Relatório Técnico

Núcleo de Computação Eletrônica

Developing a Distributed Architecture for Design Rule Checking

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

NCE - 10/01

Universidade Federal do Rio de Janeiro

Developing a Distributed Architecture for Design Rule Checking

A. P. V. Pais, M. L. Anido, C. E. T. Oliveira
Universidade Federal do Rio de Janeiro, carlo@ufrj.br

Abstract

This paper describes the design and implementation of a distributed object-oriented Design Rule Checker (DRC). The main focus is on the methodology employed to implement this distributed application. Code reusability is achieved using an OO approach, making objects available for other tools, such as circuit extractor. The paper also addresses the application of design patterns, which produce loosely coupled elements, facilitating the integration of system modules.

1. Introduction

VLSI algorithms are complex, dynamic, specialized, and demand 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. In order 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 by 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. If these resources 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.

Usually, a difficult task can be broken down into simpler tasks, and each task can be assigned to a machine. Additionally, 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 of VLSI algorithms. A distributable structure is capable of scaling 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. This paper presents the implementation of a distributed DRC algorithm, which allows the exemplification of the distributed architecture.

2.Specifying design rules

2.1.Design rule checking

Design rules usually specify minimum track width, track separation, and the extension of one polygon to another, targeting the formation of an active element. Design rule verification is based on boolean operations (AND, OR, XOR) performed on layers containing a set of geometric forms which usually are rectangles. This is illustrated by figure 1. Specifically, most design rule verification imply in generating a pseudo layer resulting from intersection or union of rectangles existing in pair of operand layers.

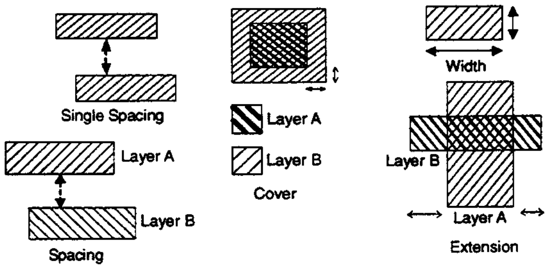

Figure 1- Design Rules Specification

The operation of design rule checking usually consists of the verification of forty to seventy different design rules, where each design rule has to be applied to hundreds of thousands or even 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, DRC is not applied to the entire layout, but to a cell or a group of cells. Additionally, cells that were previously checked and that were not affected by any modification do not need to be checked again. On the other hand, a DRC operation on the flattened circuit may require the 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.

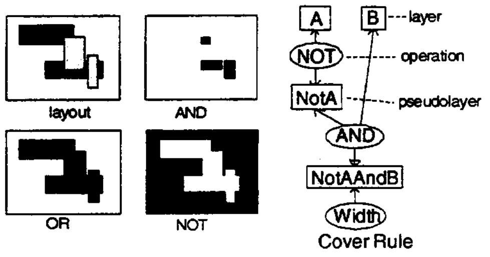

Figure 2 – mask operation flow

2.2. Describing design rules using XML

Validation of design rules requires the description of design constraints in appropriated language. Description of design rules in XML has the advantage of being portable, since there are many parsers available in a number of programming languages. XML can also embed instructions to guide algorithm execution. The following XML fragment shows how encoded design rules are converted to objects representing the verification algorithms.

```
<layer legend="Poly" lay-name="CPOL"/>
<layer legend="Tox" lay-name="CTOX"/>

<pseudolayer pseudolay-name="PolyOrTox">
  <por lay-ref2="Poly" lay-ref1="Tox"/>
</pseudolayer>

<pseudolayer pseudolay-name="NotPolyOrTox">
  <pnot lay-ref1="PolyOrTox"/>
</pseudolayer>

<rule description="DistToxPoly>=4" value="4"
   type="Spacing" rule-name="SCDP4"
   var="NotPolyOrTox"/>
```

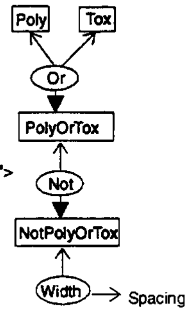

Figure 3- XML rules bound to object operations

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) paradigm [6], illustrated by figure 4. This paradigm is largely used in software systems because it yields a loosely coupled architecture.

Integrating specialized modules, developed by several people, demands a flexible system, capable of easy adaptation to several specifications. Thus, a software isolation layer between the interface and several interchangeable modules is necessary. 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 each module. The data connection provides a correspondence between the internal structure of the module and the visual presentation of the data.

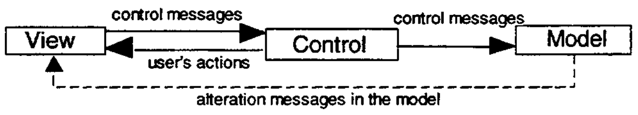

Figure 4 - MVC architecture

3.1. Using design patterns

The most internal module of the system comprises of the physical layout model together with the algorithms to validate it. Design patterns [7] allow assembling these parts in a more flexible architecture. Data structures and algorithms can be interchanged to the best fit without affect each other. Moreover design patterns enhance encapsulation providing a cleaner interface to external modules and allow its transport and the distribution on several computers.

The internal module encapsulation is provided by the following design patterns: composite [7], iterator [7] and visitor [7]. The main advantage of these patterns is to make available the contents in an encapsulated manner, hiding its details. Moreover, it makes the structure flexible and adaptable allowing the inclusion of new algorithms. Thus, the model becomes more orthogonal, being viewed by several system operations in the same manner.

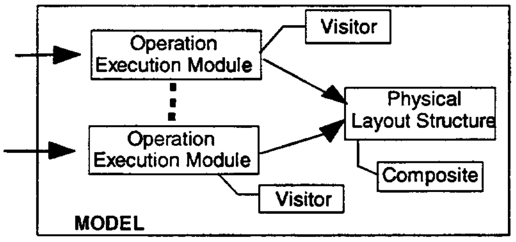
Visitor
Operation Execution Module
Physical Layout Structure
Operation Execution Module
Composite
Visitor
MODEL

Figure 5 - Physical structure Design Patterns

The composite pattern was used in the implementation of the physical layout structure. This pattern models a component as a leaf or as a composite. The leaf is the simplest component and the composite aggregates one or more components. A component is any element described in CIF [8]. Cells and layers are composites, while boxes are leaves. The use of the composite pattern permitted the use of a uniform structure, where each element is treated homogeneously, although having different meanings.

The algorithm performance depends on the structure used to model the data. The composite pattern hides the particular structure from the algorithm. A class TLayer exposes an interface supporting the manipulation of rectangles in a layout mask. Nevertheless, this class delegates the representation and the manipulation of rectangles to an internal structure. This structure can be implemented by several representations without affecting the algorithm. Structures like span [9,10], quadtree [11], corner stitching [12], etc can be experimented to the best fit.

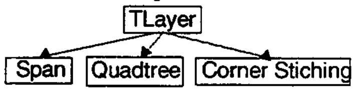

Figure 6 - Structures for mask representation

In order to perform any operation in the data structure it is necessary to traverse the lists of objects. This is accomplished by the use of the iterator pattern. The application of this pattern consists of creating an object that is responsible for scanning the compound object, without exposing its internal representation. Thus, for each component of the structure a TIterator object is created, that permits traversing the compound elements sequentially.

The visitor pattern allows the implementation of operations on the physical layout structure, which are not part of the element interfaces. The original layout structure only supports operations that modify itself by adding or removing elements. The visitor pattern allows the addition of new operations without changing the structure classes. Boolean operations (AND, OR...) necessary to the verification algorithms are added by the introduction of new visitors classes. The NOT operation, for example, consists in finding the complement of a layer, which is the set of rectangles corresponding to the empty spaces between the original rectangles. To perform this operation, a visitor is created to examine the layer and returns another layer that corresponds to the complement of the original one. Performing an operation on a data structure is accomplished by the use of the combination of an iterator and a visitor. The iterator permits the sequential traversal of the whole structure. Upon being visited, each element passes its reference to the visitor and calls the appropriated method to handle this type.

4. Distributed architecture

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

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

The distributed architecture consists in dividing the application in three modules: user interface, central control and server. For the user interface 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 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.

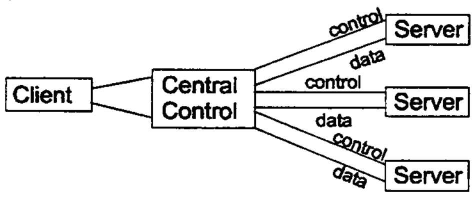

Figure 7 - three-tier dual channel architecture

In the concentrated version of the DRC, the XML script that describes the design rules is executed sequentially, in the specified order. 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. Graph nodes correspond to layers and pseudo layers. The analysis of the dependence graph depicted in figure 8 is useful to determine task scheduling. Let us suppose that there are five tasks to be executed, and only three computers available. The three tasks can be scheduled in a random manner or can follow the dependence 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.

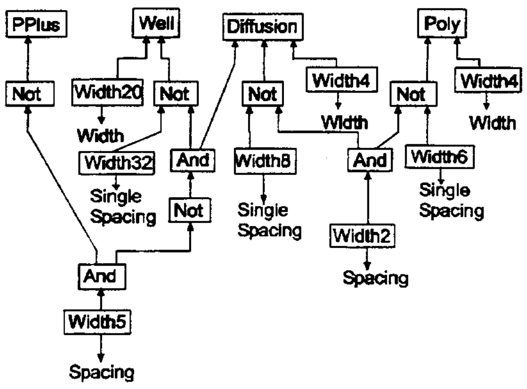

Figure 8 - rules dependence graph

5. Distributed data considerations

Distributed DRC execution implies data exchange between remote executing tasks. Starting a new task may depend on an 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, increasing completion time. Object transport involves memory image serialization, which consists of turning object contents into a data stream. Additionally, transmission of object streams is a costly operation.

The serialization process implies in an extra cost, which is 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 [13] can provide the logic necessary to this pipelined operation.

6. Conclusions

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 results from 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 use of a modular encapsulated structure can produce a robust and adaptable system.

7. References

1 Macpherson, K.; "Parallel Algorithms For Layout Verification", University of Illinois, M.Sc. Thesis, 1995.
2 Banerjee, P.; "Parallel Algorithms for VLSI Computer-Aided Design" - PTR Prentice Hall, 1994.
3 Ramkumar, B. and Banerjee, P.; "ProperCad: A portable object-oriented parallel environment for VLSI CAD", IEEE Trans. Comput.-Aided Des., vol.13, pp. 829-842, July 1994.
4 Belkhale, K. P., "Parallel Algorithms for CAD with Applications to Circuit Extraction", Ph.D. dissertation, University of Illinois, Dept. of Computer Science, 1991.
5 McLaughlin, B.; "Java and XML", O'Reilly, 2000.
6 Robinson, M., Vorobiev, P.; "Swing"; Manning Publications Co.; 1999.
7 Gamma, E., Helm,R., Johnson,R., Vlissides, J.; "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1998.
8 Mead, C., Conway, L., "Introduction to VLSI Systems", Addison-Wesley, 1980.
9 Nunes, R. B., Anido, M.L., Oliveira, C.E.T.; "A New Approach to Perform Circuit Verification - Using O(n) Algorithms" - Proc. 20th Euromicro Conference, IEEE Comp. Soc. Press, pp.428-434, 1994.
10 Nunes, R. B., Anido, M.L., Oliveira, C.E.T.; "Circuit Verification Using Spans - A Data Structure with O(n) Algorithms" - IX SBMicro, Rio de Janeiro, pp. 64-73, 1994.
11 Samet, H.; "The quadtree and related hierarquical data structures", Computing Surveys, vol.16, pp. 187-260, 1984.
12 Outsterhout, J.K.; "Corner stiching: A data-structure technique for VLSI layout tools", IEEE Trans. Comput. - Aided Des. , vol. CAD-3, pp. 87-100, Jan. 1984.
13 Brooks, P.; "Simple Buffered Collection and Buffered Iterator Patterns" - OOPSLA'95, 1995.