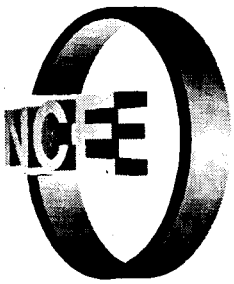

Relatório Técnico

Núcleo de Computação Eletrônica

A Distributed System for Microelectronic Algorithms

Oliveira, C. E. T.
Pais, A. P. V.
Pereira, L. A.
Parga, D. F.
Anido, M. L.

NCE - 09/2000

Universidade Federal do Rio de Janeiro

A Distributed System for Microelectronic Algorithms

Oliveira, C.E.T., Pais, A.P.V., Pereira, L.A., Parga, D.F. and Anido, M.L.

Núcleo de Computação Eletrônica - Universidade Federal do Rio de Janeiro
E-mail: carlo@ nce.ufrj.br

Abstract

Microelectronics tools tend to consume large amounts of memory and processor time. When circuit size outgrows the resources available on a station, its time for a scalable tool architecture. Applied to design rule checking of flattened masks, this distributed, object-oriented architecture summons together the power of small, cheap desktop computers. The distributed system enables processing of larger circuits, assigning distinct parts of the problem to each machine. Larger circuits can be tested and testing time reduced as more computers are aggregated to the process.

Keywords: Object oriented programming, Distributed Systems, Multithread Systems, DCOM, Microelectronic, Design Rules

Introduction

Microelectronic tools have to be prepared to work with projects that extrapolate the resources of a machine. A way to create scalable tools it is to apply techniques of distributed objects, congregating a group of machines to accomplish the task. As an example the article implements a design rule checker coupled to a mask editor. While the edition happens in the client machine, processes are distributed to other machines to verify the consistency of the published mask. This distribution is possible due to encapsulated structure of the model that supports the distribution in a three tier system.

Architecture of the Tool

This tool was projected to be integrated in a complete environment for production of integrated circuits. It possesses a modular structure[13] based on the MVC paradigm. Several desing patterns were applied to achieve a scalable architecture supporting the inclusion of new algorithm modules. The project using the separation of channels of data and control targets the interoperability among several modules.
To demonstrate the modularity of the architecture, the prototype of a VLSI layout editor[5] was implemented. In the implemented prototype two forms of visualization of an integrated circuit are presented: a graph (through rectangles) and a textual.
The system is implemented in three layers with an insulation layer between the user interface and the functional module. The isolation layer is partitioned in two main parts, one for control and another for data. The control transfers commands between the interface and the executive module. The connection of data provides a correspondence between the internal representation of the module and the visual presentation of the data.

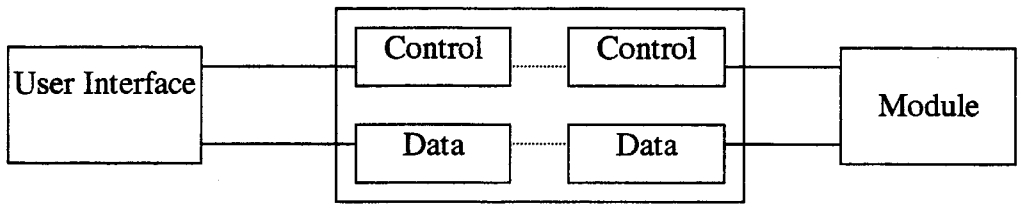

Figure 1 – Architecture of the Tool

Model of the Structure of Masks

A mask is described by a group of rectangles in several layers. To obtain acceleration of algorithms that deals with rectangles, a structure indexed to two dimensions was created. The integration of the model with the other parts of the system is obtained with the application of design patterns. This structure has an encapsulation that allows its transport and distribution in several machines.
An IC can be specified with CIF [4], a language that describes rectangles. This language structures the description of an IC in cells, layers and boxes (rectangles). A cell corresponds to a tree structure, where the cell is the root, the first level is constituted by the layers, and the second level, by the boxes. For each command of cell description, layer or box is created the respective object. Consequently, a TCell has a collection of TLayer's, and a TLayer has a collection of TBox's. A CIF file contains the description of one or more cells. To represent several cells, an object TLayout that contains each object TCell was created. The object TLayout corresponds to the complete description of the CIF file.

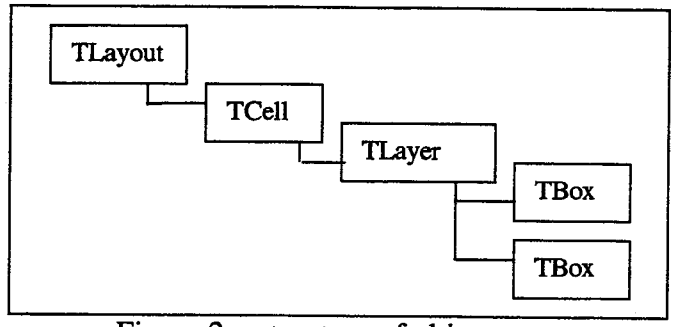

Figure 2 – structure of objects

The model incorporates a semantic meaning that needs to be treated by the tool. This treatment involves the correlation among rectangles in a same layer or between layers. As the description of an IC can involve millions of rectangles, the manipulation of extensive lists of rectangles implies a high cost in processing time and memory. To have efficiency in this treatment it is necessary to have an indexed structure.

The indexed structure was conceived starting from the concept of Span. A Span is an interval in the coordinates X or Y. To represent a rectangle it is necessary a Span in the axis X and another in the axis Y. A layer is described in this structure by a Plan. A Plan is a collection of SpanY's. A SpanY is formed by an interval in the axis Y and for a collection of SpanX's. A SpanX is just an interval in the X axis. The interval of a Span is represented by a pair Origin / Destination. The collections of Spans are ordered by the origin of each Span. An imposed restriction is that no SpanX can be consecutive to another. This restriction do not apply to SpanY's.

Previously, TLayer was defined as a list of TBox's. But, to support the indexed structure TLayer is formed by a plan of Span's. When a TBox is added a TLayer, a SpanX and corresponding SpanY are created. A SpanX is added to a SpanY, and a SpanY it is added to the Plan that composes TLayer. This way, it is not necessary that TLayer maintain a list of TBox's.

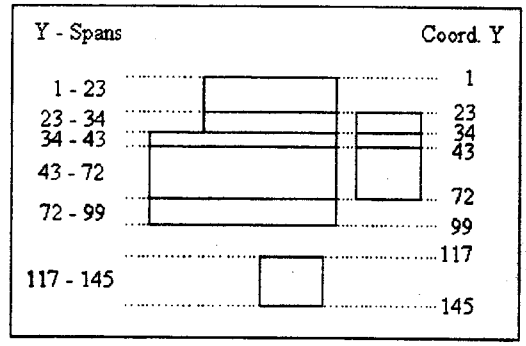

Figure 3 - *Spans*

The structure of Spans is projected to accelerate the query of related objects. This query can be operations like inclusion, intercession, union, inflation, proximity and exclusion. The acceleration is obtained as much by the 2D indexing as by a binary search in the lists.

For example, the corresponding algorithm commands the inflation of a layer using the structure of Spans:

It creates layer G.
For each SpanY of the layer A,
It creates SpanY SY.

SY = (Origin -1, Destination +1).
For each SpanX of SpanY
It creates SpanX SX.
SX = (Origin -1, I Destination +1).
It inserts SX in SpanY SY.
It inserts SY in the layer G.
The layer comes back G.

If we were using a list of rectangles, each rectangle of the list would be inflated in a similar way, and some rectangles could include others. Therefore it would be still necessary to remove the included rectangles, what would imply to compare each rectangle of the list to the other rectangles.

To accomplish any operation in the structure, it is necessary to scan the lists of objects, using the pattern iterator [2]. The application of this pattern consists of creating an object responsible for the scaning of the composite object, without exposing its internal representation. TIterator maintains the current state of the scan, allowing several separate scans in the same composition. Besides, the composite has a method that supplies its corresponding iterator. The following Delphi code sample shows a polimorphic operation in a composite:

```
Iterator := Compose.Iterator;
Iterator.FirstElement;
while not Iterator.Done do
  Iterator.NextElement.DoOperation;
```

Figure 4 – Pattern Iterator

The pattern visitor was used to acomplish the operation of painting the structure in the screen. The visitor represents an operation to be implemented in all the elements of the structure. For that a TVisitor object that executes the painting operation in each element of the structure is created. With the flexibility obtained by the use of the visitor, this solution was extended for other operations implemented in the structure.

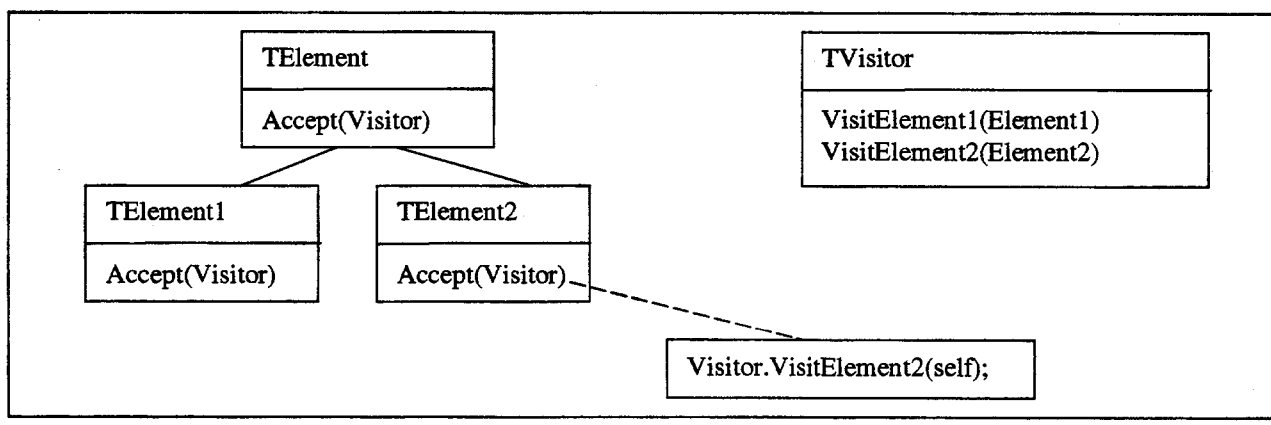

Figure 5 – Pattern Visitor

The operations implemented in the structure can involve transformations of coordinates as mirror, rotate and move. Taking as example the painting operation, it would be necessary to create objects for each one of those transformations and their combinations. Instead of that, the pattern decorator [2] was used. This consists of a flexible alternative for the use of subclasses, since the responsibilities are added

to the object dynamically. For this, it is enough to create an object TDecorator that makes the basic operation. And for each transformation it is created a subclass of TDecorator. In the example below, an object was created with three functionalities that were added dynamically.

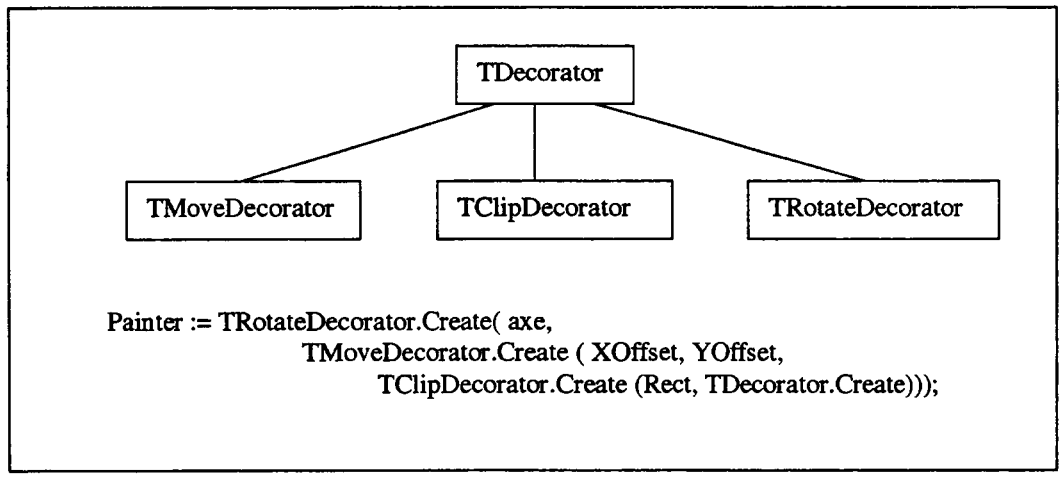

Figure 6 – Pattern Decorator

The structure of the model is rendered easily to the partition in distributed objects. The cells can be partitioned in its layers with these being allocated in several machines. To transport a layer to another machine, each SpanY is sent at a time. Each SpanY is encoded in a sequence of characters that is decoded in the destination machine. The component TPolymorphicList [13] converts objects into a sequence of bytes that can be then transferred as a stream between machines.
The structuring of the model with 2D indexing foresees the addition of several microelectronic related algorithms. An algorithm that extracts the list of transistors of the masks was implemented using this structure[9]. The ordering of the structure in vertical and horizontal segments allows that the complexity drops from N2 for O(log n)[8]. The whole structure was projected having in mind the integration with other modules and the encapsulation and partition in distributed objects. This architecture organization supports the scalability of the tool allowing the use of distributed algorithms.

Distributed architecture

The creation of a distributed system starting from an existing one usually involves a high engineering cost. It is desirable to minimize this cost. The unfolding of the architecture of the tool in a distributed system is a consequence of its original MVC structure. The system is implemented in three layers, maintaining the edition logic in the client, the control of distribution of tasks in the central layer, while the model executes the distributed algorithm. In the central control, a thread controls a group of queues that manages the system resources [11]. These resources are represented by an object proxy [2] that maps the remote object. The remote object implements the generation of a new layer starting from one or more existent layers. The remote objects are implemented as DCOM objects [3], as well as the proxies and the central layer.
The proposal of the distributed architecture is made according to a politics of minimum intervention in the original architecture. The MVC paradigm already existent it is unfolded in the formation of a three-tier system.

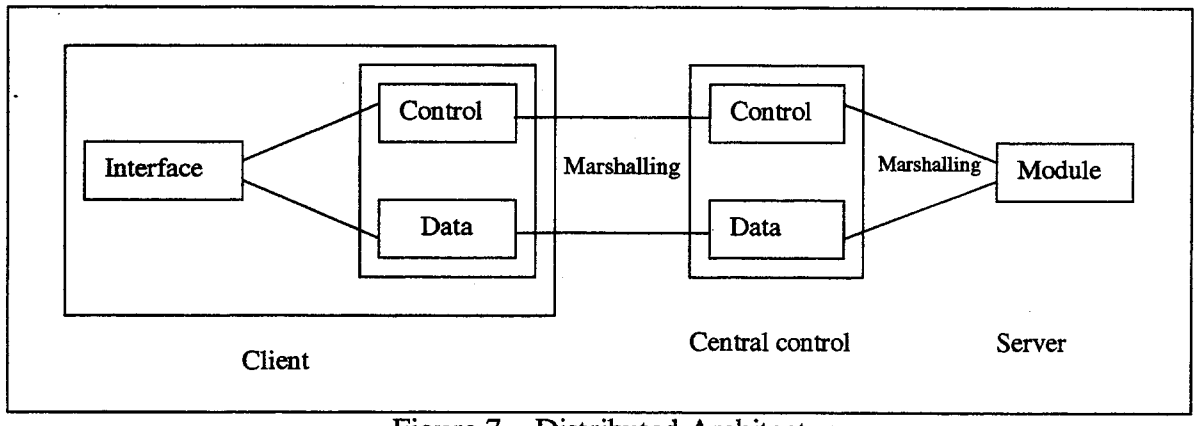

Figure 7 – Distributed Architecture

The application control located in the client, sends commands through the control channel. The control channel implements the Chain of Responsibility [2] pattern. It provides the direction of messages through the controllers located in the client and central control. The data transfers across the tiers are established through the marshalling of objects. The transmission of data is made among remote objects, which communicate through the DCOM protocol.

The distribution control resides in the central layer. The processing of the verification algorithm is controlled by a script written in XML [14] that describes a group of rules. The script is processed by a parser that generates commands, that will be executed remotely. Those commands invoke the pseudolayer creation, resultants of operations among layers. The builder [2] classifies the pseudolayers according their rule dependencies. If the pseudolayer only depends on original layers, then the operation can already be executed. Otherwise, the operation will have to wait for its resources. The verification rules are ordered manually in the XML file, since we have chosen to implement a simplified allocation heuristic. A better implementation would traverse the dependency graph and order the rules according to their precedence.

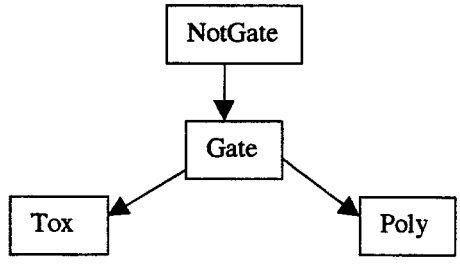

Figure 8 - Graph

The illustration above exhibits the structure of the dependency graph. NotGate is a pseudolayer that depends on another pseudolayer Gate. On its turn, Gate depends on the original layers Tox and Poly. These pseudolayers are encapsulated into object proxy's [2] that constitute the consumers and producers of the system.

We used the pattern command [2] to do a refinement in the solution of the problem. Several rules exist and depending on the type of the rule, a different sequence of commands is executed. The parser invokes the execution of the command. The command then sends requisition for the builder, as illustrated in the illustration below.

Figure 9 – Pattern Command – First level

The command pattern also acts in a posterior level. In this case, the proxy assumes the role of a command. Depending on the proxy type that it is being executed, a different block of commands is called. Thus, the structure of the pattern is used recursively. The illustration below illustrates that level. The central control starts the thread execution. The thread executes the proxy, which executes the server.

Figure 10 – Pattern Command – second level

The controller defines a thread that manages the resources based on two monitor's [11]. One of the monitors controls the liberation of the processors. This monitor is increased whenever a machine finishes its execution, and is decreased whenever a machine is allocated to a task. Initially, this monitor is increased for each available machine. The other monitor controls the queue of ready tasks. It is increased whenever there is a new task to be executed and decreased when the task is allocated to a machine. This monitor is necessary; to avoid the task being removed of the queue before it is ready to be executed.

The proxies play a primordial role in the management of resources. The central control manages the resources using three queues of objects. The first queue contains the objects that need resources. The second queue contains the objects that don't need resources. When a task depends on the termination of another task, the corresponding proxy object is allocated the wait queue. But if the task needs a primitive layer, the object is allocated the second queue. Besides, it is necessary a third queue for the tasks that are in execution. This queue reflects the processors that are busy. This way, when the task terminates execution, the corresponding object is removed of the third queue, and the processor is allocated to another operation.

When the central control invokes the first proxy from the ready queue, this proxy starts a remote server process. From the point of view of the central control, this proxy is a resource producer, while for the server, is a resource consumer. The proxy can play those roles since it implements the pattern Observer [2]. The consumer monitors the producers necessary to the execution of its own task. The producer maintains a list of its consumers. When the task completes, the producer notifies the end of the operation to all the consumers that observe it.

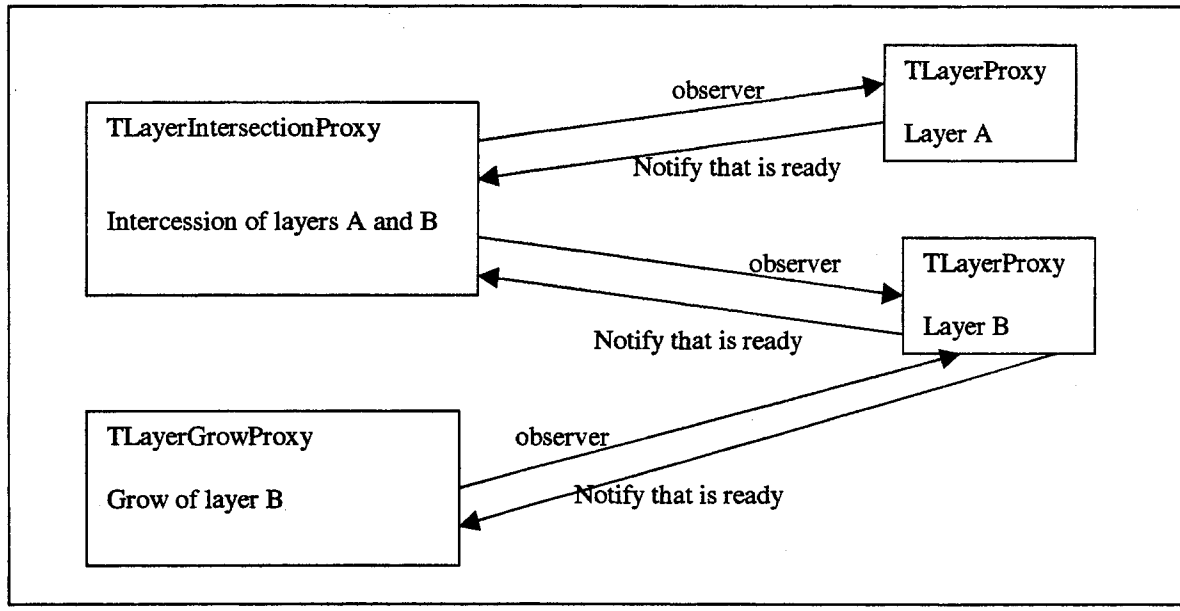

Figure 11 – Proxy/Observer

The figure above illustrates the operation of intercession of two layers. The operation can only be executed when the layers A and B are ready. When the intercession is notified by the layers, it executes the intercession operation and notifies its observers that is also ready. Besides, the proxy notifies the controller of the end of the operation and it is promoted to the ready queue.
The described architecture implements a distributed producer/consumer system [11]. Its conception was based on orientation-oriented techniques. These techniques promoted the encapsulation of the producers and consumers functionality's, and the reusability of code in throughout the layers. The standardization obtained by using the XML description language, renders the architecture flexible enough to support other implementation algorithms.

Execution of the Algorithm

The algorithm was tested using a very simple circuit that is randomly modified for generation of errors. The program uses two files: one for input and another for output. The input file contains the following data: a certain one numbers of rectangles, which are the main generators of the errors; a certain area of integrated circuit, where the errors are generated. The output file maintains the result of all the configurations.
When being executed, the program reads the input file and it inserts rectangles of several sizes and positions inside several layers of the integrated circuit. The objective is to test the time of execution and the capacity of the memory of the used computers. All results are recorded in the output file. This file contains: the number of rectangles; the area of integrated circuit; the date when that configuration was accomplished; the hour that the program begins to be executed (initial time); the hour that the program finishes execution (final time); the elapsed time;
Several error configurations were tested. The battery of tests used IBM PC Pentium 166Mhz computers, with 64 MB of memory, networked in a 10Mbits Ethernet. The results of some of these configurations are mentioned below in the table.

Number of Rectangles	Area of integrated circuit	Elapsed time (h:m:s:ms)	Number of machines
800	800	0:7:10:419	3
		0:7:05:219	5
800	400	0:6:06:988	5
		0:5:59:297	3
1000	400	0:5:13:811	4
		0:2:36:855	1

Table 1-Execution of distributed algorithm

In agreement with the table, we verified that, with a same number of rectangles, as smaller the area of integrated circuit, larger the time of execution of the program. That is due to the fact of existing a larger density (number of rectangles for area of integrated circuit), that is, if a rectangle is drawn in an area where spans already exist, it implies an increase of processing.
These results confirm the scalability, since the operations of high density could only be completed with a large number of machines. However the results were contrary to the expected in terms of performance. The simplified heuristic of task allocation can be blamed for the degradation of response time. As the graph of dependencies was not observed, they provoked an excessive number of transfers among machines. The transfer process is also made through a non-optimized marshalling procedure. In this simplified implementation the servers executed with a single thread, being idle during the transport of data.

Conclusion

Scalable distributed architectures are in general an economic solution for the problem of constant growth in information systems and in particular for CAD tools. They present a better solution than to allocate in a single machine a great amount of resources. These resources can be better taken advantage of if distributed among several workstations.
The original project of the presented system was thought carefully to be adaptive for a distributed scalable model. The only necessary modification was to add a controller for the distributed resources. The structure of objects supported the distributed model easily and it was reused without changing any code line. The architecture produced a basic scalable platform where several distributed algorithms can be developed. To increase the computational capacity it suffices to add more machines in the host list.
In spite of the contrary performance results, the experiment demonstrated the viability of implementing a scalable system. This work will can serve as a platform to develop new heuristics, allocation techniques and resource transfer of among machines, load balancing and reallocation. Other algorithms already implemented in the lumped version of the tool will be migrated for the distributed model. Extraction algorithms, simulation, routing and placement can validate the flexibility of the architecture and the applicability of the distributed model for scalability of processing resources.

Bibliographical reference

[1] Furlan, J.D., "Modelagem de Objetos através da UML – The Unified Modeling Language", MAKRON Books, 1998.

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns : Elements of Reusable Object - Oriented Software", Addison-Wesley, 1998.
[3] Eddon, G., Eddon, H., "Inside Distributed COM" – Microsoft Press, 1998.
[4] Weste, N., Eshraghian, K., "Principles of CMOS VLSI Design", Addison-Wesley,1988.
[5] Mead, C., Conway, L., "Introduction to VLSI Systems", Addison-Wesley,1980.
[6] Oliveira, C.E.T. e Anido, M.L., "TEDMOS para Windows", IX Congresso da Sociedade Brasileira de Microeletrônica, Campinas, pp. 65-73, August, 1994.
[7] Nunes, R.B., Anido, M.L. e Oliveira, C.E.T., "Circuit Verification Using Spans – A DataStructure with O(n) Algorithms", IX Congresso da Sociedade Brasileira de Microeletrônica, Campinas, pp. 65-73, August, 1994.
[8] Nunes, R.B., Anido, M.L. e Oliveira, C.E.T., "A New Approach to Perform Circuit Verification Using O(n) Algorithms" , IEEE Procedings of the EUROMICRO'94 conference, Liverpool, IEEE Computer Society Press, pp. 428-434, 1994.
[9] Alcântara, J.M.S., Oliveira, C.E.T. e Anido, M.L., "A Novel Circuit Extration Tool Based on X-Spans and Y-Spans", IEEE Procedings of the 21st EUROMICRO Conference, Prague, Tcheck Republic, September, 1996.
[10] Nunes, R.B., Anido, M.L. e Oliveira, C.E.T., "A New Approach to Perform Circuit Verification Using Spans" , IEEE Procedings of the 38th Midwest Symposium on Circuits and Systems, August, Rio de Janeiro, Brazil, 1995.
[11] Stallings,W., "Operating Systems – Internal and Design Principles" – third edition- Prentice Hall,1997.
[12] C.E.T. Oliveira, A.L.C.L. Duboc, A.P.V. Pais, D.P. Muniz, M.L. Anildo. "Aplicações de Patterns no Desenvolvimento de Um Sistema CAD para Microeletrônica" Núcleo de Computação Eletrônica, UFRJ,1999.
[13] Web - http://www.tecepe.com.br/omar.
[14] Wrox Development Tem, Ducket, J. "Professional XML"-second edition - Wrox Press