Relatório Técnico

Núcleo de Computação Eletrônica

# A Distributed Object-Oriented Design Rule Checker

Pais, A. P. V.
Anido, M. L.
Oliveira, C. E. T.

NCE - 11/01

Universidade Federal do Rio de Janeiro

# A Distributed Object-Oriented Design Rule Checker

A. P. V. Pais, M. L. Anido, C. E. T. Oliveira

Universidade Federal do Rio de Janeiro, anapais@nce.ufrj.br, mlois@nce.ufrj.br, carlo@nce.ufrj.br

## Abstract

*VLSI algorithms are complex, dynamic, specialized, demanding CPU and memory. To support such dynamic computing demands, it is necessary to employ a scalable system, which can be easily adapted and extended to run newer VLSI algorithms. This paper describes the design of a distributed object-oriented Design Rule Checker (DRC), focusing on its implementation methodology. It also proposes the use of XML to describe technology design rues. These rules were executed on a distributed OO structure. The paper also shows that by using programmable scripts that carry out a modular encapsulated structure, a robust and adaptable system can be obtained.*

## 1. Introduction

Computer Aided Design (CAD) tools, or more specifically, Electronic Design Automation (EDA) tools must cope with projects which go beyond the limitations of current computers. For example, small-sized circuits for today's standards, with approximately 50.000 to 200.000 transistors, may require a few hours to run algorithms such as DRC and Circuit Extraction. High-end circuits which are two orders of magnitude larger, would require from hundred to thousand hours to process. To attain results in a feasible time, the only solution is to distribute algorithm execution among distinct machines. The parallelization of this class of problems has been addressed in several papers and books [1][2][3][4].

Scalable tools can be created by applying techniques of distributed objects. However, powerful workstations, capable of carrying out the processing of more complex tasks, are scarce. In the days of desktop computing, the commonplace is the availability of a group machines with limited resources rather than powerful workstations. Nevertheless, if the resources of these machines were put together to perform a complex task, the performance could be similar, or even better, to the performance of a powerful and expensive workstation.

A difficult task can usually be broken down into simpler tasks, and each task can be assigned to a machine. Independent tasks can be carried out at the same time, decreasing the total processing time.

This paper proposes an architecture to face the scaling challenge for VLSI algorithms. A distributable structure is capable to scale to dimensions of larger circuits by partitioning among a cluster of computers. This structure can support several VLSI algorithms and is programmable using a XML [5] script. Exemplifying this architecture, this paper presents the implementation of distributable DRC algorithm.

Section two gives an overview of design rule checking operations and discusses the representation of design rules in XML. Section three presents an overview of the architecture responsible for algorithm execution and discusses how it can be used to support scalable and distributed implementations. Section four details the execution architecture and its implementation. This is followed by an assessment of the results, depicting weakness and advantages of XML and distributed implementations. At the end, several conclusions are presented.

## 2. Specifying Design Rules

### 2.1. Design Rule Checking

Design rules usually specify minimum track width, track separation, and the extension of one layer to another, targeting the formation of an active element. Design rule verification is based on elementary boolean operations (AND, OR, XOR), applied on physical layout masks. These masks are described using a set of geometric forms which usually are rectangles, as shown in Fig. 1.

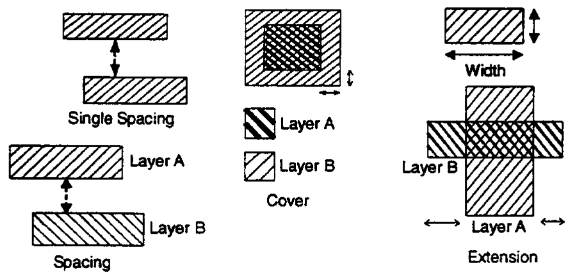


**Fig. 1: Design Rule Checking (DRC) rules**

The operation of design rule checking usually implies on the verification of forty to seventy different design rules, where each design rule has to be applied to hundreds of thousands or millions of polygons. Each design rule can be broken into a set of elementary tasks and each task can have a different set of dependencies to be solved. Each task can be considered independent from others at the level of design rules. However, in order to have an efficient solution, it is necessary to analyze other aspects of the problem, such as differences of complexity among tasks. These could lead to an unbalance of the parallelization, and also to operation trashing, that is, performing the same operation many times.

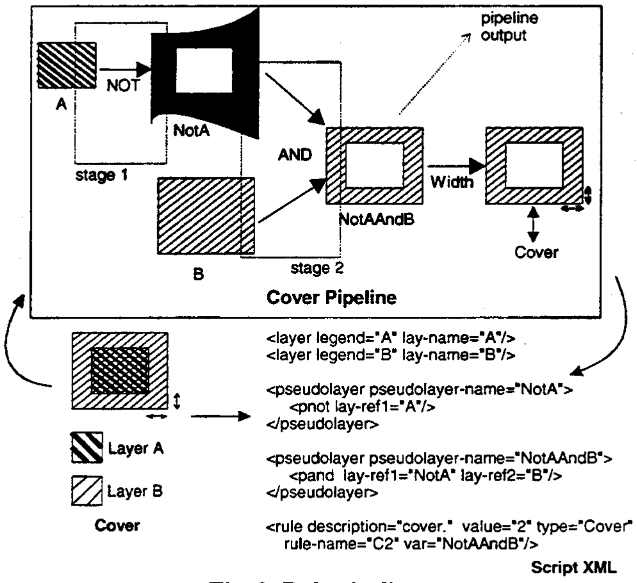
The DRC problem is characterized by the transfer of a sub-set of the data structure (the layers involved in a particular design rule) to each processor, prior to the start of the verification of a design rule. Usually, DRC is performed hierarchically and incrementally, that is, usually DRC is not applied to the entire layout, but to a cell or a group of cells. Additionally, cells that were checked previously and were not affected by any modification do not need to be checked again. This implies that the communication setup problem is not so large as it could look at a first sight. On the other hand, a DRC operation on the flattened circuit may require a transfer of hundreds of thousands of polygons (the data structure) prior to the start of a particular rule check. Fortunately, DRC operations on the flattened circuit are rare. Moreover, even this situation can be broken into distinct tasks.

## 2.2. Describing the Design Rules using XML

This paper proposes the use of a XML document to describe technology design rules. XML is a script language and a XML document is a database in text format. XML elements, described in a XML file, are organized in a hierarchy, like a tree. Each element described in XML has a set of attributes, which identifies it, and also can contain a set of other elements. A XML description forms a tree where each element can have zero or more descendants.

An element describing the fabrication technology is the root of the XML script. The tree is composed by a set of layer elements, a set of verification rules (drc element) and a set of electrical rules (extractor element). A layer is a leaf element and is identified by a name.

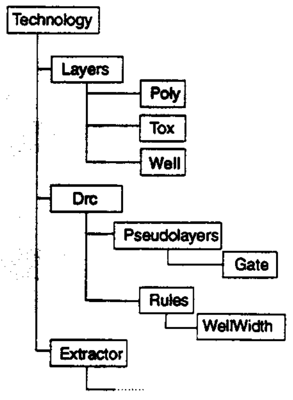
The validation of design rule presumes the execution of a boolean operation on one or more layers, generating another layer as result. This result is termed pseudolayer, which can be used as an input for another operation. A pipeline can describe a design rule, where each stage has one or two inputs and one output. Each stage executes a boolean operation. The inputs are layers or pseudolayers and the output is a pseudolayer. The width rule is executed on the pipeline output, finishing the rule. Fig. 2 illustrates this pipeline.



```
<layer legend="A" lay-name="A"/>
<layer legend="B" lay-name="B"/>

<pseudolayer pseudolayer-name="NotA">
   <pnot lay-ref1="A"/>
</pseudolayer>

<pseudolayer pseudolayer-name="NotAAndB">
   <pand  lay-ref1="NotA" lay-ref2="B"/>
</pseudolayer>

<rule description="cover." value="2" type="Cover"
   rule-name="C2" var="NotAAndB"/>
```

Fig. 2: Rule pipeline

The drc element is composed by a set of pseudolayer elements and a set of rule elements. The set of pseudolayer elements corresponds to pipeline stages of all design rules in the script. Each pseudolayer has attributes, which identify the boolean operation and the inputs (layers or pseudolayers). The rule element has attributes, which identify the rule and the pseudolayer corresponding to the rule pipeline output. The following picture illustrates how the XML script corresponds to a hierarchy of elements.



```
<tecno name="Circo12">
 <layers>
  <layer legend="Poly" lay-name="CPOL"/>
  <layer legend="Tox" lay-name="CTOX"/>
  <layer legend="Well" lay-name="CNWI"/>
 ....
 </layers>
 <drc>
  <pseudolayers>
   <pseudolayer pseudolay-name="Gate">
    <pand lay-ref2="Poly" lay-ref1="Tox"/>
   </pseudolayer>
 ....
  </pseudolayers>
  <rules>
   <rule description="WellWidth>=20"
      value="20" type="Width"
      rule-name="WW20" var="Well">
   </rule>
  </rules>
 </drc>
 <extrator>
 ....
 </extrator>
</tecno>
```

Fig. 3: XML hierarchy of design rule script

## 3. Architecture Overview

The DRC tool was developed to be integrated into a framework for the design of integrated circuits and it encompasses a modular structure based upon the MVC (Model-View-Controller)[6] paradigm, illustrated by Fig. 4. This paradigm is largely used in software systems because it yields a loosely coupled architecture.

Integrated tools for microelectronic are usually composed by the aggregation of specialized modules developed by several people. Integrating such modules demands a flexible system, capable of easy adaptation to several specifications. This requires a software isolation layer between the interface and the several modules. The isolation layer is partitioned into two main parts, one for control and another for data. The control part transfers commands between the interface and the each module. The data connection provides a correspondence between the internal representation of the model and the visual presentation of the data.

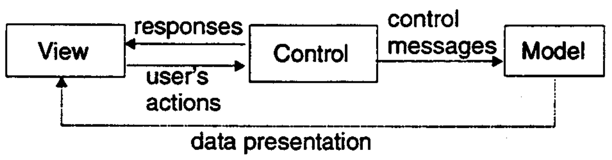


Fig. 4: Model-View-Control (MVC) Architecture

The MVC paradigm organizes the software architecture in three distinct components, encompassing the user interface, operation control and internal structure of a system. The view component concentrates on the user interface, responsible for data presentation and user interaction with this data. In this tool, it enables the edition of a physical layout mask. The user interface can issue a DRC check command and present the result as graphical error marks. The control component process the DRC check command, traversing the XML design rule script and issuing a stream of requests to the model component. The control parses the XML script and assembles a collection of basic operations that are queued on a task list. The control pumps these tasks to the model component according to an operation strategy that may vary between lumped and distributed models. The model component encapsulates the internal structure representing the physical layout model. The model component offers a set of basic operations that can be used to perform several algorithms as DRC check, circuit extraction, mask edition or layout compression. The result depends on the combination of the operations ordered by the control component. The model component reacts to each request received from the control by generating, removing of modifying model elements. The model can also send elements to the view for presentation.

## 4. Execution Architecture

A single computer can execute the rule driven algorithm sequentially, scheduling non-pending tasks one at a time. The Fig. 5 represents the execution flow on a single machine implementation. Notwithstanding, the original MVC architecture can be modified for distributed execution by replication of model nodes.

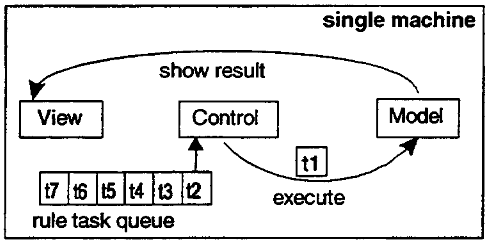


Fig. 5: Sequential DRC execution

### 4.1. Distributed Architecture

To make the architecture distributed, the isolation layer was restructured so that the communication between the interface and the modules was performed remotely. Parallelizing the algorithm follows from the analysis of design rules, which determine which tasks can be executed concurrently and the dependencies among them. That information can be specified in the design rules file, so that its interpretation determines parallel sequences of tasks.

Each design rule can be represented by a set of simple operations. The DRC tool analyses the rules and creates a set of tasks to be executed. Some of these tasks are independent from each other, while other depends on the results of already executed tasks.

Previous work shows how the combination of data parallelism and task parallelism can be used to implement design rule verification [1][2]. Macpherson [1] describes how it is possible to divide DRC execution in tasks, which can be executed simultaneously. It also shows that task parallelism is not enough to obtain efficiency. Macpherson divides the execution of one task among several processors, performing load balancing. As a result, it supports the execution of DRC of very large circuits. The architecture depicted on Fig. 6 proposes a partition where execution load can be distributed among several machines.

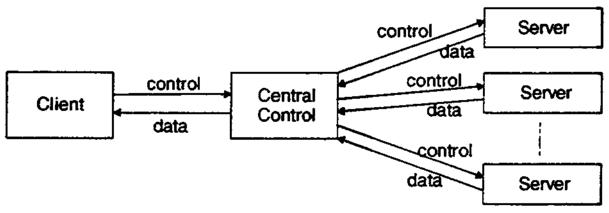


Fig. 6: Distributed architecture

The proposal for a distributed architecture is effected based upon a philosophy of minimum intervention in the original architecture. The existing MVC paradigm is unfolded for the formation of a three layer system, keeping layout editing in the client, the control for task distribution in the central layer, and the model running the distributed algorithm.

The implementation was broken in steps. The first step consists of separating the view from all the rest. Thus, the architecture is broken in two modules, a module client and a module server. Notice that Fig. 7 splits the architecture in two parts, concentrating the user interface into a client machine, leaving the server with the core algorithm processing. The transmission of control messages and data is effected through the network.

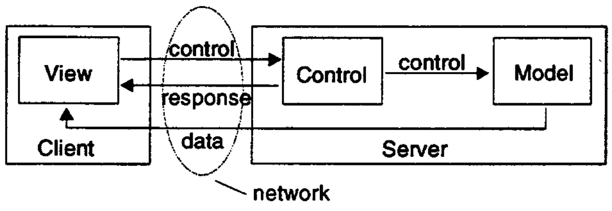


Fig. 7: Client-Server Model

The second step consists of dividing the server module into two modules: a central control and a server. A three-tier model, as illustrated in Fig. 8, allows a loosely coupled architecture where model and control are independent modules. Consequently, the control remains in the central module and the model in the server. The messages from the control to the model, and from the model to the view are transmitted through the network.

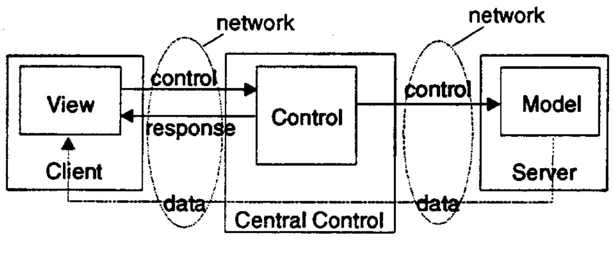


Fig. 8: Three-Tier Model

After the conclusion of the second step, three distinct software layers are obtained: client, central control, and server. For the client and the central control, there is only one module for each, notwithstanding, there can be several server modules. Thus, the central control has the objective of distributing the physical layout model among several server modules, according to computer availability. Additionally, the central control divides the algorithm into separate tasks, and delegates the execution of each task to a server computer.

## 4.2. Task Dependency Graph

In the concentrated version of the DRC, the XML file that describes the design rules is executed sequentially, in the order specified by the script. On the other hand, in the distributed version of the DRC, the script parser generates a set of objects that represent the tasks. If a task depends on another one, the corresponding objects keep connections that reflect the dependency.

Objects corresponding to tasks keep a dependency relationship, which can be represented by a directed graph. The diagram shown in Fig. 9 demonstrates this representation, where graph nodes correspond to layers and pseudolayers.

The analysis of the dependency graph is useful to determine task scheduling and the beginning of the execution of the algorithm can be taken as an example. Let us suppose that there are five tasks to be executed, and that there are just three computers available. The choice of the three tasks can be made in a random manner or it can be made by observing the dependencies graph. Each task can have an execution priority. If six tasks depend on the execution of task A and four depend on task B, task A must have a higher priority than task B. Thus, task scheduling can take into account a priority order. The use of more elaborated heuristics for task scheduling may result in improving algorithm efficiency. A proper task ordering tends to reduce the total algorithm execution time.

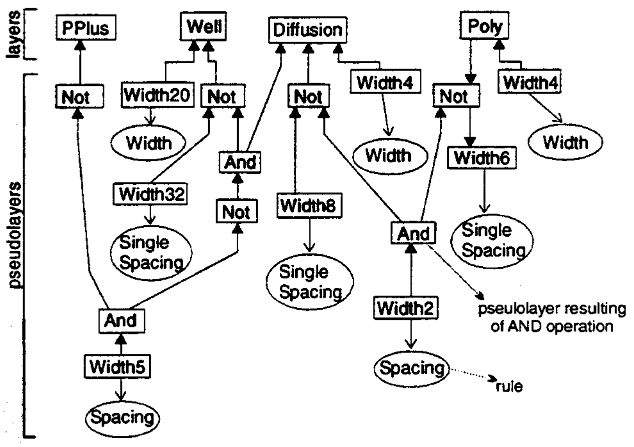


Fig. 9: Task Dependency Graph

## 4.3. Process Control

The central control manages the resources based upon three object queues, where the objects represent tasks to be executed. The wait queue contains objects waiting for resources and the ready queue contains objects that do not depend on anything any more. When a task needs the result of another task, the corresponding object is allocated in the waiting queue. If the task is an operation between primitive layers, there are no dependencies, then the object is allocated in the ready queue. Moreover, an execution queue is necessary for executing tasks. This queue reflects busy processors, allocated to rule execution. When a task ends, the corresponding object is removed from the execution queue, and another task is allocated to the processor. The scheme in Fig. 10 shows an example of parallel execution using two processing modules.

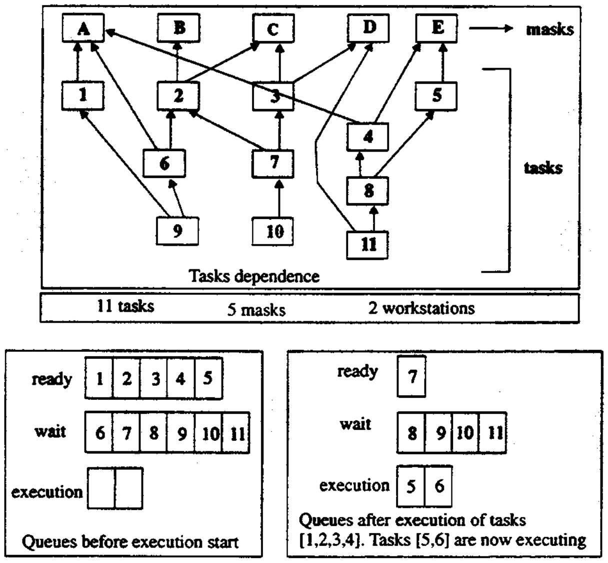


Fig. 10: Parallel DRC execution

The central control defines a thread that manages the resources, synchronized by two monitors. One of the monitors controls the liberation of the processors and it is incremented every time that a processor ends its execution, and is decreased every time that a task is allocated to a processor. Initially, this monitor is incremented for every available processor. Another monitor controls the ready task queue and it is incremented every time there is a new task to be executed and it is decreased when a task is allocated to a processor.

When the central controller invokes the first object of the ready queue, this object initiates a resource server process. This object is a producer for the central controller and it is a consumer for the server. To make the object play these two roles, it implements the pattern Observer [7]. The consumer monitors the producers, which produce the necessary data to execute the task associated to it. Dependency interaction between several tasks is exemplified in Fig. 11. The producer keeps a list of its consumers. When a task execution ends the producer signals the operation termination to all consumers that are observing. It also notifies the central controller for task synchronization. When a consumer is ready, it notifies the controller. Once the central controller receives this notification, it promotes the consumer to the ready queue.

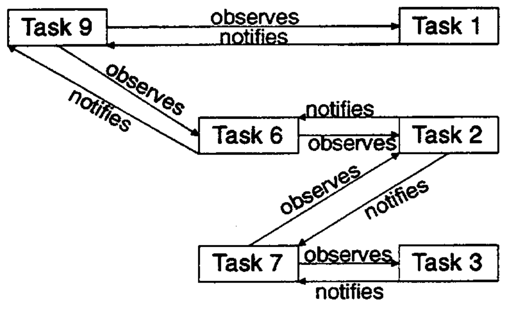


**Fig. 11: Observer**

The architecture described implements a distributed consumer/producer [8] system. Its conception was based on object-oriented techniques. These techniques ensure the encapsulation of the functionalities of producers and consumers. Moreover, this architecture is applicable both to the implementation of simple logic, such as the one used, and to the implementation of complex heuristics.

## 5. Assessment

### 5.1. XML description of design rules

Design rule description in XML requires the creation of a new dialect, capable of detailing mask constraints. The proposed dialect is flexible enough to cope with technology changes and support processing instructions for new DRC algorithms. Changes in the algorithm may require the creation of new elements, mapping layers, tasks or rules.

The XML rules description implies a script directing the verification algorithm execution. Direct interpretation of this script determines the flow of verification tasks to be executed. Parallelism extracted from the script relies exclusively on the implicit dependency graph established by the stated rule checking sequence. This simple strategy may not suffice for an optimized distribution of tasks. The XML description can be enhanced with tags explicitating a required strategy for task control. A pipeline level tag may be added to force the execution of rules at a certain dependency level. Level 0 describes process layers that has no dependencies. Level 1 is assigned to layers depending on layers and tasks pertaining to level 0. Level 2 are tasks depending of at least of one task at level 1 and so forth. Task sharing the same pipeline level tag can execute in parallel. Task can then be promoted to later dependency levels by incrementing their tags in the script, to take into account a different scheduling strategy.

### 5.2. Distributed data considerations

Distributed DRC execution implies data exchange between remote executing tasks. Starting a new task may depend on a existing or pending layer situated in a remote machine. Layer data transport may impose a severe overhead on overall execution, since network delay can match to task turnarounds, manifolding completion time. Object transport involves memory image serialization. Serialization consists of turning object contents into a data stream suitable for transfer and reconstitution in a different representation or location. Transmission of object streams is a costly operation.

The serialization process implies in an extra cost, added to both sides of a transmission link. Efficiency in distributed tasks is then associated with a delicate balance of object granularity in data transmission. In one side, large objects imply in long latency, wasting processing time. Small objects attach huge overheads stemming from serialization and connection times. One possible solution is to use pipeline techniques, where task execution overlaps with block transmission. A distributed design pattern, called buffered iterator [9] can provide the logic necessary to this pipelined operation.

### 5.3. Representation and Data Transfer

The internal layout model representation is highly effective for locality algorithms, due to its bidimensional indexing. However, as a complex, finely fragmented object, it represents a hindrance to efficient serialization and subsequent transposition to a remote or alternative representation. From this point of view, CIF [10] is a more compact representation where internal fine grain structure [11][12][13][14] is described as rectangles or closed polygonal paths. This should be a more economic language for serialization and data transfer. Moreover, since most algorithms can be reduced to a linewise vertical scan, only a small horizontal band of the circuit needs to be exploded into a full-fleshed indexed structure. The rest of the circuit can be stored in the more compact CIF representation, either in memory or in disk. CIF can also be represented in an equivalent XML format. The advantage is as faster conversion across representations. Scanning algorithms can be pipelined with a parser that may source XML either from local or remote storage. A rolling horizontal band is being generated alongside script parsing and discarded as scan line advances. This might also be an option for data distribution as the internal structure can be encoded in a higher-level dense representation. A drawback is that, XML being a verbose text format, representation length of serialized streams can be several times bigger than the equivalent binary. One extra filter can be interposed to apply a compression/decompression encompassing transmission. Compression processing is another overhead that should be taken into account. Portability, easy parsing of XML must be pondered against verbosity and extra level processing to assess the economic advantages and disadvantages of such approach.

## 6. Conclusion

This paper described the architecture and design of a Distributed Object-Oriented Design Rule Checker (DRC), focusing on the methodology employed to implement such distributed application.

A key point in the paper was the discussion about distributing the problem on several machines. Normally, it would require a radical reengineering of a system originally designed for single machine operation. However, criterious application of object-oriented techniques in the original architecture produced a readily distributable design. A distributed architecture is obtained by simply splitting and replicating existing modules.
Modules in the object-oriented architecture are encapsulated, which entails the possibility of experimenting with several algorithm implementations without affecting other modules. The XML script is not affected as well, and can be interpreted to drive a range of serial and distributed algorithms.
Finally, the paper shows that the combination of programmable scripts with a modular encapsulated structure can produce a robust and adaptable system.

## References

[1] K. Macpherson, "Parallel Algorithms For Layout Verification", University of Illinois, M. Sc. Thesis, 1995.

[2] P. Banerjee, "Parallel Algorithms for VLSI Computer-Aided Design" - PTR Prentice Hall, 1994.

[3] B. Ramkumar and P. Banerjee, "ProperCad: A portable object-oriented parallel environment for VLSI CAD", *IEEE Transactions on Computer-Aided Design*, vol. 13, pp. 829-842, July 1994.

[4] K. P. Belkahale, "Parallel Algorithms for CAD with Applications to Circuit Extraction", Ph.D. dissertation, University of Illinois, Dept. of Computer Science, 1991.

[5] B. McLaughlin, "Java and XML", O'Reilly, 2000.

[6] "Developer's Guide - Borland - JBuilder 2", Borland, 1998.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1998.

[8] W. Stallings, "Operating Systems - Internals and Design Principles", third edition, Prentice Hall, 1997.

[9] P. Brooks, "Simple Buffered Collection and Buffered Iterator Patterns", OOPSLA'95, 1995

[10] C. Mead, L. Conway, "Introduction to VLSI Systems", Addison-Wesley, 1980.

[11] J. K. Outsterhout; "Corner stitching: A data-structure technique for VLSI layout tools", *IEEE Transactions on Computer-Aided Design*, vol. CAD-3, pp. 87-100, Jan. 1984.

[12] H. Samet; "The quadtree and related hierarchical data structures", Computing Surveys, vol.16, pp. 187-260, 1984.

[13] R. B. Nunes, M. L. Anido, C. E. T. Oliveira; "A New Approach to Perform Circuit Verification - Using O(n) Algorithms", *Proc. 20th Euromicro Conference*, IEEE Comp. Soc. Press, pp.428-434, August 1994.

[14] R. B. Nunes, M. L. Anido, C. E. T. Oliveira; "Circuit Verification Using Spans - A Data Structure with O(n) Algorithms", *IX SBMicro*, pp. 64-73, August 1994.