Consistent User Interface and Task Level Architecture of a TCAD System

S. Halama, F. Fasching, H. Pimingstorfer, W. Tuppa and S. Selberherr
Institute for Microelectronics, Technical University of Vienna
Gußhausstraße 27-29, A-1040 Austria

Abstract

A modern TCAD framework must provide an interactive environment for comfortable tool control and supervision. In this paper we investigate the design of the interactive user interface and the implementation of the extension language from the viewpoint of software architecture. The use of unified concepts assures the quality and consistency of the TCAD framework.

1 Background

Besides a well balanced set of simulation tools, there are some indispensable key features for the acceptance of a TCAD framework: a common format for efficient data interchange and communication between tools, an adequate tool control mechanism, a flexible extension language, and an intuitive graphical user interface.

Our binary PIF implementation [1] serves as a database for the VISTA system [2]. It is important to note that both the binary and the ASCII representations of PIF are a subset of and hence conform to LISP syntax.

The other big components of a framework – interactive shell and user interface – demand careful attention to maintain consistency throughout the whole framework. It turns out that adhering to basic guidelines for good software architecture leads to a number of value adding features.

2 Demands

2.1 User Interface Demands

The importance of the user interface for advanced process and device simulation is often underestimated. The rapid development of software tools which use increasingly complex physical models and numerical methods calls for a generic user interface that meets the following demands: It must fit into a TCAD system, in our case consisting of a database interface [1], simulators, and execution control [3]. It must be flexible enough to accommodate easily for the unforeseeable variety of new tools without the behavior of the user interface itself being changed. On the other hand it should be possible to tailor the user interface to the specific needs of the user and to comply with existing environments, regardless of the underlying simulation tools.

2.2 Task Level Demands

On the task level, a means for efficient tool integration (in terms of execution control) as well as an extension language (which must be implemented as an interpreter) is required: Compared to a compiled language, an interpreted language has intrinsic advantages regarding quick turnaround cycles in rapid prototyping, macro modeling, and tool sequencing.

Process flow descriptions, simulator input specification, execution control parameters or even parts of physical models must efficiently be stored in the extension language structures.

The basic data types or objects of the data level implementation should be easily accessible from within the extension language environment.

3 Software Architecture and Portability

A typical size of a classical single simulation tool lies in the range of 30k to 50k lines of FORTRAN source code. Figure 1 shows a few examples. A framework for tool integration like VISTA, consisting of a procedural interface for database access, an interactive user interface, visualization tools and an extension language shell, requires about 140k lines of code in different implementation languages (predominantly C).

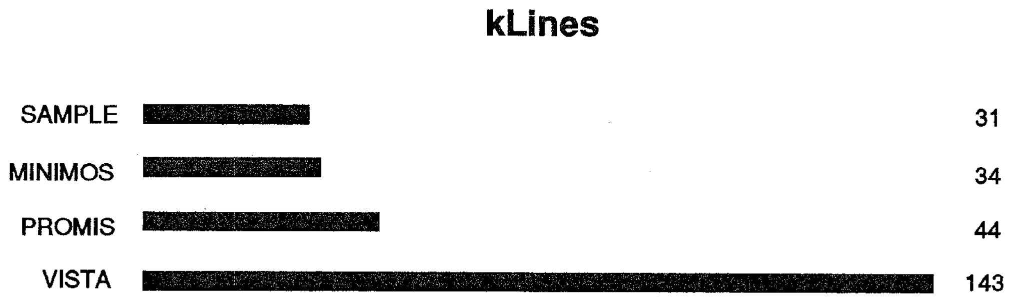

Figure 1: Comparison of source code size

It is obvious that special efforts are needed to ensure the consistency of such an amount of code.

Good software architecture is an indispensable prerequisite for the maintainability of the software system. As it is further developed by human beings, it should be possible to understand the basic structure and the details of the system with a reasonable effort. Therefore, the design and implementation of such a system must adhere to simple and clear basic concepts. It is also desirable to keep the number of different concepts as small as possible. The use of a high level of abstraction for implementing the desired functionality greatly reduces the effort for understanding and maintaining the system.

Operating system independency is a necessity for the acceptance of a TCAD framework. The enormous variety of computing environments which are used in the field of process and device simulation must not be neglected. The only feasible way to address this issue is the consequent and disciplined use of highly portable C and FORTRAN 77 programming techniques, whilst providing the source code to the end-user.

4 Implementation

4.1 User Interface Implementation

The previously imposed requirements for the user interface can only be met by a rigorous bottom–up approach: to implement this functionality, the VISTA user interface agent is built up from a set of so-called widgets which consists of the standard *Athena* widget set [5] and additional specialized widgets. The latter include vector graphics capabilities, comfortable specification of integer and real values, a visual programming interface for simulation flow control and a widget for interactive modification of PIF objects (the PIF Editor).

This extended widget set is accessible from within the TCAD shell [3]. With LISP as implementation language, a layer of functions is provided which makes the programming interface fairly independent of the underlying (*Athena*) widget set (fig. 2).

The next layer is responsible for the creation of more complex (compound) parts of the user interface, like dialog boxes for input deck assembly for the various simulation tools.

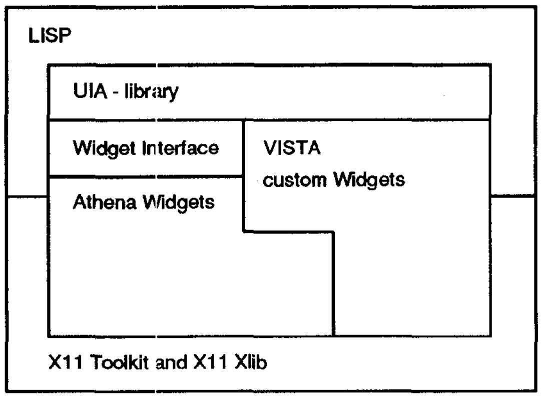

Figure 2: Structure of the User Interface

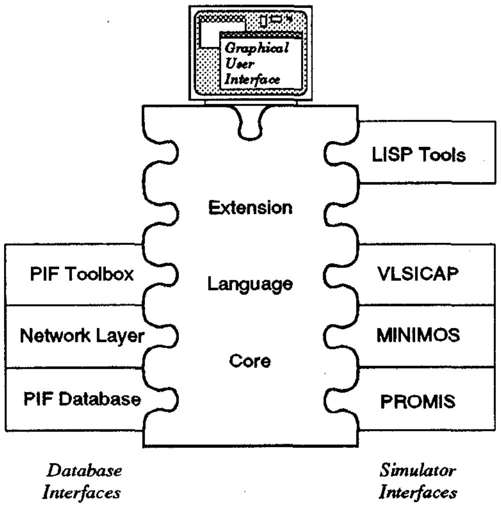

Figure 3: Extension language model

4.2 Task Level Implementation

The choice of the extension language is a crucial point. When keeping in mind the architectural criteria stated above, the right decision is quite obvious. The use of PIF – which is a subset of LISP – as the database format suggests to use LISP also for the extension and shell programming language, in accordance with the demanded minimization of different concepts.

The task level requirements are perfectly fulfilled. As an additional feature, the process flow description, simulator input or implementation of a physical model can be stored in language structures (and thus be seen as data) or executed from the command level (and thus be seen as a program).

In contradiction to C, LISP has been explicitly designed for interpretation, and a number of interpreters is publicly available. Among these we have chosen XLISP [4] to serve as the core of the VISTA extension language. XLISP offers a subset of COMMON LISP in a very compact implementation. Our extensions (which would have been neccessary for any extension language) include interfaces to the database, to the X window system and to various simulators (fig. 3). We could have chosen a more complex and powerful language (e. g. COMMON LISP or SCHEME); however, we decided for the most simple, portable and publicly available option, which is XLISP.

4.3 Callback Concept and Evaluation Loop

The X11 connection is not a feature of the original XLISP interpreter; this extension had to be introduced to implement the user interface agent in a way which is consistent with the extension language shell and the overall system architecture.

As a direct consequence, the VISTA callback concept was derived from the X11 toolkit callback method. Events coming from the user interface are passed to the XLISP interpreter. In response to any user action (like a button press), a LISP expression is evaluated. This expression is a property of the widget which issued the callback and can be used to change parameter values, trigger other events like the execution of a simulator or start the evaluation of any LISP program.

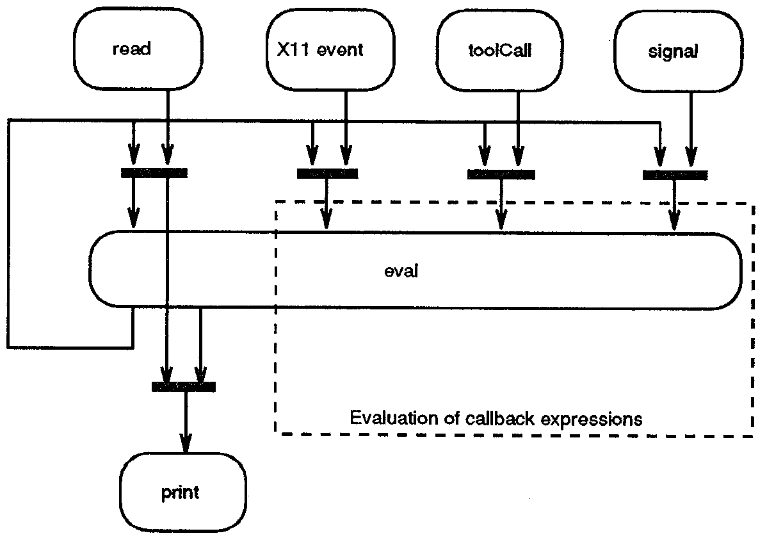

Figure 4: Petri net of the TCAD shell evaluation loop

The same callback concept is used to control the simulator execution. The termination of a simulation tool is signaled to the the parent process (cf. "signal" in fig. 4), which again causes the callback expression to be evaluated.

Callbacks can also be evaluated during the execution of a tool by explicitly calling a function which returns immediately and just triggers the evaluation of a previousely assigned callback expression via the network layer (cf. "toolCall"). This is especially useful for the online visualization of computed data.

One of the advantages of the callback concept is that the tool which triggers the callback does not have any information about the action which is taken in response to the evaluation (= call) of the callback. This enforces a very proper interface for coupling tools at the execution level.

Furthermore, the callback concept allows the TCAD shell to implicitly parallelize evaluations of callback expressions or to distribute evaluations adaptively on different hosts, without any change in the tools which requested the evaluation (like optimizers or iterators).

Another application of the callback concept is the exception handling or the handling of network events.

4.4 Tool Abstraction

Abstract descriptions of tools are used to generate the C interface code for task-level tool integration automatically. The tools can thus be easily linked with the XLISP interpreter or be included in separate executables, in either case yielding new LISP functions which can then be used in TCAD shell programs.

XLISP is used to generate the interface code, according to the software architecture demands stated above.

This frees the programmer from the bothering task of writing all the interface code manually and also greatly contributes to the correctness of the code.

5 Additional Features

5.1 User Interface Generation

The automatic generation of dialog boxes is a feature which is stimulated by the combination of the flexible LISP language with the X11 toolkit user interface. The abovementioned abstract tool description is fed into a user interface generator which produces the interaction panel for the corresponding tool. At a closer look, this is a spin-off of the consequent pursuit of the software architecture guidelines.

5.2 CASE Utilities

The symbol manipulating facilities of the LISP-based extension language make it the ideal platform for various CASE (Computer Aided Software Engineering) tools which help to ensure the operating system independence and consistency of the framework in many ways.

In the VISTA system, the XLISP interpreter is used to extract and maintain an up-to-date documentation of all system components. A hierarchically organized, interactive online documentation subsystem retrieves the requested information out of special comments within the code at the time a module is loaded, while another LISP tool generates UNIX man pages and large parts of the programmers' manual out of the same comments.

The bulk of the C language PIF database access functions are automatically generated out of a formal PIF syntax specification (written in LISP), and even parts of the VISTA extension language interpreter are generated this way (a minimum version of XLISP is required for the bootstrapping phase).

6 Conclusion

Our experience shows that the decoupling of the user interface from the actual simulators and the generic functionality due to the bottom–up design of the user interface agent is very well suited for both integrating existing simulation tools as well as implementing new ones.

The use of LISP as the basis of the extension language has proved to be a superior solution to "conventional" programming languages as well as simple command line parsers. Against common expectations for LISP based systems, only a small amount of computation time is spent in the interpreter. Since the extension language environment only controls the execution of simulation tools and handles callback-events triggered by simulators or the user interface, the overall TCAD framework performance depends mainly on the tools themselves.

Acknowledgement

This project is supported by the laboratories of: AUSTRIAN INDUSTRIES – AMS at Unterpremstätten, Austria; DIGITAL EQUIPMENT Corporation at Hudson, USA; SIEMENS Corporation at Munich, Germany; and SONY Corporation at Atsugi, Japan.

References

[1] F. Fasching *et al.*, *A PIF Implementation for TCAD Purposes*, Proc. SISDEP, Vol. 4, pp. 477–482, Zurich, Switzerland, 1991

[2] S. Selberherr *et al.*, *The Viennese TCAD System*, Proc. Int. Workshop on VLSI Process and Device Modeling, pp. 32–35, Oiso, Japan, 1991

[3] H. Pimingstorfer *et al.*, *A Technology CAD Shell*, Proc. SISDEP, Vol. 4, pp. 409–416, Zurich, Switzerland, 1991

[4] D. M. Betz, *XLISP: An Object-oriented Lisp*, Version 2.0, Peterborough, NH, Febr. 1988

[5] A. Nye and T. O'Reilly, *X Toolkit Intrinsics Programming Manual*, Vol 4, 1990, O'Reilly & Associates, Inc.