the simulator data at a given timepoint. Such a file will be used to restart simulation from this particular point without having to run the whole simulation again.

Waveform editors and waveform compiler may be used to create, update, and display the data contained in result files.

Existing simulators read input data as textual netlists, which are created from database when simulation is requested, and must be parsed by the simulator to create its own data structures. Similarly, output results like SPICE waveforms are provided in some fixed format which must be parsed and translated into what is used by the waveform analyzer. Back-annotation of capacitances from layout is generally handled by recreating new netlists that incorporate the new values of delays or capacitances.

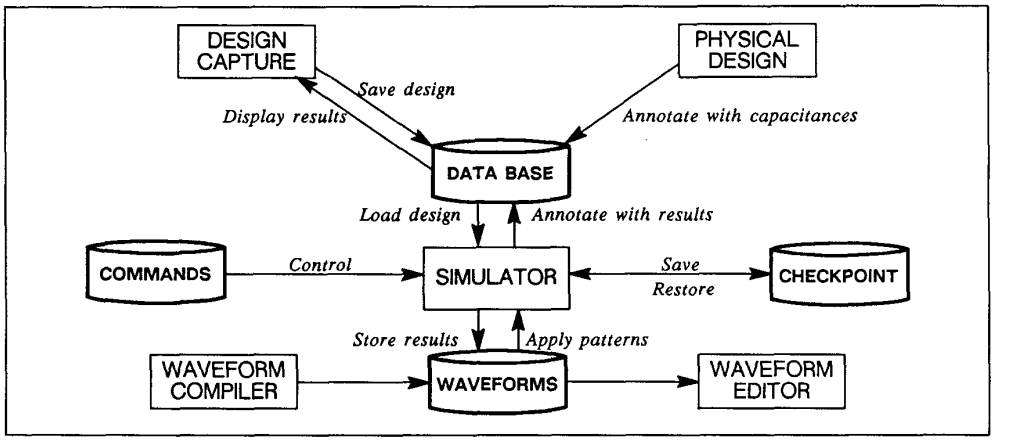

Figure 1 : Interaction of simulation and other components of a CAD system

3 – Simulator internal data structures

Figure 2 shows the data used internally by a simulator, where data can be split in several ways. Data can be partitioned according to its use, into *algorithmic data structures* used by simulation algorithms, and *access data structures* used to relate algorithmic data to the outside world. Data can also be partitioned according to what it represents, into *network data structures* associated with nets and components of the simulated design, and *observation data structures* associated with simulation control and observed results.

When simulation is applied to a circuit, cell information, design hierarchy and net data, shown as *design tree* and *nets* in figure 2, is loaded into memory with simulation specific data (views) like functional models, delay or capacitance information. It is used to to create the *circuit objects* that will be used by simulation algorithms and which represent instantiated cells of the network at their requested simulation level. These objects, are associated either to nets and components of the design like events, gates, transistors, functional models, or to simulation algorithms like the event queue for logic simulation, and conductances of the admittance matrix for circuit simulation. Simulation requests are stored in the *Observation Directed Acyclic Graph*, which maintains hierarchical information and relationships between the various *Observation Objects* created to handle the various simulation requests (observed results, breakpoints, etc...), in the same way that the design tree provides the structure of simulation objects for the simulated network. Note that observation hierarchy does not correspond to a tree, but to a directed acyclic graph since one net may appear in more than one simulation request.

4 – An efficient interaction mechanism

Instead of using slow translation between database information and simulation input text files, plus various ad-hoc interfaces for back-annotation of capacitances and simulation results, we use a single mechanism to handle initial loading of design data, writing and reading checkpoint files, and exchange of additional data for back-annotation like net capacitances or simulation results.

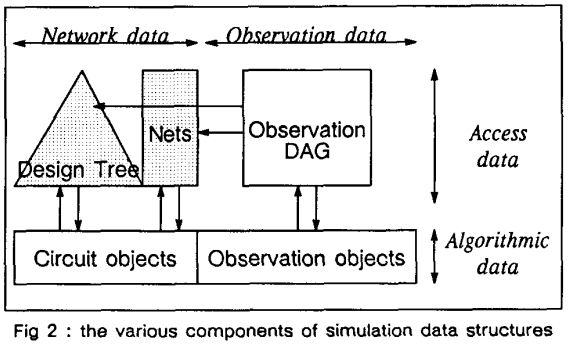

Fig 2 : the various components of simulation data structures

Access data is used either to initially create algorithmic network data and to allow for interactive communication with the simulator during the course of a simulation run. When running in batch mode, access data can be dropped to save memory.

The whole data of Figure 2 must be kept to save the current status of a given simulation, and will be stored in checkpoints file of Figure 1. A subset of the whole data, namely design tree and net data, is shared with the other components of the design system, while the rest is purely simulation specific.

The main idea consists of keeping database information under a format which is exactly the one used at runtime by each tool in memory, and which is usable without translation, through a mechanism known as *persistent heap* [5-7] where files in the database are considered as a direct extension of the dynamically allocated memory. Persistent heaps are similar to the facilities of high-level programming languages, like `new` in Pascal or `malloc()` in C, with the extra property that allocated data can exist beyond the execution span of a given program by being placed in a database, so as to be reused by the same or another program. Beyond simple memory allocation, this requires the ability to write and read binary data structures, while still maintaining pointers that could exist between allocated data, potentially relocated to a new area of memory. Such a mechanism offers the following three advantages: high-speed interfaces are available since translation is not required, except for pointer relocation; data consistency between applications is ensured since they all share the same physical data; finally program complexity is significantly reduced, since communication is handled by the same mechanism, in a manner which is totally transparent to programmers.

Since data may be written or read selectively in the data base, it is necessary to provide a mechanism for selective access to information contained in such a persistent heap. The solution for that is to partition the heap into zones that may be manipulated independently. For instance, one zone may be allocated to store net capacitances written by the physical design subsystem and read by the simulator. In the same way, another zone is used to store net values at a given time which are written by the simulator and read by the design capture subsystem in order to display them on the schematics, or by the waveform editor.

Figure 3 shows how simulation persistent data is partitioned into the various zones that are necessary to perform the above-mentioned interactions between a simulator and the other components of the CAD system. Arrows indicate the source from which data is created. The subset of data which appears within the upper shaded rectangle of Figure 3 is shared with all components of the CAD system and corresponds to the shaded components of Figure 2. It contains the following data:

- *Local cell information*, i.e. a single copy of the information shared by all instances of a given cell, like identification of ports, instances of other cells, local nets, etc... Application specific views of components, such as functional models or mask layouts are also stored as heaps that are not shared, but which can be selectively loaded by the application which uses them.
- *Global design information*, i.e. an expanded hierarchical tree where each cell instance has a unique record, and the global net structure which breaks cell boundaries. Both types of data can then be annotated in order to store instance or net specific parameters, like delays, capacitances, or simulation results.

The lower shaded rectangle contains the data which may be exchanged between application programs during the course of a design. When these programs communicate, data is written into or read from the communication heap, as shown in Figure 4. When reading or writing a checkpoint file, only simulation specific data appearing in the lower non-shaded rectangle of Figure 3 is manipulated, since the rest is already contained in the database.

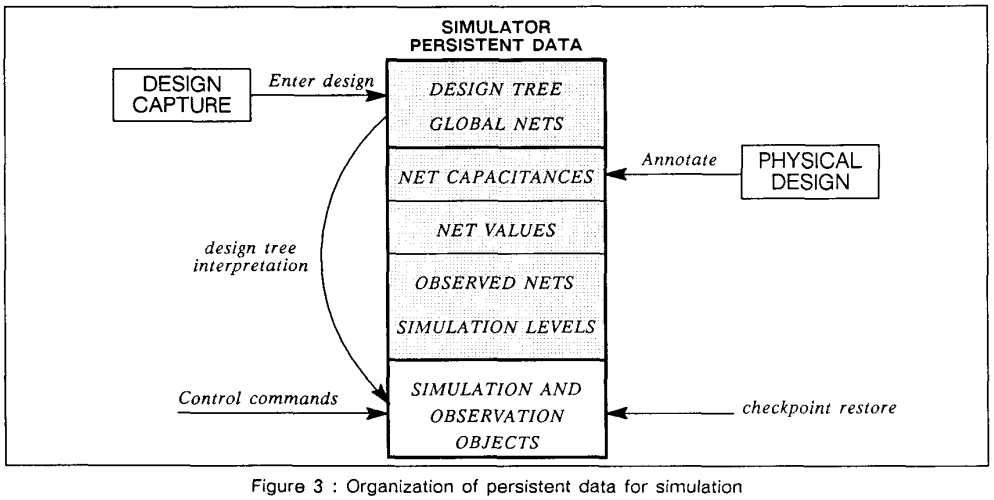

Figure 3 : Organization of persistent data for simulation

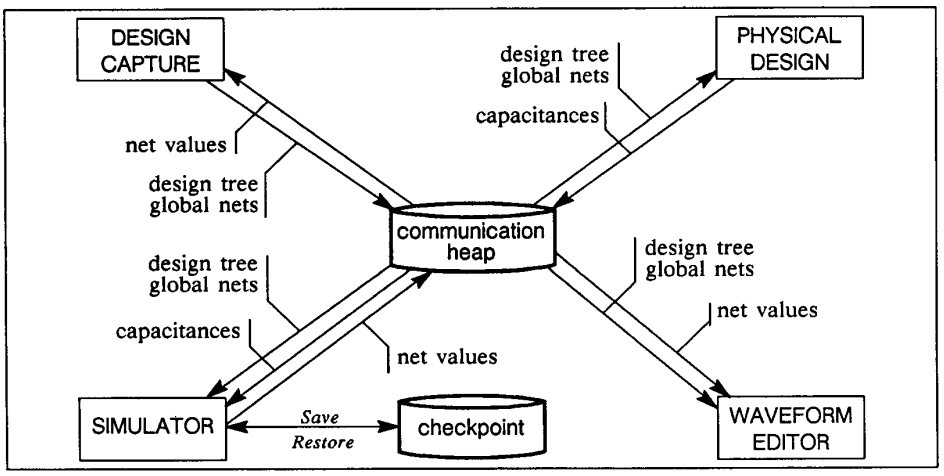

Figure 4 : Communication flow

An interrupt server is associated with the communication heap, in order to regulate access since only one application should write at a given time. It also oversees handshake between applications, when synchronous access is necessary. For example, simulation should not overwrite the net-values-zone before any interested application has read it.

5 -Implementation of the persistent heap mechanism

In addition to the usual information necessary for pure memory management, e.g. the lists of allocated or free blocks of memory, implementing a persistent heap requires the storage of extra information about the data objects stored within blocks, mainly to identify pointer fields.

One solution is to associate a map with each user-defined data structure, which maintains information about its various components, i.e. whether they are pure data or pointers, whether a pointer must be relocated or zeroed out, the type of the pointed data area (bss,data,text), the nature of the pointed data, etc... The heap can use that information for automatic relocation of pointers when reading or writing the database. Each map is identified with a unique map parameter which is provided by the C preprocessor discussed below, and which is used to call the memory allocation procedures, thereby replacing the size parameter usually expected by such procedures, as in the `malloc()` C function.

The data management system knows all the base data types that are used by data structure declaration, like signed and unsigned integers of various sizes, floating point data, character strings, etc... and the maps that are maintained for each data structure recorded by the data manager are very similar to those that would be necessary for a C interpreter.

To achieve this, the grammar for C declarations and definitions is extended with new keywords and rules that allow declaration of persistent data types and data objects. These extended declarations are parsed to produce two pieces of standard C code that can be compiled with the rest of the application source code, which contain respectively standard C structures and map descriptions, as shown in figure 5. Map identifiers, as used by the allocation procedure, are computed from the original names appearing in the declarations.

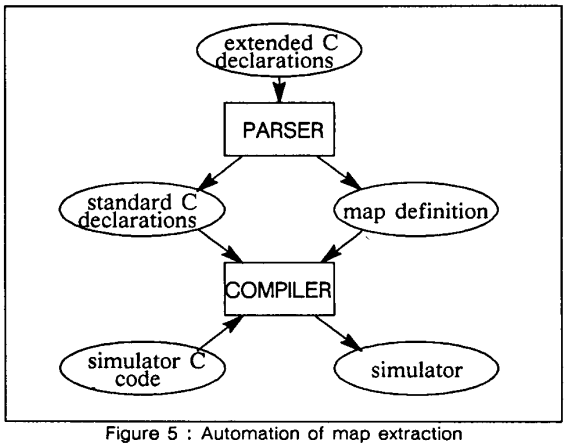

Figure 5 : Automation of map extraction

Once such maps are available, debugging utilities are easily built on top of the system, such as improved dbx-like pretty-printing of data using the printing functions that are associated with the data fields. For instance, a pointer field can be printed with an identification of the pointed object, and bit-packed data like zoomwords for gate-level simulation can be expanded and interpreted in a meaningfull manner. More generally, each type of data may have a default set of attributes which can be replaced when a given data structure is declared. Such flexible and automated debugging facilities are particularly useful for simulation, where data structures are complex and the total amount of data is very large.

Data is transferred in one block between files and memory, then data objects are sequentially relocated by order of appearance, according to their type: Pointers to static data are not relocated, unless a heap is used by different programs; pointers to file descriptors represent a special case because file states must be restored when reading the heap. Pointers to relocatable data are compared to memory management segment bounds [5], and the base of the matching segment is subtracted before write, and added after read. The CPU time spent in pointer relocation is roughly the same as the CPU time required to transfer data from/to files.

Another possibility for communication would be to use the shared memory feature of some operating systems instead of the file system. This would avoid to duplicate the shared data stored in each application at runtime, but software portability would be lost, and each application could not make free use of pointers from shared memory to non-shared data private to itself.

6 - Conclusion

A highly-integrated simulation system has been implemented which avoids repeated translation of database information into various ad-hoc text files used as input by the simulator. Communication between various tools in the CAD system is performed by using persistent programming techniques where the database is viewed as an extension of dynamically allocated memory. In addition, this persistent heap mechanism allows for enhanced debugging facilities and offers a way to perform input and output by a single tool which is totally transparent to its user and independent of the manipulated data.

7 - References

[1] Barzilai Z., *et al*, *"HSS - A High Speed Simulator"*, IEEE transactions on CAD, Vol. CAD-6, No. 4, July 1987.

[2] Ulrich E., *et al*, *"High-speed concurrent fault simulation with vectors and scalars"*, Proc. 17th Design Automation Conference, June 1980, pp 374-380.

[3] Newton A.R., *"The simulation of large scale integrated circuits"*, Memorandum No. UCB/ERL M78/52, July 78.

[4] Kleckner J.R., *"advanced mixed-mode techniques"*, Memorandum No. UCB/ERL M84/48, June 84.

[5] Atkinson, M.P., *et al*, *"Algorithms for a Persistent Heap"*, Software Practice and Experience, Vol. 13, No. 3, March 1983.

[6] McCaskill G., *et al*, *"An EDIF Based Design Database"* European EDIF Forum, Brussels, Belgium, September 1987.

[7] Zara R. V., Henke D. R., *"Building a layered database for design automation"* , Proc. 22nd Design Automation Conference, June 1985, pp 645-651