GARDEN—An integrated and evolving environment for ULSI/VLSI CAD applications

by A. H. V. de Lima
R. C. B. Martins
R. Stern
L. M. F. Carneiro

The design and specification of efficient and powerful Ultra Large Scale Integration/Very Large Scale Integration (ULSI/VLSI) computer-aided design (CAD) systems to deal with the current integrated circuit manufacturing technology is beyond the capabilities of the usual software development methodologies. This paper presents GARDEN, an integrated ULSI/VLSI design environment conceived to cope with problems in the evolution of the computing environment. It also highlights the utilization of the Vienna Development Methodology (VDM) for the specification, design, implementation, and maintenance—in short, all of the software life cycle—of this CAD system, under development at the IBM Brazil Rio Scientific Center.

The design of integrated circuits and systems is a category of complex engineering tasks that cannot be performed without the aid of sophisticated and reliable computer-aided design (CAD) systems. The complexity of Very Large Scale Integration (VLSI) circuits demands software tools to both help engineers in the design steps and ensure proper operation of circuits once they are manufactured. Since circuit design complexity doubles every two years, bearing what can be called Ultra Large Scale Integration (ULSI), there is an increasing need for more powerful and efficient design systems.

Over the past few years, CAD systems for integrated circuits evolved tremendously to cope with the increasing circuit complexity. These CAD systems were usually oriented toward the designer, addressing mostly circuit and manufacturing process problems. As expected, and as far as the designers are concerned, these systems turned out to be, in many cases, extremely efficient and user friendly.

Despite the importance of this aspect, from the development point of view most of these CAD systems present a somewhat cumbersome architecture. Normally, they have coded dependencies on the underlying hardware and operating system. Also, they sometimes rely on specialized software such as a particular database management system (DBMS) or programming language. Maintaining and improving such systems presents a permanent challenge to developers. As an example, "porting" existing applications to new hardware or a new operating system

usually requires a considerable amount of effort. In the same way, changing the database system to a more efficient or specialized one is, in most cases, virtually impossible, since it may require a complete revision of the existing code.

Being independent of factors such as hardware, operating system, DBMS, programming language, etc. is one of the most desired features for a large group of computer applications in which VLSI CAD systems are included.[1] Providing such a degree of independence tends to increase the life span of computer applications, and it turned out to be a very challenging research activity during the past several years. Although a longer life cycle for applications reduces the stress on users, it can be extremely painful to developers to cope with new and enhanced techniques that can be used to provide more efficient and reliable services.

Solving this problem can be rather complex. To be efficient, an implementation of a computer application often needs to exploit some internals of the target computing environment. Such an application is inclined not to survive for a long time without a periodic code revision. A possible approach is to insulate application-specific operations from the computing environment, a task that can be extremely complex, even for very specific applications.

To carry out such an approach, a very precise definition of the problem being addressed by the application is mandatory. This definition can be achieved with a formal software development methodology. This knowledge is then mapped into requirements that are used to define a strategy to insulate the application from the computing environment.

GARDEN[2] was conceived to be an integrated environment for ULSI/VLSI CAD applications and to cope with the problem of the evolving computing environment. The main strategy of GARDEN is the existence of very specialized interfaces that provide a layer of insulation between the CAD tools (applications) and the computing environment. Those interfaces will allow the latter to evolve without requiring changes to existing tools. It is also part of GARDEN's strategy to provide the user with a uniform and consistent view of the system and tools, aiming for user friendliness and dropping the constant and expensive need for training.

Usually, the design of large CAD systems is a fairly intricate problem. Because the design of ULSI/VLSI circuits and systems exhibits a very complex nature, CAD systems that aid the execution of such tasks represent one of the most complex problems for software engineers.

The traditional process for developing software proved to be inefficient for more complex systems since the famous *software crisis* of the 1970s.[3] One of the main features of formal software development methodologies and their derived environments is the description, in a precise and clear way, of the entities involved in the software creation process.[4,5] Therefore, these methodologies present an appropriate environment where the development of complex CAD systems can be performed in a reliable, disciplined, and safe way.

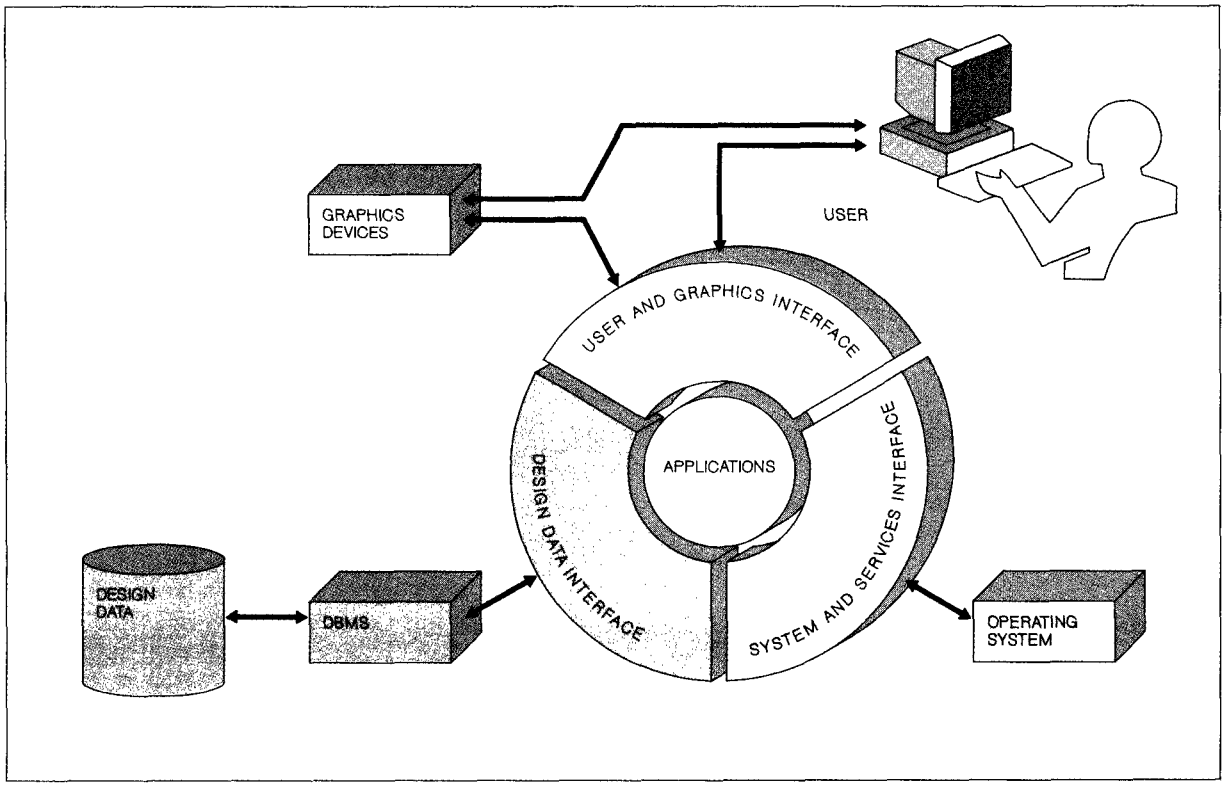
These formal methods can be classified into two large groups: property-oriented and model-oriented. The first group uses mathematical formalism to describe the *properties* that the system under design must obey. In contrast, model-oriented methodologies use well-known objects to *model* the system. An example of the latter group is the Z Methodology, from the Programming Research Group of the University of Oxford, that is being used at the IBM Laboratory in Hursley Park in the United Kingdom for redefining the Customer Information Control System (CICS).[6] Another example is the Vienna Development Methodology (VDM),[7–9] used for the specification and development of compilers for languages such as Ada® and CHILL, for a formal model of System R, and for other applications.[5] VDM was the formal methodology selected to support the development of the GARDEN project. The choice was based on the existence of some knowledge about VDM in the project team and on the availability of a reasonable number of references pertaining to it.

The evolving GARDEN ULSI/VLSI CAD environment

A general picture of the GARDEN CAD environment, including the interfaces, is presented in Figure 1. It is said to be evolving since the interfaces will enable new computing environments to be supported by the system, without the necessity of changing existing application programs (tools). The primary function of these interfaces is to provide a layer of insulation between the computing-environment-dependent features and the application programs.

Changes and enhancements applied to the interfaces are automatically extended to existing application

Figure 1 An overview of the GARDEN environment



programs, if the bindings for the interfaces do not change. Modifying these bindings is a more complex problem since it may require dropping the support for a particular binding to the detriment of a more efficient one. Such operations must be carefully carried out so that existing applications are not disturbed. A certain time must be allowed for applications to migrate to the new bindings before support for them is dropped.

GARDEN's interfaces, namely *User and Graphics*, *Design Data*, and *System and Services*, are oriented toward the necessities of integrated circuit design tools. As with general-purpose application program interfaces (APIs), the definition of GARDEN's interfaces are allowed to evolve, enabling the GARDEN environment to take advantage of enhancements provided by new or modified computing environments.

The software development methodology used in the GARDEN project (VDM) provides a control mechanism to avoid the careless adoption of features that are present in only a particular computing environment. The formal definition of the interfaces is being carried out independently of the target computing environment, and the same approach will be used for the implementation of changes and enhancements to the interfaces.

The general idea is to treat the interfaces and the application programs as independent program units. Changes applied to each individual part should not disturb the other, unless carefully documented. The only required task to be performed whenever there is a program unit change is to link-edit the entire system, and it can be done dynamically on computing environments that provide the appropriate support.

These interfaces orient GARDEN toward the designer and the computing environment and also toward the application development engineer. This characteristic of the system is very important as it frees tools developers from coding system and hardware-dependent routines, allowing them to concentrate on new tools and better algorithms. Another advantage,

embedded in this approach, is that complex tasks, such as coding operating-system-level routines, graphics drivers, DBMS calls, etc., are coded and optimized just once.

GARDEN encapsulates tools and its interfaces into a single environment capable of supporting multiple concurrent users and multiple active applications, with multitasking capability whenever the operating system provides support for it.

The GARDEN CAD environment provides more than just tool encapsulation.

However, the GARDEN CAD environment provides more than just tool encapsulation. The success of such a design environment depends also on the existence of features that are related to ULSI/VLSI circuits and systems design. To meet these requirements, GARDEN is being developed with the following characteristics as its focus:

- Built-in *design management* will provide such functions as design version control and data integrity control.
- No specific *design methodology* is enforced by the system, implying that the system does not define the design steps that must be performed. The design methodology will be enforced either by application programs or by the design team management. GARDEN provides the means for applications to define precise design methodologies and to control them.
- Full *design traceability* permits design and analysis data to be traced backwards in time to search for possible design errors.
- *Mixed mode*[10] and *hierarchical* design data representation allow tools to exploit, concurrently, the *description* and *composition* hierarchical natures of ULSI/VLSI designs.
- *Design documentation* facilities are being implemented into the Design Data interface, allowing a wide range of documentation strategies.
- *National language support* enables the environment to be adapted to languages other than English.
- Support is given to other levels of circuit design. The Design Data interface of GARDEN is being defined in a flexible way so as to support other levels of circuit design such as packaging, boards, system, etc.

Even with the use of software engineering, a system like GARDEN is not capable of supporting all of the available computing environments, mainly because of limitations imposed by the environments themselves. GARDEN development strategy is to provide support gradually to several computing environments, and initially it will comply with the following platforms:

- CPU types and operating systems: Included are the System/370 in which there is support for the Virtual Machine/Conversational Monitor System (VM/CMS) and Advanced Interactive Executive™ (AIX™, IBM's UNIX®-like operating system), the IBM 6150 (RT PC®) with support for AIX, and the PS/2® Models 70 and 80 with support for AIX.
- Graphics devices: All points addressable (APA) devices supported by the IBM implementation of the PHIGS (Programmer's Hierarchical Interactive Graphics System) API (graPHIGS®).
- Programming languages: Pascal (IBM VS Pascal), C (IBM C/370 and C/AIX), REXX (Restructured Extended Executor), and assembler when required. These programming languages are being used to code the interfaces and applications. Other languages may be added to the list if they comply with the linkage convention established, and if they do not disturb the overall run-time environment.

These platforms (both the hardware and the software) were selected on the basis of computing facilities that are available (or planned) at the IBM Brazil Rio Scientific Center. It must be noted that this list is not exhaustive and can be extended in the future to accommodate other computing environments.

The effort of GARDEN to support different computing environments parallels IBM's Systems Application Architecture™ (SAA™). Since SAA addresses a different set of computing environments and is oriented toward computing applications that share a distinct set of requirements, GARDEN will incorporate its own support for different computing environments.

The interfaces of GARDEN. The core of the GARDEN environment is its interfaces, which together with the application programs (tools) define a sophisticated ULSI/VLSI CAD system. As mentioned before, the purpose of the interfaces is to insulate the tools from the computing environment and to provide a consistent set of functions to help CAD application development engineers in coding efficient and reliable tools. GARDEN encapsulates tools and interfaces into a single environment capable of supporting multiple concurrent users and multiple active applications, with multitasking capability whenever the operating system provides support for it.

The interfaces consist of large sets of functions that perform actions under the control of tools and provide a uniform work environment to users. They are being organized into application development toolkits, and, for optimization reasons, different functions will perform the same basic task. This repetition is necessary because the requirements for different tools may differ, and whenever possible, GARDEN will provide an optimal function to satisfy such requirements.

The decision to divide the environment into three different interfaces (User and Graphics, Design Data, and System and Services) was based on the nature of CAD systems and the characteristics of the computing environments. Graphics devices, a mandatory item for CAD systems, are somewhat independent of the operating system and the data organization, but they are very closely related to the users. In the same way, data organization can be treated independently of operating system tasks.

Each of the interfaces is described below, with the Design Data interface having a more detailed description because it is currently in a more advanced stage of development.

User and Graphics interface. The purpose of the User and Graphics interface is to provide a set of functions for efficient and friendly communication with the users. The underlying idea is that application programs should not be aware, for example, of how to communicate with the workstation hardware, nor be concerned whether a given input device, such as a mouse or tablet, is available. Also, if the user has selected a command from a pop-up menu or typed it in a command line, it must be totally transparent to the application program.

To perform these operations, the User and Graphics interface contains input and output functions for both text and graphics, and specialized functions to handle interactive input such as selecting commands, with or without parameters, design objects, and areas on the display screen.

Under control of the interface, real interactions with the display hardware (terminals or workstations) will be performed using the PHIGS API that provides a vast, comprehensive, and powerful set of graphics functions and works with many display devices. Despite its power, this API was defined for general-purpose graphics applications and does not exhibit any knowledge about the objects being modeled. GARDEN's graphics functions are being defined at the circuit design level, together with the necessary high-level support for functions such as windowing, pop-up menus, command selection, etc.

Since the application programs will not interact directly with the users, the interface provides a common environment for all applications running under its control. This powerful feature will also simplify the time-consuming training steps required whenever a new application is added to the environment, since all applications will share a common set of operating procedures and commands. Also, from the tool development point of view, coding of new applications can be simplified with the experience and code taken from existing ones.

Another feature of this interface enables users to customize their own working environment. This point is very important where human factors are concerned. Studies note that it is extremely hard for the designers of CAD systems to define the best way to present information to users.[11,12] Thus, allowing the users to customize their own working environment provides greater satisfaction with the system, and the most evident result is the increase in productivity that can be obtained. Users will be able to define their preferences, and the definition will be matched by the User and Graphics interface at execution time. Also, users will be able to dynamically modify their working environment.

Customizing the environment includes graphics and command selection aspects and allows the user to select the national language that the entire environment will use (if all of the required tools provide support for it). National language support is another very important human factor that needs to be considered, and it goes beyond having a language to communicate with the users. GARDEN is being designed to provide national language support func-

tions for tools and environment commands (at the user level), for displaying proper national-language-dependent characters on the workstation screen, and for all types of messages generated by either the system or application programs.

The User and Graphics interface incorporates the window and task management functions of GARDEN.

The User and Graphics interface also incorporates the window and task management functions of GARDEN. It provides support for multiple simultaneously active tools, each one being assigned to one or more application windows. The design of GARDEN also allows multiple instances of the same application to share execution code, if the tool developer has complied with the basic structure for an application to run under the GARDEN environment. Multitasking is also possible under operating systems that incorporate such a facility. Some form of "simulated" multitasking capability is being provided for operating systems, such as VM/CMS, that do not support it, with the applicable restrictions.

System and Services interface. As far as operating system independence is concerned, the System and Services interface is the key one in the GARDEN environment. The main goal of this interface is to provide an efficient insulation layer around the operating system functions and services and yet achieve a great level of portability across a given set of operating systems. Another important aspect of this interface is that it is responsible for supporting application programs (tools) and the two other interfaces.

The difficulty regarding the specification of this type of interface is related to the fact that it is an extremely complex task to define what functions and services the tools and the other interfaces will require. Also, once such a set of requirements is known, care must be taken to ensure that it can be efficiently implemented on all of the target operating systems, either directly or indirectly. In the same way, consideration must be given for some operating systems that may have unique requirements. The design of the interface must clearly identify such situations and provide an implementation that does not interfere with the behavior of the environment under different operating systems.

An efficient method with which to attack the above-mentioned problems is to organize the requirements into groups of functions, according to some well-defined criteria, addressing both the similarity of the functions and the implementation steps. Under this approach, the interface presents a group of common operating-system functions such as date, time, elapsed or virtual time, user clocks, etc. Although most implementations of high-level programming languages provide these functions, a common binding across systems and languages is necessary to ensure portability.

The interface also includes an efficient set of disk I/O routines. These functions do exist in high-level programming languages, but they may not have the same behavior under different operating systems. The first and most obvious reason is the different file-naming conventions adopted by various operating systems. Also, some file systems may not implement all of the file formats that are present in other systems. The main goal of such routines is to provide a single way to perform disk I/O functions, insulating tools from system-dependent mechanisms and conventions.

Another important reason for having disk I/O functions is performance. Some compilers, to be compatible across operating systems, use I/O simulation routines that are available on some operating systems. Although they can provide source code compatibility, the simulated I/O routines are not as efficient as using native services. The disk I/O functions of GARDEN are coded with the use of native services from the operating system, requiring, in some cases, the use of assembler code. It should be noted that the kind of disk I/O routines that will be supported by the GARDEN environment is very simple (sequential access on a logical record basis), as most of the storage and retrieval of information will be directly provided by the Design Data interface.

Changes and enhancements applied to the environment must not disturb application programs. To handle this situation, the GARDEN environment will

be insulated from tools. Such changes will require only the affected interface modules to be recompiled and the new environment to be link-edited with the application programs at run time. Dynamic loading of executable modules, provided by some operating systems, can further improve this mechanism. If available, the functions for dynamic loading and unloading of tools are provided by the System and Services interface.

Also, this insulation, combined with the dynamic loading, permits the environment code to be shared among several users by having a single image of the code in memory. Such an approach may also be extended to frequently used tools, improving the overall performance of the system.

An important characteristic of interactive environments is the existence of a high-level interpreted language. The GARDEN environment incorporates an interface to a high-level language—REXX—allowing users to define their own set of macros and commands. Also, all of the steps for customizing the environment are performed using prologue and epilogue macros. Through this high-level language, users are able to issue commands to the interface and tools, allowing fast development of extensions to existing applications, thus making the system more usable and user friendly.

To enable all tools and the environment to communicate through a common set of commands, it is necessary to have an efficient command-parsing routine and a well-defined method for passing commands to the tools. It is accomplished by a combination of the User and Graphics and the System and Services interfaces. The user will select a command, using any of the available input methods, and the User and Graphics interface passes it to the System and Services interface.

The command is then parsed and dispatched for execution. A command may be directed to any of the interfaces or tools, and a given command can be executed by more than one application or interface. The command parser and dispatcher are defined to enable each tool to identify its commands and how they are executed.

The System and Services interface also includes functions such as hard-copy support for both printers and plotters, operating system message-handling, the triggering of specific actions based on another action, etc. The list of possible functions presented here is not exhaustive, and it will grow to accommodate requirements from tools and operating systems, including a well-defined and organized set of efficient functions.

Design Data interface. The last and, from the ULSI/VLSI point of view, most important interface, is the one that handles circuit design data. It is common to incorporate a DBMS into VLSI CAD systems, but it presents problems that were already mentioned. The Design Data interface of GARDEN is being developed for portability and provides functions that operate with integrated circuit design entities. The main strategy of this interface is to create a well-defined method for multiwrite access to design data information, addressing points such as design management, design methodology, and design consistency.

A design can be viewed at different levels of a definition hierarchy.

Nowadays, the organizational aspects of integrated circuit design data have become very complex and have recently been the object of several research works.[13-18] The control of design objects must take into consideration both their dynamic and structural natures.[19,20]

A design can be viewed at different levels of a definition hierarchy such as register transfer, logic, transistor, layout, etc. Also, it is necessary to correlate the different descriptions of the design for management purposes and to ensure that what is sent to manufacturing satisfies the functional specifications.

Another important point is that integrated circuit designs are usually described using a composition hierarchy. A hierarchical description is generally used to reduce the overall complexity of circuit design and provides several useful features to designers and tools. However, it increases the complexity of the design description by introducing extra structures in the data model.

The definition and composition hierarchies can be combined in a *mixed-mode* hierarchical model. Thus, a design can be described as a composition of other designs, each of which is not necessarily described at the same definition level. For example, a microprocessor may have its data path defined at the register transfer level, whereas its arithmetic logic unit (ALU) is defined at the transistor level and the control logic at gate level. This form of representation is becoming widely used, especially for simulation purposes.[21-24]

The last point that needs to be considered in modeling integrated circuit design data is the temporal dynamicity of the objects. The entire description of a mixed-mode hierarchical design must be maintained across time. This approach provides design version control, allows different design alternatives, and ensures the traceability of design errors.

It is necessary to introduce some control mechanism on top of the design model to allow the prescription of the design steps that lead to a complete and correct design. These design methodologies are usually described as the sequence of synthesis and analysis tools that must be used to ensure that the circuit will operate properly after being manufactured.

The GARDEN Design Data interface is being developed to address all of the previously mentioned problems. It includes a sophisticated mixed-mode hierarchical data model that captures the temporal evolution of its objects. Instead of enforcing a particular design methodology, the interface includes mechanisms for applications or users to specify the steps to be followed, along with how to control the consistency of a design among its several representations at different definition levels.

The interface also includes other important features such as support for design documentation and the insulation of design data aspects that are particular to a set of tools. For design documentation, the idea is to support different strategies. The data model treats documentation as an object that can optionally be attached to other objects, but the definition of how the design will be documented is under the control of the design methodology.

Design Data organization—The GARDEN Design Data interface provides a data organization structure that is geared toward the control of the elements involved in the design of integrated circuits. This organization allows the representation of various definition levels and the development of design alternatives.

The universe of design data in GARDEN is named a repository, and it is organized into a set of design libraries and a set of processes. Each library consists of a set of designs and has a process associated with it (information related to design, verification, and manufacturing). Also, a library may include other libraries, from which designs can be referenced.

This arrangement allows designs to be hierarchically composed using pieces (other designs) from different libraries. Such flexibility imposes the need for control functions to provide the minimum degree of integrity that is mandatory in a complex design system. A library may include other libraries only if the process associated with each of the libraries allows it. Besides, a complex hierarchical library structure requires a sophisticated integrity mechanism to ensure that an invalid design will never be created, either directly or indirectly. Cross-library references are kept with a back reference that permits the creation of an efficient integrity control, as presented in Figure 2.

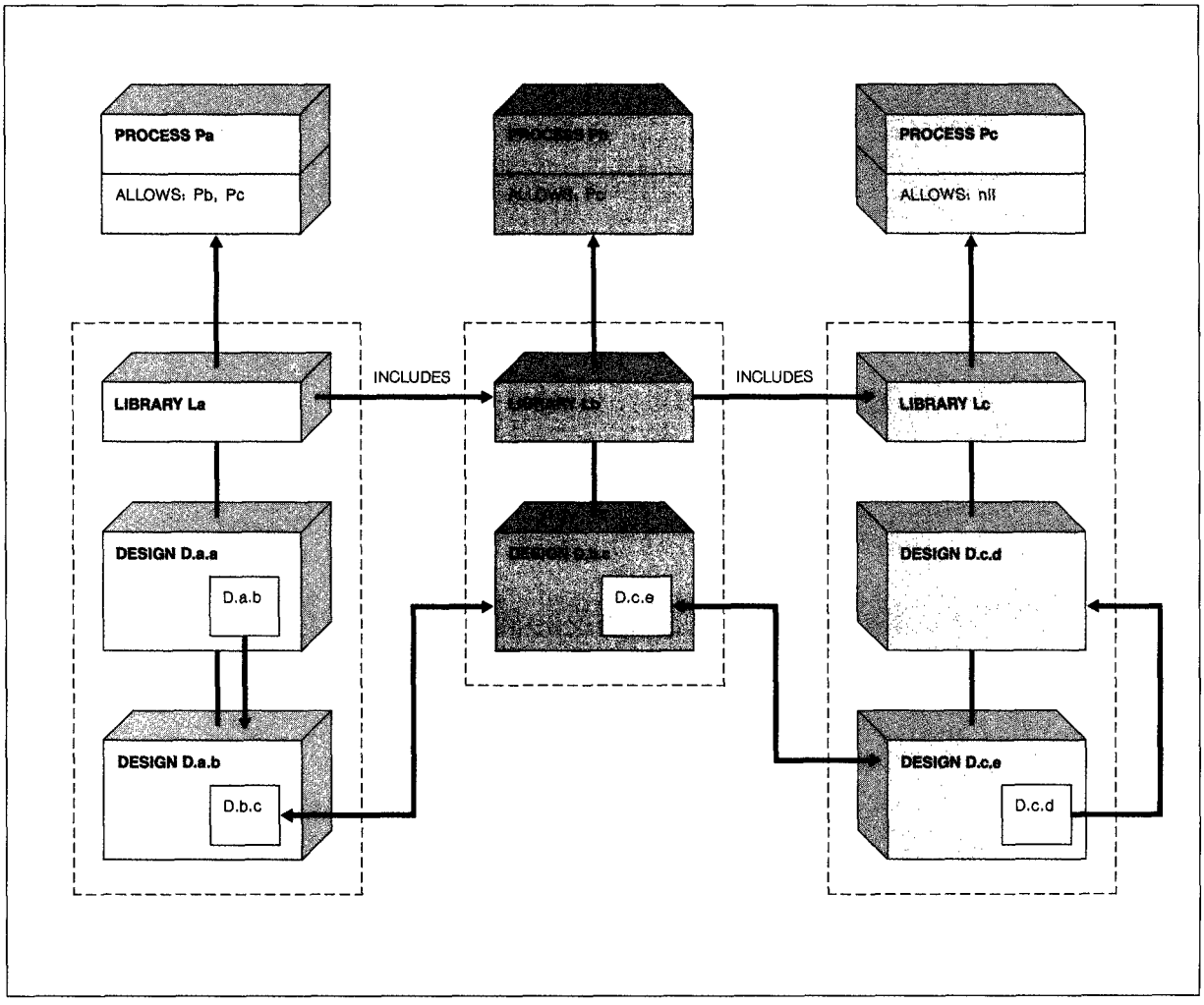
A design is a very complex object composed of several other smaller, but not yet simple, objects. Conceptually, a design is an entity that can perform a given electrical function and presents an invariant interface to the external world (external view). This interface is based on the type and number of external ports that are available in the design.

The internal representation of a design may change, giving rise to the definition of design variations: a design may have several different implementations. Each variation is associated with characteristics such as power, speed, area, etc. For example, the design of a 32-bit ALU can exhibit a low-power variation and a high-speed variation. Although they are totally different electrical circuits, these variations perform the same function and share the same external appearance (interface).

Furthermore, a variation presents different views, resulting from several definition levels that can be used to describe an integrated circuit. The views of a variation are arranged into three groups: the Layout group, the Hardware Description Language group, and the Mixed-Mode Hierarchical Description (MMHD) group (Figure 3).

The Layout View group consists of the different layouts of a variation, such as stick diagrams or mask-level information. For design integrity reasons, a variation can have only a single mask-level view. Layout views can be hierarchically formed by the

Figure 2 The cross-library reference mechanism



combination of primitive geometrical information with other information obtained by referencing layout views from a variation of other designs. Also, the electrical descriptions generated by circuit extraction tools are kept in this group, as extracted views, with a pointer to the source layout view.

Different Hardware Description Languages (HDL) can be used to define a view on a source code file basis, without any consideration of their syntax. Such an approach frees the Design Data interface from having detailed knowledge of existing hardware description languages such as EDIF, VHDL, etc. Whenever necessary, application programs can be written to compile these descriptions, named HDL source views, and the resulting object code file can be stored as an HDL object view.

HDL views and extracted views can be used for the definition of more complex designs: the Mixed-Mode Hierarchical Description. These views are hierarchically defined using references to views belonging to variations of other designs. The instances of these referenced views present instances of their external ports, which are defined at the design level of the referenced view. Port instances are connected through nets to other port instances or to the ports of the view being designed.

Temporal organization—The data organization aspects described previously do not provide any knowledge of the temporal dynamicity of the designs. Such behavior is captured by the interface at the view level and is expressed by the introduction of two time-dependent concepts: the *modification* and the *iteration*. These concepts can be depicted as a tree-like structure, as presented in Figure 4, where each node represents a *state* (version) of the view and, combined with all other organizational aspects, defines the design management mechanism that is built into GARDEN.

Iterations can be defined, using database terminology, as successive "commits" of a view. Modifications are created by the designer whenever a different design strategy is necessary. Thus, a new branch is created for every modification, and each node represents an iteration.

The concepts of modification and iteration (MI), shown in Figure 4, are applied to the objects that exist under a view. To operate in either read or write mode, the view has to be *opened* at a particular MI (state). In write mode, when a view is closed, either a new iteration or modification is created. A modification is created if the next iteration of that particular modification already exists, or by explicit request of the application program.

The implementation details of how the interface and the underlying DBMS store the MI-controlled objects

Figure 3 The different view types of a design's variation

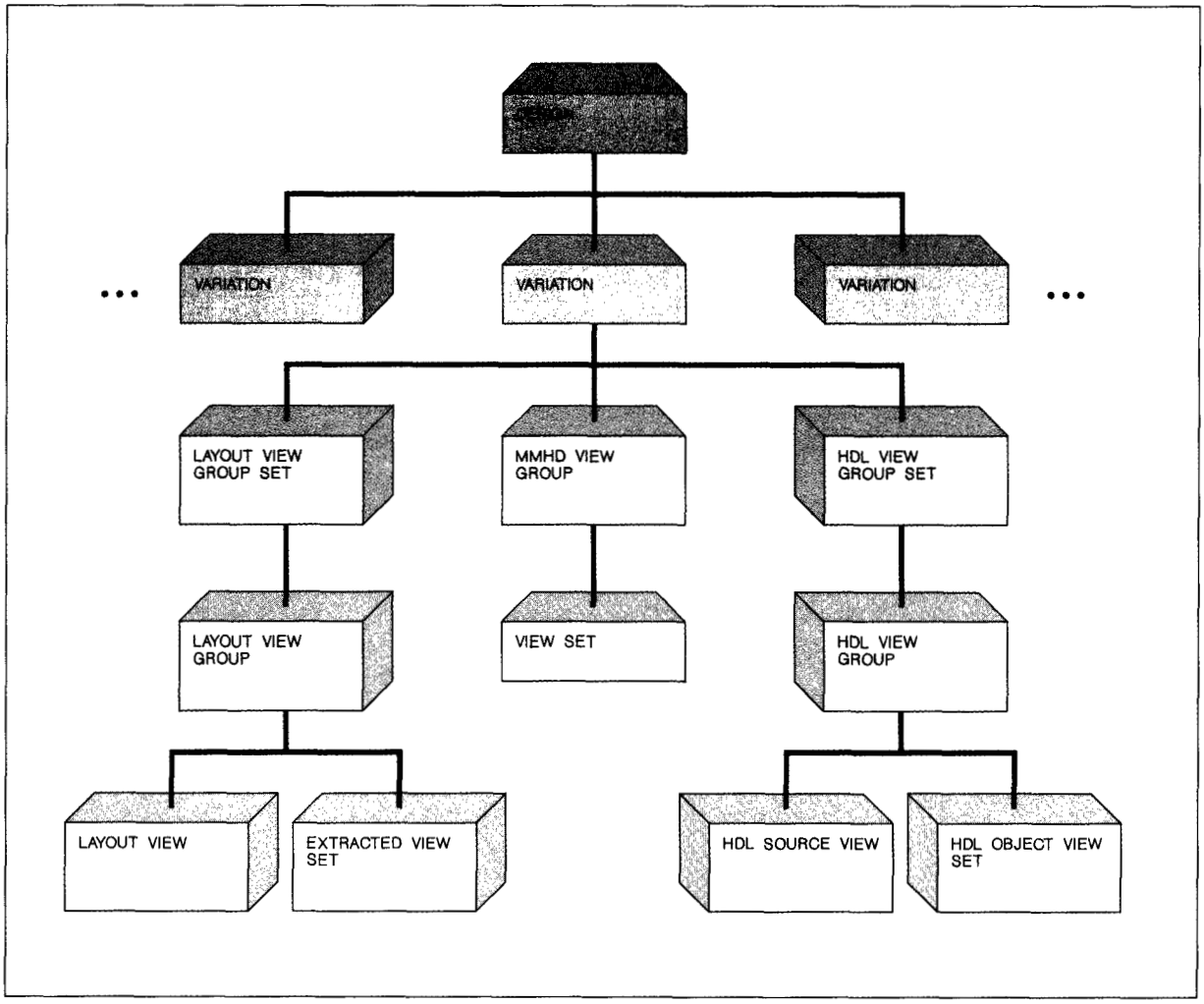
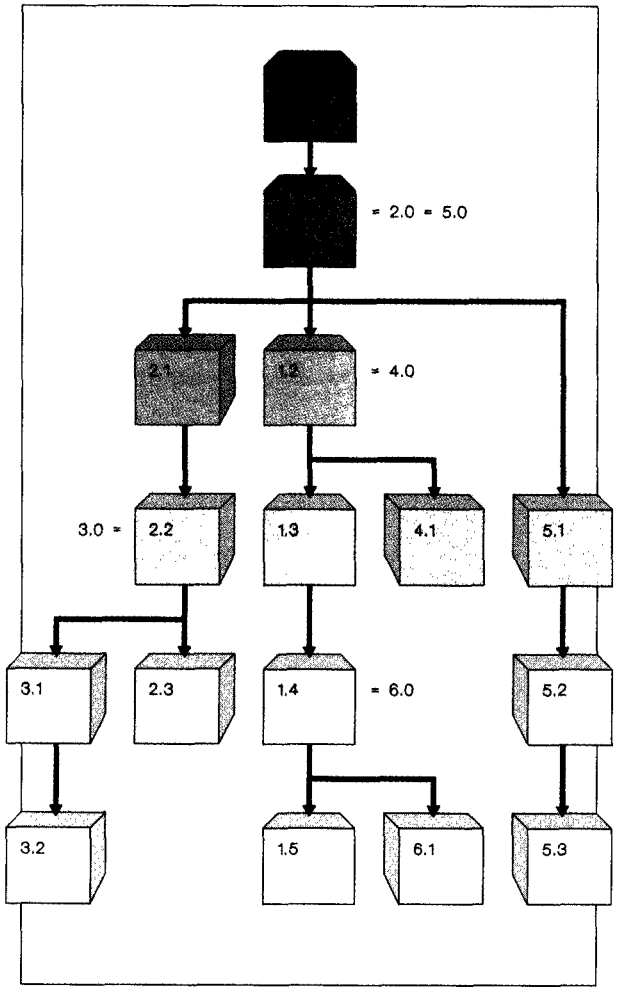


Figure 4 The tree-like structure of an MI-controlled view



are totally hidden from the tools. Once a view is opened at a given state, its components (which depend on the view type) can be retrieved independently of the temporal organization. Depending on the complexity of the objects, they will be stored using negative delta files (the latest version always exists) or by storing the creation MI and the deletion MI together with the object. (Note that it is possible to have more than one deletion MI.)

The MI-based control cannot be used for objects that are not placed under a view (for example, the ports of a design). To allow these objects to evolve and yet account for their temporal dynamicity, the interface associates different versions of these objects with a time stamp (TS). Thus, each of these objects is then viewed as a more complex object that consists of a sequence of time-stamped objects.

The interface retrieves these objects at their latest version, but application programs can also retrieve a particular version of a time-stamp-controlled object. Retrieval is accomplished by providing the interface with a time instant, and the version that was current at the specified time is then retrieved. The time for the retrieval operation can also be changed by opening a view at a particular MI. Whenever this occurs, all of the time-stamped objects are, by default, retrieved using the time information that is associated with the specified MI.

This form of organization also provides the necessary structure for allowing multiwrite access to the design data. Whenever an application program opens an object in write mode, all of the objects that are defined below it are also automatically locked. Such a locking mechanism is valid from the iteration level up to the design level. Libraries and processes are treated independently of the designs since their information does not directly affect the description of the designs.

Design Data consistency—The Design Data interface does not possess detailed understanding of the semantical contents of each view. Hence, the interface does not include the consistency mechanisms that require such knowledge. This is not a limitation of the interface, but, rather, it is a feature that enables the GARDEN environment not to enforce a particular design methodology or the use of a specific hardware description language.

Instead of having built-in mechanisms to control the consistency of the designs, the interface provides the means for the development of application programs to perform such tasks, based on a particular design methodology and on the use of certain hardware description languages. This approach allows the definition of different design environments and is capable of dealing with different or new hardware definition languages and design methodologies.

The definition of built-in consistency checks may also be complicated if a design contains several variations that are further organized into views which are, in turn, controlled using modifications and iterations. A consistency mechanism to comply with all of these possibilities will certainly impose many restrictions.

The idea is to provide a very flexible model for the design data, without disregarding the consistency aspects. Therefore, the interface allows a view, at a given MI, to be correlated with other views (the MI of each view being correlated does not necessarily need to be the same). This mechanism can be used by application programs to group together view states that are consistent among them. This information can then be used by other application programs to ensure that the design satisfies the requirements of a particular design methodology.

This grouping operation can be performed at the view level, at the variation level, or at the design level. Such mechanisms, combined with the appropriate application programs and the ability of the GARDEN environment to trigger specific actions, permit a precise definition of design methodologies that include, as a consequence, the necessary consistency checks.

Design traceability—Another feature of the Design Data interface of GARDEN is the provision for handling the input and output files of design verification and analysis operations. These files are stored in association with the view state to which they correspond, allowing the entire design process to be traced backwards in time. A design verification or analysis operation corresponds to an object that contains references to the input files, the view state that was used, the output files, and the identification of the tool. To provide flexibility and control, each view has its own set of verification and analysis operations.

A single input file can be used by more than one operation, but the output ones are specific to a particular operation. Output files are grouped into output data sets that are also placed under view. Since an input file can be used in more than one operation, input data sets are also available under design and variation. Another characteristic of the input files is their temporal dynamicity. As with other design objects, they are maintained using time stamps but without being associated with a particular view state.

Some tools use the output of other tools as one of their inputs. To accommodate this situation, an input file can be initially defined as a reference to an output file. Also, existing input files can be used as the base for new ones. This use allows the definition of general-purpose input files at the design or variation levels, and these files can be further modified for use at the view level.

Design documentation—The capability of tracing design and operations data backward in time is not the only aspect in GARDEN related to the documentation of the design process. It is often desirable to store textual documents together with the design objects. Textual design documentation can be any form of human-readable information, such as a brief description, memos, or even complex documents that are the input for text-processing programs.

The problem associated with having textual documents in the data model is where to place them. A particular design methodology may specify that all documentation must be placed at the design level, at the same time that another methodology states that it is to be kept at the view level. The solution adopted in GARDEN is to place a documentation object, named *written information*, at all places where it may be necessary. Thus, objects such as libraries, processes, designs, variations, views, operation data, etc. can all have documentation associated with them, allowing the design methodology to define whatever documentation strategy may be required.

The interface does not account for modifications applied to simple textual documentation such as descriptions or memos. Whenever they are modified, the new value will replace the existing one. Conversely, the interface treats more complex textual documentation objects by using the time-stamp mechanism, and the retrieval procedure is the same as for other objects (except input files).

Private and public repositories. The use of the Design Data interface of GARDEN, with all of the features described in this paper, can result in complex and enormous data repositories. To minimize this problem, the interface is being developed to support two types of repositories: the public repository and the private logical repository. The public repository is unique, but each user can have a private logical repository. The private repository includes logical in its name because it does not contain private copies of the objects. Instead, only the updates are stored in the private repository. The resulting object is the combination of the data from the public repository with what exists in the private one.

Whenever a design object is opened in write mode, it is *logically* transferred to the user's private repository. The object can then be closed and opened again as many times as necessary before the *real* close transfers it back to the public repository. The public repository treats the object as being locked until it is transferred back from the private logical one.

This locking scheme does not present the usual burden of the checkout process that exists on conventional database systems, where a lock can stay forever. Although this situation holds true for some objects in GARDEN, most of the time the object being locked will be a state of a view. More than one user is capable of opening a view state in write mode if the migrations from the private logical repositories are performed sequentially. The first user will, whenever possible, create a new iteration of the same modification. The next transfer operation will necessarily create new modifications.

The locking scheme does not present the usual burden of the checkout process that exists on conventional database systems.

While a view state is in the private logical repository, its iterations and modifications are visible only to the owner of the private repository. The user (or the application program) may decide to treat all iterations as being just one. Hence, only the last iteration is transferred back to the public repository. This operation is carried out to provide the same result as a simple open–close that does not use a private repository.

For other data objects, only one user can have write access to them, since modifications applied to these objects can affect other objects in the repository. This is true for objects such as designs and variations that are composed by other complex objects. In these situations, an object may stay locked forever, as in some conventional database systems. The interface provides functions to determine the ownership of an object lock and privileged commands to cancel it. For these objects, the interface also allows the elimination of intermediate changes, just as for the view states.

Such compression mechanisms reduce the amount of data that are kept in the real repository, since some intermediate alterations can be automatically disregarded. Because this process may be used for any design object in GARDEN, the overall result is a reduction in the size and complexity of the real repository. It is also important to mention that the Design Data interface incorporates privileged functions for repository maintenance, such as placing part of the data off line (on tape for example), as an alternative to reducing the storage requirements.

Applications. As mentioned previously, several features of the GARDEN environment rely on the development of application programs or tools. GARDEN applications are much more than just tools such as graphical editors, simulators, checkers, etc. An application can be virtually any computer program that interacts with the user to perform some action on the design data.

The interfaces provide only the framework for the development of a ULSI/VLSI design system that can be adapted to a particular design environment using specific application programs. Among all of the applications that can be developed for this purpose, it is important to mention those that are necessary to enforce a particular design methodology or to ensure consistency of the design according to given criteria.

However, GARDEN is not just its interfaces and some application programs for customizing the environment. Design tools are, by far, the most important application programs that can be developed. One of the main ideas behind GARDEN is to support the development of efficient and reliable design tools, leaving considerations about the computing environment for the interfaces.

Therefore, the GARDEN project will include the development of design tools such as graphical editors, simulators, timing analyzers, etc. Also, the Design Data interface is being defined to provide support for other levels of electrical circuit design such as integrated-circuit packaging, boards, etc. These features, and the mixed-mode hierarchical representation will allow tools to combine, in the same design, circuits defined at different design levels. Another important concern of the GARDEN project is to use a formal methodology for the specification and development of both the interfaces and application programs.

The GARDEN software development process

Providing all of the previously described functions in a single integrated and evolving environment is

not a simple task. To achieve its goals the GARDEN CAD environment is being developed by using *software engineering techniques*. This approach is necessary to help ensure the operation of the system and to ease coding steps, because conceptual errors are being identified and corrected during specification steps. GARDEN is intended to be a long-term project as far as development and support are concerned, and it is being performed according to the Vienna Development Methodology (VDM).

As with any specification technique based on models, formal or not, VDM relies on the definition of explicit models of the system under specification, using either abstract or concrete concepts. Figure 5 presents the overall development process of VDM. The complex mathematical foundations of VDM are beyond the scope of this work.[9,25] However, this methodology can be briefly summarized by the following list of actions:

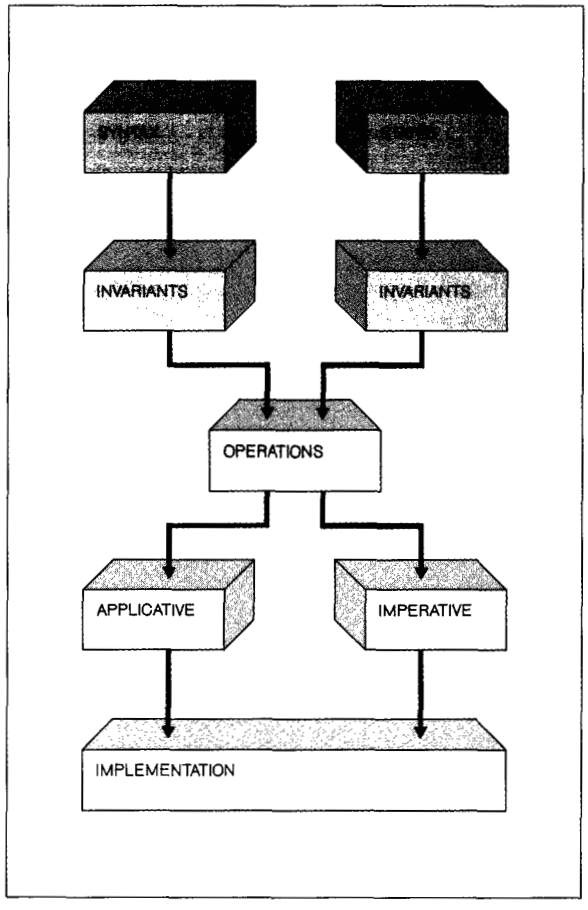
- Definition of states and semantic domains
- Construction of invariants for the semantic domains
- Definition of abstract syntax and syntactic domains
- Construction of well-formedness constraints for the syntactic objects
- Specification of operations and functions for the system under definition

The first two items of the list define the model of the system. The remaining ones characterize the interactions that can occur with the model. The last point of the methodology is that operations and functions can be described either in an applicative style (functional languages) or in an imperative style (procedural languages). The implementation steps are performed using successive refinements of the definition of functions and operations.

In order to describe VDM concepts, some objects of the GARDEN CAD environment will be used. Note that the description given in this work is not the entire specification of the GARDEN CAD system, but just a small subset to enable a concise description of the approach used.

States and semantic domains. States and semantic domains are defined from well-known mathematical objects.[26] The semantical understanding of the model being constructed is derived from the semantics of the mathematical expressions that are used. To build such a model, one may use mathematical

Figure 5 The VDM process



objects such as sets, sequences, Cartesian products, mappings, and predefined objects.

Predefined objects are usually Boolean values, integer numbers, natural numbers, rational numbers, denoted by $\mathbb{B}$, $\mathbb{Z}$, $\mathbb{N}$, and $\mathbb{Q}$, and sets obtained by enumeration of atomic symbols. All the usual operations that can be applied to the above-mentioned constructs can be freely used in the specification to define new and more complex objects. It is common to use operations such as union, intersection, discriminated union of sets, the head or the tail of a sequence, the range or the domain of a mapping, the Cartesian product of domains, etc.

As an example of these constructs and operations, a **Repository**, the name for the universe of all design **Libraries** and **Processes** in GARDEN, can be represented as a Cartesian product of three mappings, one

of Libraries, one of Processes, and another one of Documents as shown below.

Repository :: Lib × Process × Document

To name a particular element that belongs to one of these sets, the following expression can be used:

is-Lib(l_i)
is-Process(p_i)

where $1 \leq i \leq n \wedge n \in \mathbb{N}$

GARDEN captures the temporal description of its objects. For example, design documentation may evolve, and it is necessary to create a structure that captures such a mechanism. Documents in GARDEN are expressed as sequences of time-stamped names, time-stamped delta files, and other constructs. Using VDM notation, this is written as:

Document :: seq of TSName × seq of TSDelta × ...

To define some structures that are more complex, it is necessary to use more elaborate mathematical constructs than just sets and sequences. For example, a Library can be expressed as a mapping between *identifiers* and *fields*. It is further defined as the Cartesian product between a Name, a set of other Libraries (that might be included by this one), a Document, the Process related with this library and a set of Designs. A bonus from using this type of formalism is the possibility of ensuring the Identifiers are unique.

Lib = Id $\mapsto$ Fields1
Fields1 :: Name × Lib × Document × Process
× Design

Also Process can be defined using this technique.

Process = Id $\mapsto$ Fields2
Fields2 :: Name × ProcessInfo × Document
× Process

Invariants. It is necessary to add some restrictions, called invariant rules, to the previous definitions of the semantic domains. These restrictions are written using a predicate logic language, enriched with some symbols to simplify its understanding. Typical symbols that can be used are the $\neg$ (negation), $\wedge$ (conjunction), $\Rightarrow$ (implication), and $\Leftrightarrow$ (logical equivalence).

To the usual set of logical symbols, the let and in tokens are added to improve the overall readability of expressions such as:

let $l_i \in$ Lib$_j$ in ...

Thus, invariants are restrictions to the values that objects under definition can assume, and a single object can present more than one invariant. Also, invariants can evolve during the specification steps of a project. As an example of an invariant, a property of the GARDEN Libraries can be written as:

inv-Lib $\triangleq$ let make-Fields1(n_i, l_i, doc$_i$, p_i, d_i)
= Lib(id$_i$) $\wedge$
let make-Fields1(n_j, l_j, doc$_j$, p_j, d_j)
= Lib(id$_j$)
in
(id$_i \neq$ id$_j \Rightarrow n_i \neq n_j$) $\wedge$
$\neg(p_i = \emptyset) \wedge \cdots$

It denotes that the names within a Lib are unique and there must exist a Process associated with a library. The creation of invariants relies on a detailed knowledge of the objects that are being modeled. The specifications of these rules create a part of the so-called *proof obligations.*[25]

Abstract syntax and syntactic domains. A model must be dynamic to describe the real world, and this aspect can be carried out with a set of operations that modifies the states. An abstract syntax allows an understanding of the syntactic domains involved in the operations, without considering the implementation details. Because of these facts, another important step of VDM that can be performed in parallel to the definition of the states is how to operate and modify states.

On the basis of previous examples, it is clear that it will be necessary to provide functions to list names of Libraries and names of Processes. Using an abstract syntax, this can be denoted by:

List = List-Libraries | List-Processes | ...
List-Libraries = Name*
List-Processes = Name*

As with semantic domains, the syntactic domains are described in a very flexible form. Hence, rules for well-formed constraints must also be added to the specification. These constraints express the syntax properties that the operations must obey.

As an example, the well-formed constraint for the **List-Libraries** is expressed as follows:

well-form-**List-Libraries**(l) $\triangleq$ $l = \emptyset \vee \forall\, x,$
$(x \in l \Leftrightarrow x \in$ Lib)

Operations and functions. Once the syntactic and semantic domains are defined, it is necessary to establish what the functions and operations will perform. This step is normally accomplished using well-specified high-level definition languages. In these languages, functions and operations are expressed in either an applicative (functional) or imperative (procedural) style. There is no preference for either style since they both can precisely express the syntax and semantics of the function being specified.

In the applicative style, the entire specification will resemble, as the implementation evolves, a program written using a functional language. For example, the **List-Libraries** function is defined by:

List-Libraries(l) $\triangleq$ if $l = \emptyset$
then $\emptyset$
else let x = (Id, Fields1) $\in l$ in
{Name.Fields1} $\cup$
List-Libraries($l - x$)

type: **List-Libraries** : Lib $\rightarrow$ Name*

Conversely, the imperative style implies the use of pre and post conditions to characterize the function under definition. Usually, these conditions are expressed as:

f: Input $\xrightarrow{m}$ Output
let $o = f(i)$ in $\forall\, i$ (($i \in$ Input $\wedge$ pre-$f(i)$)
$\rightarrow$ ($f(i) \wedge$ post-$f(i, o)$))

The same function, **List-Libraries**, expressed previously in the applicative style, can be rewritten using the imperative style:

List-Libraries(l) =
x :: = range (l)
while $x \neq \emptyset$ do
let $f \in x$ in
output(Name.f)
x :: = $x - f$
end

type: **List-Libraries** : Lib $\rightarrow$ Name*

To characterize this function the following conditions must be added:

pre-**List-Libraries** $\triangleq$ is-Lib(l)

post-**List-Libraries** $\triangleq$ let r = **List-Libraries**(l) in
r = Name.range(l)

As presented previously, GARDEN incorporates some form of cross-library integrity control, based on the characteristics of the process associated with each Library. Each Process has a list of every other Process considered to be *compatible* with it: a Library may include another Library only if the Process of the included Library is compatible with the one to include it (note that *compatibility* is a unidirectional property). A library is capable of accessing other libraries as an extension of itself. This access operation is carried out recursively until a Library with an *incompatible* Process is found or all referenced Libraries are verified. Thus, the **AccessLib** function, which determines what other libraries a given Library can access, is defined by:

AccessLibAux $\triangleq$ {((x,p), **AccessLibAux**(x,p)) |
let x = (Id, make-Fields1(n_i, l_i, doc$_i$,
p_i, d_i)) in $\forall\, y$, $\forall\, z$ ($y \in l_i \wedge$
$z \in$ **AccessLibAux**(x,p) if
$z \in l_i \vee$
(($z \in$ **AccessLibAux**(y,p)) $\wedge$
((p = Process.range(z)) $\vee$
(Process.range(z) $\in$
Process.range(p))))}

type: **AccessLib** : Fields1 $\rightarrow$ Lib

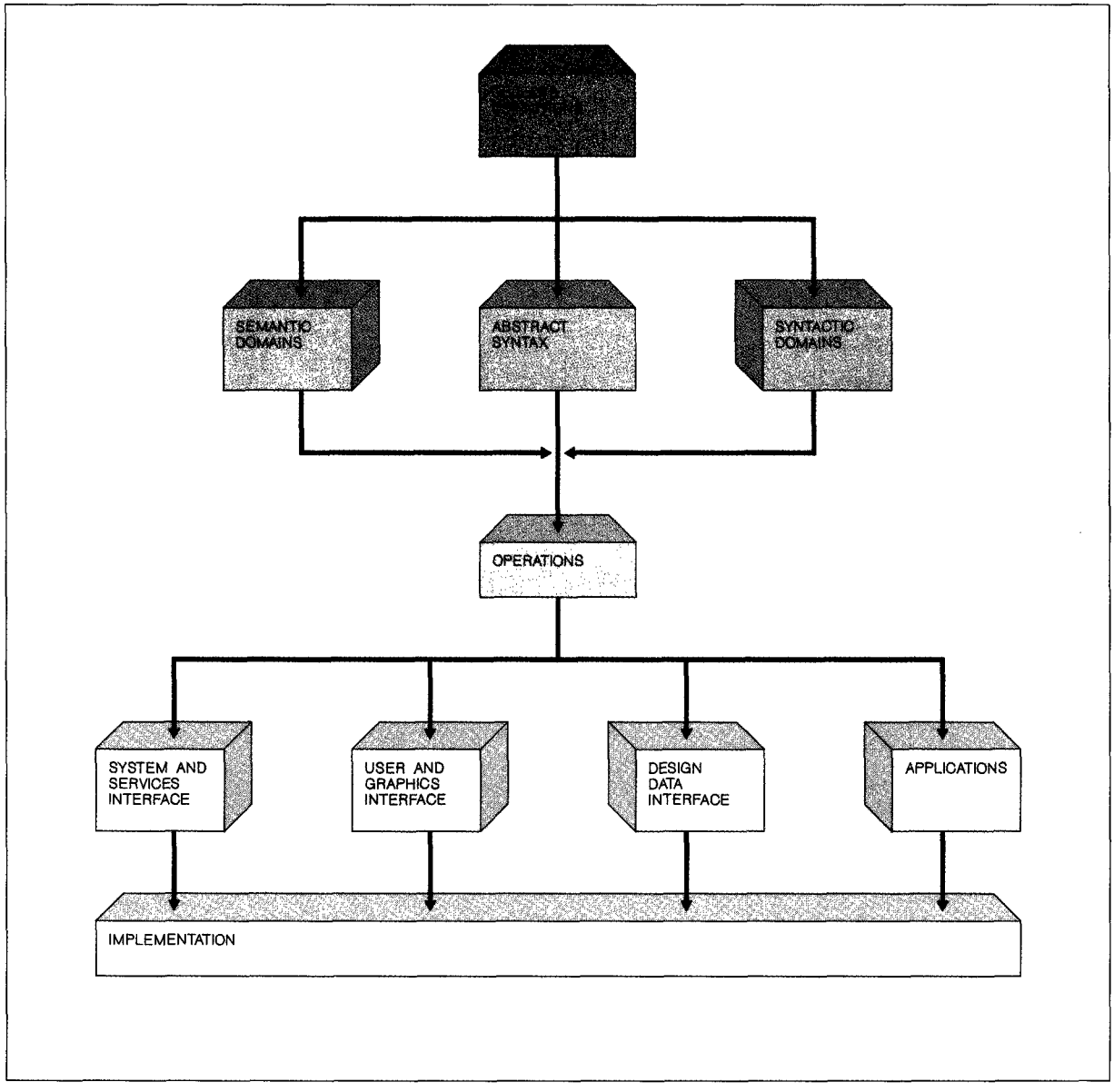
AccessLibAux $\triangleq$ {((x,p), **AccessLibAux**(x,p)) |
let x = (Id, make-Fields1(n_i, l_i, doc$_i$,
p_i, d_i)) in $\forall\, y$, $\forall\, z$ ($y \in l_i \wedge$
$z \in$ **AccessLibAux**(x,p) if
$z \in l_i \vee$
(($z \in$ **AccessLibAux**(y,p)) $\wedge$
((p = Process.range(z)) $\vee$
(Process.range(z) $\in$
Process.range(p))))}

type: **AccessLibAux** : Lib $\times$ Process $\rightarrow$ Lib

The auxiliary function **AccessLibAux** is necessary to prevent an endless recursion when a Process in the chain is not compatible with the one of the starting Library. Thus, the Process of the starting Library is passed as a parameter to the auxiliary function **AccessLibAux.**

Another feature, present in definition languages, are mechanisms for successive refinements, from high-level definitions up to implementation details. This

Figure 6 GARDEN software development graph



process of refinement, also called reification, will create another part of the *proof obligations.*

A model for software development. Applying formal techniques in software development does not preclude the use of a development life-cycle model. In fact, it is also possible to adapt a suitable life-cycle model to a more formal scheme like the *software development graph.*[9] Basically, this scheme consists of three types of graphs:

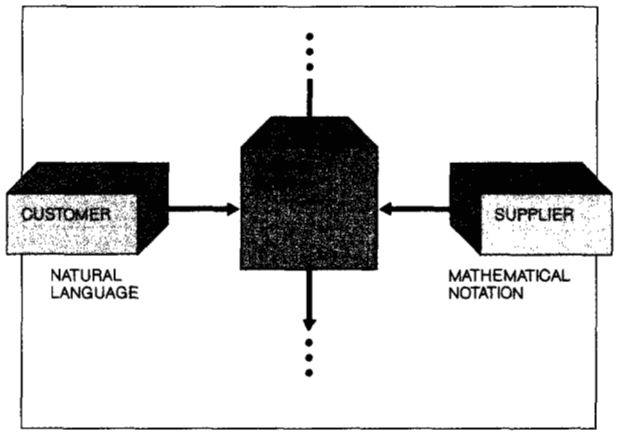
- *Meta-graph,* applied to the development of a given class of software (e.g., operating system, CAD)
- *Project graph,* a meta-graph which includes details of a given instance (e.g., operating system *x*, CAD for ULSI/VLSI)

- *Configuration graph*, a graph for a specific configuration of a given project graph (e.g., operating system x for target machine y, operating system x for target machine z)

Figure 6 is a summarization of the development of the GARDEN project. The objects definitions node corresponds to the *requirements* phase of the traditional life-cycle model. The other three levels in the diagram correspond to the *design* phase. The last level, implementation, corresponds to the *coding* phase. The other phases, *testing*, *use*, and *maintenance*, are not presented in the diagram.

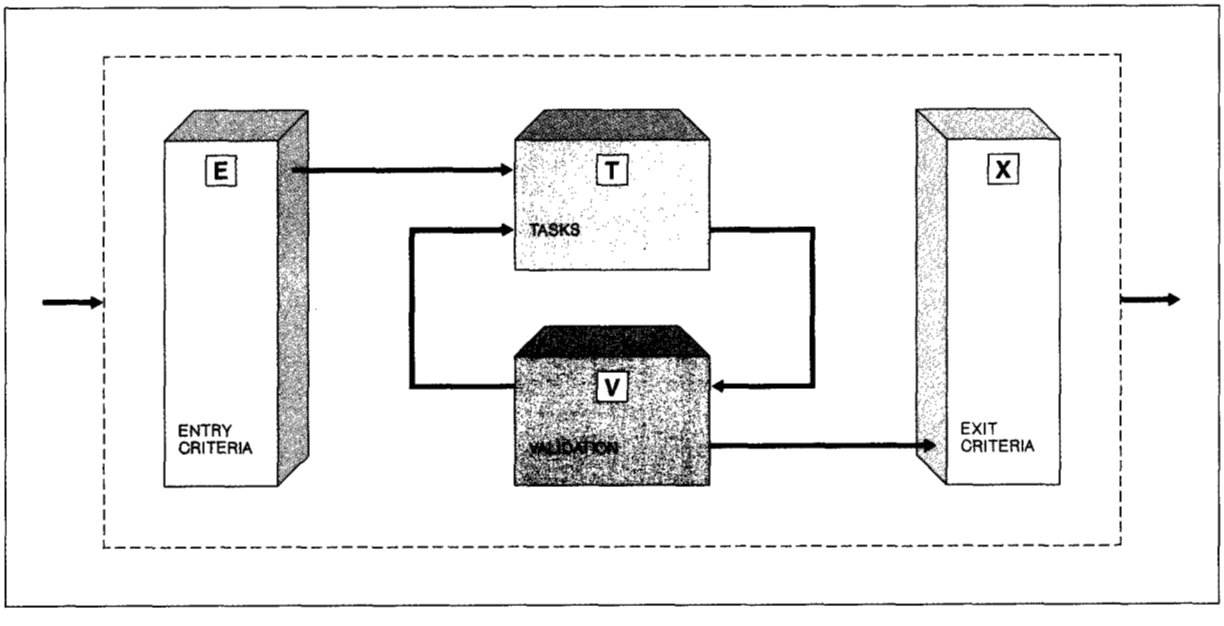
The contractual model of software development[5] is associated with every node in the development graph, as shown in Figure 7. The developers (suppliers) and the users (customers) must interact until a complete agreement is reached, as far as the function of each node is concerned. In this model, the users express their requirements in natural language, whereas the developers use mathematical notation for the semantics of the system under definition. This agreement is formalized into a document which is the actual contract of that node, and is performed for every node in the graph. For the GARDEN project some members of the development team, which possess expertise in VLSI design, act as users, while the role of the supplier is performed by the remaining members.

Figure 7 Contractual model for software development



For the developer, the node takes the form of an extension of the ETVX model,[27] applying it to the program architecture process and to the entire project graph. This model, presented in Figure 8, requires that for every node of the graph the following items are to be defined: an entry criteria, an exit criteria, tasks to be executed, and a validation criteria. The formal structure of VDM eases the definition of these items.

Figure 8 The ETVX paradigm



Concluding remarks

Currently, GARDEN's User and Graphics and Design Data interfaces are under specification. Simultaneously, parts of the System and Services interface are being implemented (for early support to tools and the other interfaces), while its formal specification is being performed. Also, the first prototype versions of CAD tools such as a timing analyzer[28] and a simulator are being completed.

The use of formal software development methodologies in the GARDEN project is providing an important mechanism for the elimination of complex conceptual doubts and errors during the specification phase. Besides, the clarity and precision of the specification language make it a simple and efficient vehicle for the dissemination of project documentation. Also, all aspects of the system are captured by the formalism of the specification methodology. Project management is then simplified, especially in dealing with members that join or leave the project team.

Acknowledgments

The authors wish to express their gratitude to G. O. Annarumma and P. Molinari Neto, members of the GARDEN project team, and especially to Javad Khakbaz. Several of the ideas presented in this paper are drawn from the results of their collaboration.

Cited references and notes

1. John Andrews, "CAD software users address database problems," *VLSI Systems Design* **7,** No. 9, 58–63 (September 1986).
2. The GARDEN project started in 1987 as a small project to support only the design of gate array integrated circuits, and the project name was originally an acronym for Gate ARray Design ENvironment. During specification phases, the scope of the project was enlarged, but the original name was not changed for historical reasons.
3. Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Book Co., Inc., New York (1982).
4. Narain Gehani and Andrew D. McGettrick, Editors, *Software Specification Techniques*, Addison-Wesley Publishing Co., Wokingham, England (1986).
5. Bernard Cohen, William T. Harwood, and Mel I. Jackson, *The Specification of Complex Systems*, Addison-Wesley Publishing Co., Wokingham, England (1986).
6. B. M. Yelavich, "Customer Information Control System—An evolving system facility," *IBM Systems Journal* **24,** Nos. 3/4, 264–278 (1985).
7. Dines Bjøner and Cliff B. Jones, Editors, *The Vienna Development Method: The Meta Language*, Volume 61 in *Lecture Notes in Computer Science*, Springer-Verlag, Berlin (1978).
8. Dines Bjøner and Cliff B. Jones, Editors, *Formal Specification and Software Development*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1982).
9. Dines Bjøner, *Software Architectures and Programming Systems Design*, to be published (1989).
10. Some authors use the term mixed mode to refer to chips that contain both analog and digital parts. This term is also used in GARDEN to refer to circuits that have parts described in different levels of a description hierarchy.
11. Robert W. Bailey, *Human Performance Engineering: A Guide for System Designers*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1982).
12. Francine S. Frome, "Improving color CAD systems for users: Some suggestions from human factors studies," *IEEE Design & Test*, 18–27 (February 1984).
13. Stanley B. Zdonik, "Object management systems for design environments," *IEEE Database Engineering* **8,** No. 4, 23–30 (April 1985).
14. Randy H. Katz, M. Anwarrudin, and E. Change, "A version server for computer-aided design data," *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, Las Vegas (June 1986), pp. 27–33.
15. Christian Jullien, Andre Leblond, and Jacques Lecourvoisier, "A database interface for an integrated CAD system," *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, Las Vegas (June 1986), pp. 760–767.
16. Shlomo Wiess et al., "DOSS: A storage system for design data," *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, Las Vegas (June 1986), pp. 41–47.
17. Rajiv Bhateja and Randy H. Katz, "VALKYRIE: A validation subsystem of a version server for computer-aided design data," *Proceedings of the 24th ACM/IEEE Design Automation Conference*, Miami (June 1987), pp. 321–327.
18. M. A. Breuer et al., "Cbase 1.0: A CAD database for VLSI circuits using object oriented technology," *Proceedings of the 1988 ACM/IEEE International Conference on Computer Aided Design*, Santa Clara, CA (November 1988), pp. 392–395.
19. Don S. Batory and Won Kim, "Modeling concepts for VLSI CAD objects," *ACM Transactions on Database Systems* **10,** No. 3, 322–346 (September 1985).
20. D. Rieu and G. T. Nguyen, "Semantics of CAD objects for generalized databases," *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, Las Vegas (June 1986), pp. 34–40.
21. Mahesh H. Doshi, Roderick B. Sullivan, and Donald M. Schuler, "THEMIS logic simulator—A mixed mode, multilevel, hierarchical, interactive digital circuit simulator," *Proceedings of the 21st ACM/IEEE Design Automation Conference*, Albuquerque (June 1984), pp. 24–31.
22. Joel J. Grodstein and Tony M. Carter, "SISYPHUS–An environment for simulation," *Proceedings of the 1987 ACM/IEEE International Conference on Computer Aided Design*, Santa Clara, CA (November 1987), pp. 400–403.
23. Daniel G. Saab et al., "CHAMP: Concurrent Hierarchical And Multilevel Program for simulation of VLSI circuits," *Proceedings of the 1988 ACM/IEEE International Conference on Computer Aided Design*, Santa Clara, CA (November 1988), pp. 246–249.
24. Geoffrey Sampson, "An ideal mixed-mode simulator," *VLSI*

Systems Design **9,** No. 11, 70–78 (November 1988).
25. Cliff B. Jones, Editor, *Systematic Software Development Using VDM*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1986).
26. Michael J. C. Gordon, *The Denotational Description of Programming Languages: An Introduction*, Springer-Verlag, New York (1979).
27. R. A. Radice et al., "A programming process architecture," *IBM Systems Journal* **24,** No. 2, 79–90 (1985).
28. G. O. Annarumma, *Timing Analysis Based on a Hierarchical and Mixed-Mode Approach*, Technical Report CCR-075, IBM Rio Scientific Center, Rio de Janeiro, Brazil (1989).

Arnaldo Hilário Viegas de Lima *IBM Brazil, Rio Scientific Center, P.O. Box 4624, CEP 20.001, Rio de Janeiro, Brazil.* Mr. Lima received his M.Sc. in electrical engineering from the University of Brasília in 1987. He received his B.Sc. in the same area from the same university in 1984. Mr. Lima joined the Rio Scientific Center as a research staff member in 1986. His current research interests include CAD environments for integrated circuit design, data modeling and layout representation for integrated circuits, integrated circuit design rules checking, graphical interfaces, and operating systems interface techniques.

Raul César B. Martins *IBM Brazil, Rio Scientific Center, P.O. Box 4624, CEP 20.001, Rio de Janeiro, Brazil.* Dr. Martins received his Ph.D. in computer science from the Pontifical Catholic University of Rio de Janeiro in 1984. He received his M.Sc. in electrical engineering from the Military Institute of Engineering in 1974 and his B.Sc. in the same area from the same institution in 1972. Dr. Martins joined IBM in 1985 as a research staff member at the IBM Brazil Software Technology Center, and he now holds the same position at the Rio Scientific Center. His current research interests include mathematical foundations of software engineering and formal methodologies for software development. Dr. Martins is the author of several technical papers presented at international conferences on software specifications and design.

Ronaldo Stern *IBM Brazil, Rio Scientific Center, P.O. Box 4624, CEP 20.001, Rio de Janeiro, Brazil.* Mr. Stern received his M.Sc. in computer science from the University of Southern California at Los Angeles in 1987. He received his B.Sc. in electrical engineering from the Pontifical Catholic University of Rio de Janeiro in 1985. Mr. Stern joined the Rio Scientific Center as a research staff member in 1988. His current research interests include CAD environments for integrated circuit design, graphical interfaces, and software engineering methodologies.

Luiza Maria F. Carneiro *IBM Brazil, Rio Scientific Center, P.O. Box 4624, CEP 20.001, Rio de Janeiro, Brazil.* Ms. Carneiro received her M.Sc. in computer science from the Pontifical Catholic University of Rio de Janeiro in 1987. She received her B.Sc. in electrical engineering from the same university in 1984. Ms. Carneiro joined the Rio Scientific Center as a research staff member in 1988. Her current research interests include CAD environments for integrated circuit design, software engineering methodologies, and object-oriented languages and systems.

Reprint Order No. G321-5376.