Institut für Informatik
der Technischen Universität München

Lehrstuhl für Informatik mit Schwerpunkt
Wissenschaftliches Rechnen

Hierarchical Pattern Matching in VLSI

Marko Milošević

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Tobias Nipkow Ph.D.

Prüfer der Dissertation: 1. Univ.-Prof. Dr. Hans-Joachim Bungartz

2. Univ.-Prof. Dr. Erich Barke, Leibniz Universität Hannover

3. Univ.-Prof. Dr. Thomas Huckle

Die Dissertation wurde am 25.03.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 06.07.2009 angenomen.

Abstract

Structural pattern matching is an important part of the microchip design verification process. It is necessary to isolate semantic structural contexts in a given design netlist in order to be able to perform flexible and intelligent checks like, for example LVS (Layout Versus Schematics), ERC (Electrical Rule Checks), gate level netlist timing analysis and others. Because of that, many different algorithms were devised to support this particular segment of chip verification. The theoretical basis for these algorithms is pattern matching in graphs, i.e. subgraph isomorphism. Algorithms developed so far are working with flat input netlists. This is not efficient and limits the application of the mentioned algorithms due to the flat netlist´s extensive size. Making the pattern matching hierarchical can improve the processes of chip design verification and simulation.

We provide the solution for the problem of the structural pattern matching in hierarchical netlists by defining the new methodology which employs the concept of Layered Views to present the hierarchical layout of a given netlist in a "friendly" way to an arbitrary application domain (user) algorithm. This general framework solves typical problems that algorithms working with hierarchical netlists are facing. Particularly, we propose the Virtually Flattened View (VFV), a sophisticated concept that prepares the hierarchical data for the user algorithms and allows them to see that data as if they were flat. We achieve this by materializing (creating a proxy copy) a small data portion which is kept consistent with the source hierarchical netlist by specific algorithms and data structures. The view offers the possibility to emboss the materialized data portion into the primary design's hierarchy, as a separate instance, altering the primary hierarchy. The outcome of this process is again a valid hierarchical netlist. We, further, apply the defined concepts to Incremental Pattern Matching, originally developed for flat input netlists only. In this way we obtain the methodology to solve the problem of pattern matching in hierarchical netlists.

For several reference scenarios, quantitative and qualitative improvements of our approach are demonstrated. The quantitative improvement is discussed through runtime and memory requirement tests. The qualitative improvement comes from the fact that the new methodology allows full-chip analysis and concise, hierarchical result reports.

Zusammenfassung

Der Verifikationsprozess integrierter Schaltungen beinhaltet eine ganze Reihe wichtiger Prüfungen wie LVS (Layout vs. Schematics), ERC (Electrical Rule Check), Statische Timinganalyse und andere, die flexibler und effizienter durchgeführt werden können, wenn der funktionale Aufbau der Schaltung der Prüfung zugänglich ist (und nicht nur eine rein transistorbasierte Netzliste ohne weitere Struktur vorliegt).
Aus diesem Grund ist eine strukturbasierte Mustererkennung, die es erlaubt, die für den Verifikationsprozess wichtigen Kontexte aus der Schaltung zu isolieren, ein wesentlicher Differentiator für die Qualität der eingesetzten Verifikationsprogramme hinsichtlich Performanz und Fehlerabdeckung. Dies hat in der Vergangenheit zu etlichen Aktivitäten in diesem Gebiet geführt, so dass eine Vielzahl unterschiedlicher Algorithmen und Implementierungen zur Mustererkennung vorliegt. Gemeinsam ist ihnen die Identifizierung von Mustern in Graphen, also die Erkennung von Teilgraphisomorphismen.

Die bisher entwickelten Algorithmen setzen flache Netzlisten ohne innere Struktur (Hierarchie) voraus. Das ist bei grossen Datenmengen nicht effizient und limitiert das Anwendungsgebiet. Gelingt es also, die Strukturerkennung auf hierarchischen Daten zu ermöglichen, so kann eine sehr grosse Verbesserung der Verifikationsperformanz erzielt werden.

In dieser Arbeit stellen wir eine Lösung für die hierarchische Erkennung von Mustern in hierarchischen Netzlisten vor, die auf der Einführung der neuen Technik sogenannter "Layered Views" beruht. Mit ihrer Hilfe werden die hierarchischen Daten den Applikationen auf eine sehr benutzerfreundliche und einfach zu nutzende Weise präsentiert. Insbesondere schlagen wir an dieser Stelle "Virtually Flattened Views" (VFV) vor. Diese präsentieren die hierarchischen Daten in einer Weise, die der Applikation erlaubt, sie zu interpretieren, als kämen sie von einer flachen Datenbasis. Typische Probleme, die beim Arbeiten mit hierarchischen Daten gelöst werden müssen, lassen sich auf diesem Weg einmal lösen, die Applikationen können in weiten Teilen unverändert von einer flachen Implementierung auf eine hierarchische Implementierung portiert werden, nur durch die Umstellung auf die Nutzung des VFV als Beispiel eines "Layered Views". Der VFV wird durch eine sehr lokale Ausflachung der hierarchischen Datenbasis implementiert, die dynamisch den Anforderungen der flachen Applikation entsprechend aktualisiert wird.

Auf diesem Weg können wir aber nicht nur die hierarchischen Daten lokal flach zur Verfügung stellen, wir können auch die Ergebnisse der Mustererkennung, die nun ja flach entstehen, ohne weiteres in die hierarchische Datenbasis unter Modifikation der existierenden Hierarchie zurückschreiben. Das Ergebnis der Mustererkennung ist also wieder eine hierarchische Netzliste. Weiter gehend wenden wir die neuen Techniken auf die inkrementelle Mustererkennung an, die ursprünglich nur für flache Daten implementiert wurde. Insgesamt gesehen haben wir damit das Problem der Mustererkennung in hierarchischen Netzlisten vollständig gelöst.
Für einige Referenzszenarios, die aus realen Industrieapplikationen stammen, demonstrieren wir die quantitativen und qualitativen Verbesserungen, die mit unserem Ansatz erzielt werden können. Die quantitativen Aspekte werden anhand von Laufzeit und Speicherverbrauchsvergleichen diskutiert. Die qualitativen Verbesserungen erzielen wir zum einen durch sehr kompakte (hierarchische) Ergebnisse, zum anderen können nun erstmals Netzlisten für das komplette Design bearbeitet werden, während vorher nur Teilausschnitte geprüft werden konnten.

Acknowledgemens

I thank my professor, Prof. Dr. Hans-Joachim Bungartz for leading me through this project methodologically and giving me self-confidence in crucial moments. I want to thank Dr. Martin Frerichs, Dr. Tilman Neunhöffer, Hannes Armruster and the rest of the ATS department of Qimonda AG for the substantial support of my work. I am especially grateful to Dr. Alexander Seidl for having the organisational side of my project in a perfect grip. In the end, I want to thank my family and friends for understanding and believing in me.

Marko Milošević

Contents

Appendix

Table of figures

1 Introduction

1.1 Motivation

The number of transistors that can be placed inexpensively in an integrated circuit has been increasing exponentially for more than four decades, confirming the observations and predictions of Moore's Law [1]. In fact, it has been doubling approximately every two years. For dynamic memory (DRAM) chips the growth in complexity has been even faster, as their capacity doubles about every one and a half years. Integrated circuit design must keep step with the increasing complexity. The fabrication setup process of VLSI designs is very expensive, as well. Each mask set that is necessary to be printed prior to massive production of micro-chips costs several millions of dollars. Further, the time needed for the development of a modern semiconductor product is critical. “Time to market” is typically given in very narrow windows. If one misses the optimal time to ship a new product, one also leaves the most of the revenue to the competitors. In memory production business, avoiding mentioned cost penalties is even more crucial as the margin in that business is very low. For given reasons no trial and error approach is allowed in order to prove the correctness of a design that is to be produced. Thus, verifying and proving that the design architecture is correct and feasible to manufacture in the given realistic technology prior to actual fabrication (achieving “first time write” principle) is utterly important.

The above stated requirements have coined numerous methodologies to model and check the IC designs. One of the central methods to fight the design complexity is employment of the concept of hierarchical abstraction. The overall development of electronic designs is colored by hierarchical approach, both from designing and building the schematics to the verification process. Different tools were introduced over time to support the verification process, thus to check the designs from various aspects. Depending on actual technologies the set of tools employed to perform the verification adapts and evolves. As the technology develops and inevitably shrinks to smaller scales different new problems related to the physical effects that could be neglected in the past emerge. In order to treat these new issues we have often new tools that get integrated into the design and verification methodology.

The fact that the designs are hierarchical shapes EDA tools. The tools can greatly benefit from the hierarchy as it offers completely irredundant view on a design, but to achieve that “oasis” an often big price mirrored in the required solution algorithm complexity has to be paid. In some cases this complexity is moderate and there are even tools that naturally benefit from the hierarchical representation, on the other hand there are tools for which the years of development are necessary in order to reach the stage where they can successfully employ the hierarchical concept. Making tools hierarchical can be seen also as one step in the tool evolution process. The typical development of the tool is driven by the importance of the check it performs and the complexity of the data that is verified. As the data which is the point of analysis constantly gets more complex and thus cumbersome, new and new solutions have to be integrated into the tool methodology to keep the effort spent to manage the data in acceptable range. The graph given in Figure 1.1-1 is showing the typical effort “waves” [2] that the tool/check experiences during its evolution. In the example we see that the check was at first performed manually, that was possible e.g. in the times

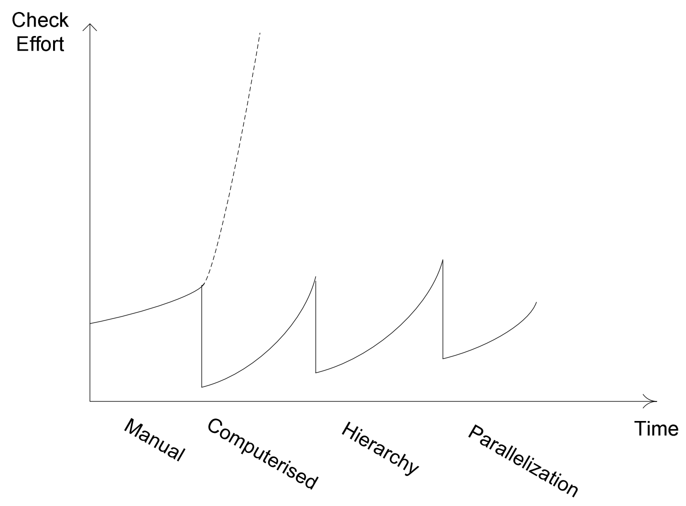


Figure 1.1-1 – Typical tool evolution curve

and cases when the designs were not having more than 100 transistors. As the data complexity has grown (exponentially), to prevent the exponential growth of check effort (given as a dotted line) a computer tool gets introduced. The tool in the moment when it is introduced brings amazing enhancement and we can notice the drop of effort to the values that are even smaller than the effort employed at the beginning of the evolution. Cycles like this repeat after each (revolutionary) improvement forming the wavy shape.

Depending on the evolutive stage, at the given current state of the art we have two kinds of tools: first that have reached the development stage to work directly on the hierarchical designs and the second which consists of the tools whose algorithmic implementations work exclusively on the designs that were previously flattened, thus simplified. The flattening process collapses all hierarchical levels and makes the model of the IC design whose layout is identical to that of a chip which is being printed into silicon. An additional class of tools (filtering tools) that enable flattening and other helpful transformations of the hierarchical designs have emerged, as well.

Today, to the first group typically belong important physical (design) rule checks (DRC) and layout vs. schematics (LVS) methods. Accordingly, in Electrical Rule Check (ERC) domain, where one checks the electrical correctness of the design, we have a lot of methods that are implemented so that they can benefit from the hierarchy, e.g. ESD checks, floating nodes check and device high voltage checks (where one checks if the given device in the design gets exposed to voltages it can’t withstand). As another example device reliability checks can be considered, where one adapts the device robustness to its duty cycle (the frequency of exposing the device to the stressed, non-conducting, mode). In all of these checks one does not need to have the broad information about the environment in which a given device is defined.

In cases where this is needed, the environment typically crosses hierarchical boundaries and is orthogonal to them. In these cases introducing the algorithm that works directly on the hierarchical data is far from trivial. In some cases the solutions for these problems were found, like in mentioned DRC checks, but still, as we have pointed out, in most of the cases we perform the algorithm on flattened netlists. One of the examples is the parasitic extraction problem. The dependencies between parasitic nets are typically cross-hierarchical and instance dependent.

Another, for the motivation of this thesis the most important, example of the tools that work on flattened schematic data are those that employ structural pattern matching in electronic designs. These tools are frequently used in ERC, but also can enhance different other areas of EDA. In some sense structural pattern matching makes the schematic design processing more intelligent. By employing the matching one can become aware of the context in which a given design device is used and thus gain additional power to optimize the given device or to analyze its configuration with greater precision finding unwanted incorrectness.

One important realistic application of this process is the ERC check where one sets the proper dimension of the drivers of the latches in the electronic circuits. The ratio of the parameters of the transistors that are members of the given driver circuit have to be adjusted to the driven circuit load. This is a task suitable for pattern matching. One can identify all latches in the electronic circuit and than find their corresponding drivers. After this step one can compare the parameters of the transistors identified as driver building elements to the requirements that are imposed to them and adjust them. As this adjustment is highly specific to the given instantiation place and the designers that employ hierarchy and build different contexts out of generic parts (predefined subcircuits) can't be aware of all quantitative aspects easily, the benefit of a tool that pin points incorrect configurations is vital.

As we have mentioned one of the prerequisites (a pre-processing step) for today's state of the art structural pattern matching for IC circuits is assuring that input schematic designs are flat. This is present throughout the community for, to our knowledge, all available solutions.

This approach introduces several disadvantages. First, the size of the flat design can't be even compared to the hierarchical and it overwhelms the typical resources of today's computers. If the analysis is still possible the memory requirements are then typically so high that more expensive 64-bit machines are required and the corresponding runtime becomes an issue, too. One of the most challenging problems that comes as a consequence of the fact that the transformed (flattened) design is used is back-annotating the results to the original schematics. This can be difficult as, by working on the flat netlist, we obtain redundant results that are over-bloated and hard to compare (and find out that they are actually coming from different occurrences of the identical subcirucit of the hierarchical designs). The described problem creates additional time demanding analysis activity (man power) of the tool user and disables the automatization of the process and its integration to modern hierarchical design development environments.

For that reason, there is a need to enhance the structural pattern matching process and solve the algorithmically very demanding problem in order to allow performing of that task directly on the hierarchical schematic designs. Similar problems to those that we have pointed out in the above text are present in all of the tools that are, at the common state of the art, performing checks on the flattened netlists.

1.2 Objectives and scope

Our main goal is to achieve the algorithmic solution for the problem of structural pattern matching in hierarchical designs. Since the complete proven solution(s) for pattern matching problem in flat IC designs already exist and also other tools that were written to work on flat data share some similar obstacles which disable them from running hierarchically, we want to try to find a common solution that could be applied to any flat algorithm. For this reason, we have decided to search for our solution directly in the database which prepares and exposes the design data to the client application. We want to upgrade the standard database presenting abilities by allowing views which make the hierarchical organization of the given design relative.

The fact that the modern, standardized EDA databases are typically object-oriented gives us a beautiful chance to include advanced and very useful concepts that the object orientation brings in our solution. Hence, for reading and understanding of this thesis one needs to be familiar with advanced object-oriented concepts and UML notation language, which is the most suitable and in the same time the most general way to describe different aspects of the object oriented concepts. Furthermore, our solutions will include different design patterns that make the solution more robust.

The expected results are at first, the functional correctness of the model that needs to present the data to the user algorithm in a (friendly) flat way and keep it consistent with the original hierarchical data. We further expect that the upgraded originally flat structural pattern matching tool run generates irredundant results after, by our contribution possible, precise calculations directly in the place where a given topological context has been defined (relative to the specific subcircuit). Additionally, we expect better runtime of the pattern matching application and more economic memory consumption. This is challenging as the problem of structural pattern matching to which we want to apply the model that presents the hierarchical data in a flat fashion is NP complete. Taming these two parameters should push the border of the size of the designs that are manageable towards today’s full chip sizes. That fact puts one into position to run the corresponding checks in realistic application cases in sublinear times (sublinear concerning the flat netlist size). We expect that the algorithm complexity depends rather on the hierarchical than on flat design quantities.

One additional important quality that we want to achieve is to use the existing successful pattern matching industrial project completely transparently with the upgraded database and that the solution we propose is possible to be used with no or small corrections with other tools that favour flat to hierarchical netlist representation. Note that possible corrections of the solution that would be applied to other tools would also be a continuation of this research and would contribute to the evolution of the hierarchy transforming (hiding) data presenter.

We will apply the proposed solution to an industrial project, the pattern matching tool - “classify”, which implements the incremental structural pattern matching principle (studied by several groups) and experimentally check our expectations and value the benefits that the proposed approach brings. In order to do that we will employ real-life industrial test cases that are thoroughly quantified, so that we can gain confidence and better understand the performed tests’ outcome.

1.3 Outline

Chapter 2 of this thesis gives an overview of the state of the art flat graph pattern matching algorithms, for the application in EDA CAD. Further, it presents the enhancement of the incremental search oriented graph pattern matching algorithm that was developed by Qimonda AG and the Institute of Microelectronic Systems (IMS), at the Leibniz University in Hanover. Moreover, the adaptation of the mentioned algorithm that prepares it for the hierarchical usage is given. This proposed solution resolves the problem of supply nets (extensive time is needed to search for the pattern whose potential target circuit image includes supply nets), common for different algorithms which solve the problem of structural pattern matching.

The concept of hierarchical abstraction is discussed in Chapter 3. We present the wide application of the hierarchical concept in nature and science, with the accent on its application in EDA CAD. Thus we present the formal model of the folded encapsulated hierarchical graph. Moreover, the history and evolution of EDA databases which implement the folded encapsulated hierarchical graphs and the overview of an example EDA database (the case study of a logic domain database) are given. Using the example database we discuss different hierarchical algorithms and abstract data structures which typically used by the tools which analyse hierarchical data. The chapter is concluded with a vision of flexible views on the hierarchical data after it was pointed out that the hierarchical layout of the given data model is not unique, but polymorphic. By polymorphism here we mean that a given flat design can be represented by many hierarchical netlists which are then synonyms.

Chapter 4 brings the general concept of the layered views on the hierarchical design data. View's architecture and layering technique are discussed. The visions of possible example applications are given, as well. One of the visions of the hierarchical views is the view that hides the hierarchy in order to represent the local data portions that appear to be flat, bringing all the devices employed to the same level. We call it virtually flattened view.

In Chapter 5 we demonstrate the detailed implementation of the virtually flattened view, following the view architecture defined in Chapter 4. The chapter starts with the explanation of the high-level, object-oriented architecture of the view, followed by detailed presentation of each of its conceptual parts. In this chapter, a set of unique data-structures that enable the view creation and maintenance are explained. We present the novel context saving tree and the embossing process that alters, just locally, the layout of the hierarchical design, adding the flat view as the separate, new subcircuit. Additionally, we present the covering technique, which is used for fast changes of the topology of different design subcircuits, affected by the embossing process. This technique is crucial for fast VFV algorithm runtimes.

In Chapter 6 the application of the Virtually Flattened View is given in order to solve the problem of hierarchical pattern matching, together with the case study that serves as the evidence of the qualitative and quantitative achievements of the new approach. Therefore, the process of adaptation of the generic VFV to the application domain is explained. This is achieved by creation of a specific hybrid layer that flavours the generic classes of VFV with the properties needed for pattern matching. In the case study we analyse the quantitative and qualitative aspects of the VFV application in incremental structural pattern matching.

Chapter 7 summarizes the results of the overall research.

2 Graph Matching Concepts in VLSI

In this chapter we are going to give the necessary theoretical background for the problem of structural recognition in VLSI. We present the importance of this concept with its application. Throughout sections 2.3, 2.4 and 2.7 the overview of the field development is given as well. Details about the problems, strategies and solutions that favour the understanding of the thesis are apostrophed with more thorough descriptions in section 2.4. In this section we describe the concept of incremental pattern matching as a solution for subcircuit recognition (SR) problem. We point out the importance and the benefits this approach brings, but as well isolate the problems the realisation of the concept has faced during the years of real-life industrial application. In section 2.6, we propose the algorithmic solution for the performance problem of the incremental pattern search engine. In some realistic application cases it was demonstrating indeterministic complexities. We conclude the chapter (section 2.8) with the analysis of the further development directions of this field, particularly the need of enabling SR algorithms to work on hierarchical input netlists. Thus, this chapter serves also as a realistic foundation that justifies and settles the motivation to develop the general pattern matching algorithm for hierarchical chip designs.

2.1 Basics of graph notation

In order to explain the algorithms for graph matching it is necessary to formally define the notation of graph [3]. In general, *graph* G represents a pair of two sets, V and E, G = (V, E), such that $E \subseteq V \times V$. The elements of the set V are called *vertices* and the elements of the set E *edges* (also known as *lines).* It is common and convenient to represent graphs with the graphical notation where vertices are drawn as dots and the edges as lines connecting them. An example of such a structure is given in Figure 2.1-1. The graph has in total six vertices. There exist four edges. In the figure we represent the graph formally, using sets and as well graphically. Note that in the graphical notation for a graph any shapes are not of interest, but just logical connections between the given entities.

G = {V,E}
$V = \{ v_1, v_2, v_3, v_4, v_5, v_6 \}$
$E = \{\{v_1, v_3\}, \{v_1, v_2\}, \{v_3, v_4\}, \{v_1, v_5\}\}$

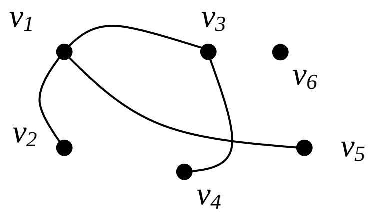


Figure 2.1-1 – Example of the graph notation. Black dots represent the vertices and the lines the edges.

We can define several quantities to measure graphs. For instance, the number of vertices defines the *order* of the graph. The order of graph is determined as $|G|$, additionally the number of edges is usually determined as $\|G\|$ (alternatively, the corresponding orders can be determined with $|V|$ and $|E|$, respectively). In respect to the order graphs can be finite, infinite, countable, etc. The graphs we use will be always finite. The order of the example graph is: $|G| = 6$.

The graph without vertices and edges is an empty graph (∅,∅). We simply write shorter: ∅. The vertex v is said to be *incident* to an edge e if $v \in e$. On the other hand, the two vertices incident to an edge are called *ends*. We say that two ends of an edge are *adjacent*.

We define, further, the *degree* of the vertex: d(v). The degree of vertex equals the number of edges which it is connected to. For instance, the vertex v_1 (Figure 2.1-1) has the degree d(v_1) = 3, while the vertex v_6 has the degree d(v_6) = 0. The vertex with the degree value that equals zero is also known as the *isolated vertex*.

We use graphs to model different complex entities. Their atomic parts and connections can have properties. Therefore, we define properties also for the model (graph) elements. Vertices and edges can contain information. If the information is trivial and each atomic part has a label, we say that the graph is a *labelled graph*. The information can be as well more complex and each of the graph elements can contain a set of *attributes*. In that case we have an *attributed graph*.

It is possible to classify graphs according to the values of the defined properties. For this thesis it is important to define a class of graphs where the edges are restricted in a specific way.

Let r ≥ 2 be an integer. The graph G = (V, E) is r-partite if the set V can be divided in r partitions such that no edges exist between the vertices to the same given partition of the set V. Therefore an edge is allowed to be placed only between the vertices of different partitions. A special case of this class of the graph is 2-partite or *bipartite graph*. The vertex set V of these graphs can be divided in two partitions, $V = X \cup Y$, in which the vertices have no mutual connections, or formally: $\forall e \in E : e = \{x, y\} \therefore x \in X \wedge y \in Y$. We show an example of such a graph in Figure 2.1-2.

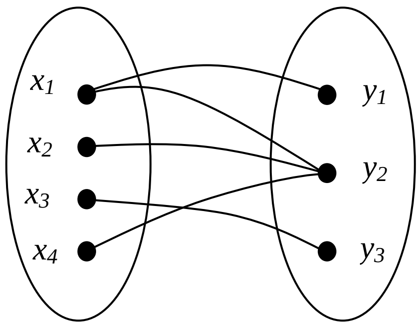


Figure 2.1-2 – bipartite graph

Bipartite graphs are widely used for pattern matching in the area of chip design verification. They are particularly used in the area of EDA to model the electronic circuit.

We will further define a *hyper graph*. A hyper graph is the generalisation of the graph where the edges are not exclusively connecting two vertices, but a set of vertices. Formally, a hyper graph H is a pair $H = (V,E)$ where V is a set of elements, called *nodes* or *vertices*, and E is a set of non-empty subsets of V called *hyper-edges* or *links*. Therefore, E is a subset of P(V)\ ∅, where P(V) represents the power set of V. The power set is the set of all possible subsets of a set. While graph edges are pairs of nodes, hyper-edges are arbitrary sets of nodes, and can therefore contain an arbitrary number of nodes. One example, together with the graphical notation of the hyper graph is given in Figure 2.1-3. This example is similar to our general graph example from Figure 2.1-1. It has as well six vertices still, as we can see the connections group more than two vertices together. In the graphical representation the edges are given as overlapping areas rather than lines.

$H = \{V,E\}$
$V = \{ v_1, v_2, v_3, v_4, v_5 , v_6 \}$
$E = \{\{v_1, v_3\}, \{v_1, v_4, v_6\}, \{v_2, v_4\}, \{v_1, v_5\}\}$

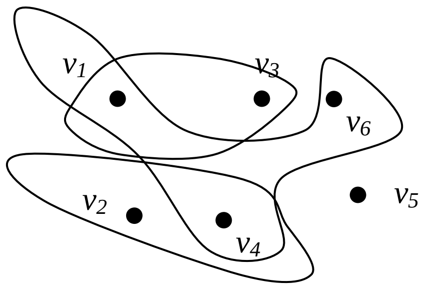


Figure 2.1-3- Hyper graph

Hyper graphs are an alternative for modelling the electronic circuits. They are suitable due to the fact that the electronic circuit elements often have a big number of multiple mutual connections and the edge concept of the hyper graph allows that.

2.2 Graph matching

As it has already been pointed out, different applications of the graph theory request comparing two graph structures. This was a motivation to develop a family of graph matching algorithms. Graph matching can be formulated as follows: given two graphs (pattern graph)$G_p = (V_p, E_p)$ and (target graph) $G_t = (V_t, E_t)$, find one-to-one mapping f: $V_p \rightarrow V_t$, such that $(u,v) \in E_t \Leftrightarrow (f(u), f(v)) \in E_p$.

Traditionally the first group of algorithms that were developed to solve the graph matching problem were search oriented algorithms. Typically they require a rigid identity between two structures that are compared. For this reason they are also known as *exact pattern matching* algorithms. Traditional, search oriented (exact) methods are based on recursive breadth first or depth first search (with backtracking) from the selected candidate starting point inside the target graph. Therefore, for different starting places the algorithm attempts to test the environment of a given vertex for the isomorphic structure. However, different useful heuristics made these methods powerful and tuned for appropriate applications. The heuristics typically take advantage of the specific information which the graph nodes, as models of the application area entities, carry. Different preparation processes are done in order to achieve typically linear runtimes. The specific algorithms are therefore developed exactly for certain type of graphs that they are analyzing.

On the other hand, the group of algorithms developed chronologically later can accept also more relaxed requirements concerning the resemblance between the pattern graph and the target graph. They are error tolerant. In this case, alternative approaches are used, such as optimization theory, neural networks, genetic algorithm, probability theory, etc. These algorithms are performing *inexact pattern matching*. We can now shortly summarise: exact pattern matching algorithms are optimal from the angle of accuracy, while inexact pattern matching algorithms are fast, but approximate and are not 100% reliable.

Different research groups have developed particular algorithms from both classes. From the search oriented class we will mention the algorithm of Ullmann [4] where a greedy heuristics is applied. The algorithm continues the recursive search choosing the path which satisfies the set of local statistical constraints. On the other hand important is the approach in the algorithm of Corneil [5], where the graph gets globally partitioned to prune the number of appropriate candidate starting points. An additional approach in this class is the graph matching using Binary Decision Diagrams [6]. In the area of inexact algorithms various theoretical concepts are applied. We will apostrophe the approach based on optimisation theory and statistical physics – graduated assignment [7, 8]. This approach adapts the optimization function of the general graph matching problem by developing it as a discrete Taylor series and reduces it to the assignment problem. In contrast to graph matching that is NP complete [9], the assignment problem has a known polynomial complexity solution: the *softassign algorithm* [10]. Graduated assignment led by graph labelling with good discriminatory properties has shown respectable results.

Both algorithm classes have found applications in different areas. Newly developed techniques for inexact pattern matching, replace traditional search oriented algorithms in the domains where the speed is essential and, more important, where the target graphs are expected to be just an approximation of the pattern graph. In general this is used for image recognition applied in different areas such as: character recognition, computer vision, GIS (Geographical Information Systems) and medicine.
Exact pattern matching algorithms are still used in areas where the complete accuracy between two graphs is essential.

Exact and inexact pattern matching both define two sub-problems (Figure 2.2-1). We can compare two different graphs in order to prove if they can be matched. Additionally, it is possible to check if a given graph is contained in another graph. This is a problem of subgraph matching, which can be defined as follows:

Given a graph S and a larger graph T, find all the subgraphs of T that are equivalent to S.

The subgraph isomorphism problem is computationally far harder than the graph matching problem. Although both belong to the class of NP complete problems the graph matching can employ different global statistics when comparing two graphs that can simplify building effective heuristics. This kind of simplification is not possible in subgraph isomorphism problem.

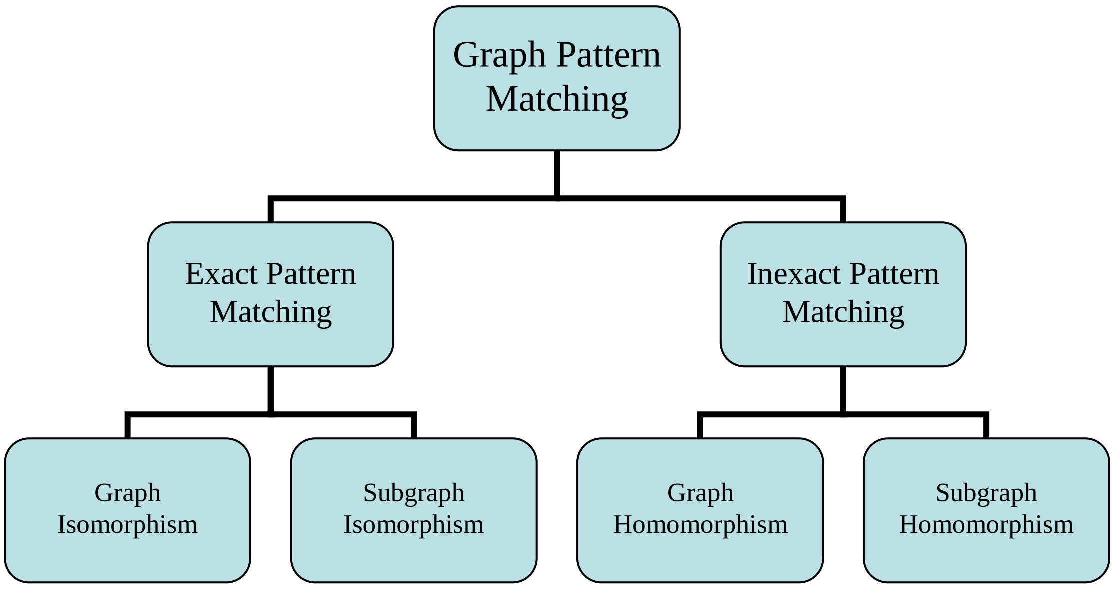


Figure 2.2-1 – Classification of the pattern matching in graphs into two general groups: Exact pattern matching and inexact pattern matching.

2.3 Subcircuit recognition, the application of subgraph matching

The application of pattern matching in EDA is vital for different parts of the design verification process. The application domain is called subcircuit recognition (SR).

It is useful to recognise some meta-structure from the groups of interconnected devices in many application scenarios. We can, therefore, use SR to understand the semantics of certain devices. The role of an identical transistor is not the same in an analog circuit and in some logic gate. During the chip design verification process it is necessary to compare the netlist which is extracted from the layout to the original schematic netlist. This process is known as LVS (Layout vs. Schematics). In case where the netlist extracted from the layout is flat (composed exclusively from atomic elements) the benefit of SR is obvious. We can isolate the hierarchical structure and compare it to the original hierarchy of the schematics. Still even if the netlist extractor is hierarchical, the hierarchy of the extracted netlist is usually slightly different, so rebuilding the original functional blocks (that exist in the original hierarchy) is necessary to prove the identity between these two netlists. On the other hand the (from layout) extracted netlist sometimes contains some extra physical characteristics of the design which are modelled as parasitic interconnection networks, the passive RLCK networks that realistically model the dynamical behaviour of the device interconnections. By applying SR we can isolate parasitic networks that can be later evaluated (acquiring statistics important for timing characterisation) or reduced [11, 12]. SR can optimise the simulation too. Certain parts of the design that are structurally expensive to simulate and whose internal states are not of interest can be recognised and abstracted as behavioural models, or just simplified models (we come back here to parasitic network reduction).

Last but not least, SR is enabling static timing analysis of the custom transistor level design. We can abstract each implementation of the logic gates as a separate subcircuit, perform the timing characterisation on the given block definitions and further analyse their interconnections in the produced gate level netlist. Gate level netlists allow also the functional verification of the netlist. Thus, by SR we make the functional verification of the transistor level design possible. There are numerous other possible applications for the SR in the area of EDA. Simply, SR makes the verification process more intelligent and context driven. All of these reasons strongly justify the thorough research and the development of the general, flexible and powerful SR strategies.

It is common to model the electronic circuit as a graph. For different purposes different types of graphs are used. For the purpose of subcircuit recognition (SR) the application of bipartite graphs (section 2.1) is common.

As we have shown the vertices of the bipartite graph are divided in two partitions (sets). In the domain of SR one set of vertices models the devices (transistors, resistors, capacitors etc.), or in other words *elements* of the electronic circuit, while the other models the interconnections between the devices. These interconnections are called *nets*. A net is an optimal way to represent a connection between arbitrary number of devices.

This is due to the fact that the representation of the common connection between n elements by direct mutual referencing would demand $n \cdot (n-1)$ references

($\frac{n \cdot (n-1)}{2}$ edges), while employing the special class of the vertices that model the connection requires $2 \cdot n$ references (n edges). This can be illustrated with a simple example given in Figure 2.3-1. In the figure, we have a complete graph of four vertices. In order to interconnect its vertices we need six edges. If we introduce an additional vertex class (shown white in the example), we need only four edges to represent the same connections between the graph vertices.

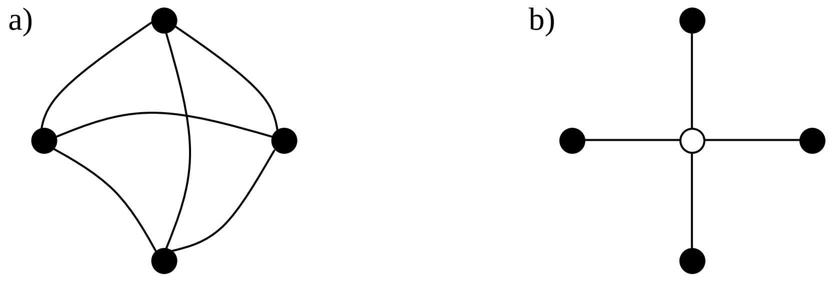


Figure 2.3-1 – a) standard graph with one class of vertices. b) bipartite graph, where connections are modelled separately as a vertex class, shown as a white circle.

Interconnections between the multiple devices are very common in electronic circuits. Think of the supply connections (power and ground connections). Millions of devices are all connected to a single supply interconnection. Representing them with the strategy under (a) would require dramatically more space than the bipartite graph strategy. We can say that the memory requirement complexity of the first strategy has the complexity O(n^2), while the second has O(n).

Both classes of vertices typically have a type defined with them. Devices are typed simply by the kind of the entity they model. Nets can be typed by the semantics of the signal they are carrying. This kind of typing is important for different algorithms that are interpreting the electronic circuits which are modelled by graphs, but not necessary for the storage of the design alone. One can broadly distinguish supply connections and signal connections. As each circuit that is modelled by the graph has nodes which model active devices (i.e. transistors), they would be typically connected to some power source, having part of their terminals on a constant potential.

To summarise, the model of the electronic circuit falls (typically) into a class of bipartite attributed graphs. We can consider the example of the bipartite graph rep-

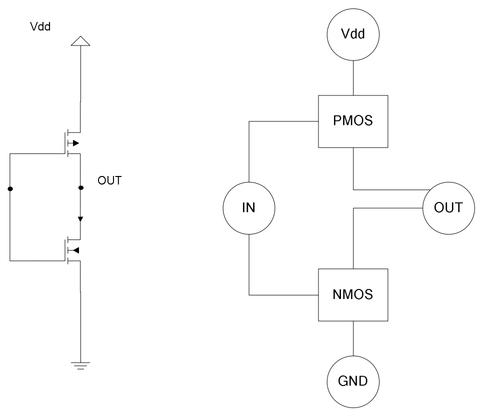


Figure 2.3-2 – Bipartite graph representation of an inverter circuit, realised in CMOS technology.

resentation of an inverter. Figure 2.3-2 shows the schematic representation of the inverter together with its graph model. The devices are represented by squares while the nets are given as circles. By mapping the theoretical model of a graph to the electronic circuit, we can now exchange our vocabulary and use the terminology of the EDA equivalently with/instead of original terms from the graph theory. Using this new vocabulary we can now (re)define the problem of subcircuit recognition.

Subcircuit recognition isolates the instances of a specified *pattern circuit (*or simply *pattern)* inside the larger *target circuit.* The example of this problem is given in Figure 2.3-3. In the given example we have a pattern which defines structurally a functional circuit NAND (a). The circuit is built up from the proper link of the parallel connection of PMOS transistors and serial connection of two NMOS transistors. The instances of this pattern are being searched in an example of the target circuit, given in (b). That target circuit contains the image of the pattern and it is marked with a dotted rectangle. Note that the pattern has also defined node types. Not any conglomeration of transistors connected in the similar way like the NAND pattern will than lead to a match. It is necessary that the source terminals of the PMOS transistors are connected to the fixed supply voltage (Vdd) and that the source terminal of the NMOS is grounded.

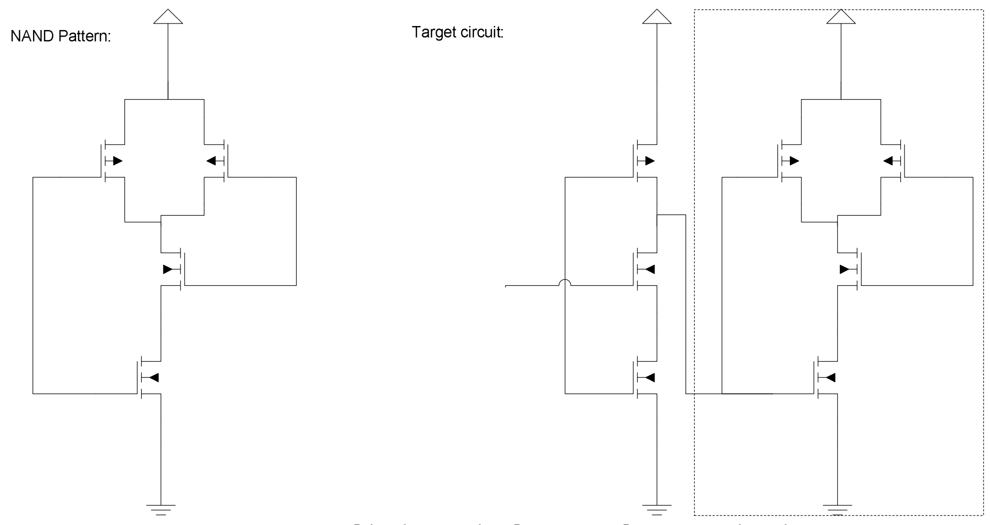


Figure 2.3-3 – NAND pattern and its image in the example target circuit.

This fact is crucial for the correct matching but as well of a big help for the algorithms that were devised to serve like an engine for this problem.

After a pleiad of technology dependent algorithms, where the patterns and different approaches to match a given subcircuit were hard-coded into specific functions, more general tools arrived, based on subgraph isomorphism. Since the subgraph isomorphism problem is in general NP complete and the designs on which EDA tools are to work on are complex with the trend of increasing that complexity, more general and always more efficient solutions were searched for. Different solutions transfer and adapt the known strategies from the theoretical field of graph matching, enriching them with the domain specific heuristics.

The heuristics are led by the sparsity of the graphs that represent integrated electronic circuits. The topology of the bipartite graphs that model circuit designs has some typical properties. For instance, as we have mentioned the number of device classes is limited. They all have terminals which connect them with other devices. The number of terminals is strongly limited as well, typically not bigger than four and each of the terminals has its semantics. Nets are not limited by the number of incident device terminals. Still, typically, the design consists of a lot of small local nets opposed to several big supply nets, or some signal nets with a big fan-in or fan-out factor, depending on the circuit semantics.

Specifically for the pattern we can distinguish two kinds of nets. *Internal nets*, which are strictly connected to the devices that exist in the pattern and *external nets*, which can "communicate" with the rest of the target circuit. They actually connect the pattern image to its environment (in the target circuit). For instance, the net between two serially connected NMOS transistors in our example pattern is internal. The nets that connect gates of the corresponding PMOS and NMOS transistors (the input nets) are logically external.

The heuristics of the algorithms that were developed is typically led by these basic properties of VLSI electronic circuits. The algorithms which employ the corresponding heuristics to favour the typical properties of the VLSI electronic circuits are typically linear. Still, unfortunately, the heuristics do not grant the linear complexity. In some cases even the tuned depth first and breadth first search algorithms demonstrate indeterministic complexities (towards the exponential worst case complexity defined by the general theory of NP complete problems).

One of the first algorithms and the project that tends to define the general tool for subcircuit recognition in electronic circuits is the approach of Lüllau et al. [13]. This group has devised a specific partitioning algorithm that labels each device or net in the circuit with a specific integer number. This number depicts the immediate neighbourhood of the given device. The most interesting fact of the labelling algorithm is the application of prime numbers. Each device type, or device terminal type (for instance in a transistor drain source or the gate) is coded by a distinct prime number. The overall label of the device than is the number that is obtained by multiplying the codes of the adjacent device terminals to it. The algorithm uses an abbreviation of the bipartite graph (multi-place graph). In this graph model apart from the set of vertices we define the set of *spiders*. Spiders correspond to edges and nets together (where the net is the *body* of the given spider and the edges are its *legs*). This representation reminds also of hyper graph. Therefore the label is given to the spider instead of a net of the bipartite graph. The label of a spider is the product of the labels (prime integers) of all its legs. The important property of a label which is obtained by multiplying prime numbers is that one can easily test if the certain combination of device terminals is incident to the given device (spider) simply by dividing the device (spider) label by the given "sublabel". If the labels are dividable without residuum the test is considered successful. In this way the algorithm saves a lot of time that would be spent for the unsuccessful tests, just by one arithmetic operation. The authors claim the expected linear complexity.

Several other algorithms exploit the idea of labelling the pattern and the target circuit that originates in the Corneil's algorithm. The algorithm that further develops the application of this idea in the area of SR and has achieved the respectable linear complexity in most of the application cases and in the same time became one of the most referenced algorithmic solutions for the SR problem is the algorithm of Ohrlich et al. - Subgemini [14, 15]. The pattern and the target circuits in this approach are modeled as the bipartite graph. This algorithm defines two phases. In the first phase the labeling algorithm conceptually similar to Lüllau (Corneil) is being applied. This algorithm achieves non-local labeling of different nodes with respect of their neighboring topology in the target and pattern circuit. This enables it to achieve extensive pruning of the search space and to isolate a typically short candidate vector of possible instantiations of the pattern, from which the algorithm proves if the target circuit contains the match or not by the breadth first search. Subgemini authors show the efficiency of this heuristic method by tests and still point out its weaknesses. First is that the algorithm is unable to match any pattern image circuit with shorted external nets. This is simply due to a fact that in the bipartite graph any shorting of the net is equivalent to net merging. That means that the image of the pattern circuit with some external nets shorted has than less nets that the pattern itself. This automatically leads to the fact that Subgemini fails to identify the given pattern instance. Another problem of Subgemini is the fact that any target circuit matching process that includes evaluation of a supply net is experiencing long runtimes. This problem comes from the fact that all active devices (MOS transistors, Bipolar transistors) require the power supply which means that a pair of its terminals is always on the common high voltage and the ground. In bigger circuits this leads to very large nets whose analysis (linear search) always implies long runtimes.

Different groups have worked on the problems that Subgemini has faced. One interesting solution for the shorted external net problem of Subgemini is given by Ling [16, 17]. Shortly after the publication of Subgemini he points out the problem of shorted external nets and offers the solution by transforming (upgrading) the bipartite graph with some of the properties of the hyper graph. He introduces specific *edge units* (EU). Each EU describes the connection between two device nodes (over a net). If we observe a shorted net in this way we can conclude that the set of EUs in the shorted net is the superset of the EUs of the non-shorted external net. This means that the Ling´s algorithm can find as well the instances with shorted external nets.

2.4 Incremental pattern matching

As we have described, different groups have worked on enhancing the matching process in order to optimize the solution to the problem of SR. In parallel to these inventive heuristics an additional approach has been developed. This approach is actually the upgrade of the atomic SR problem, where one locates the images of the given pattern circuit inside the target circuit. In *incremental pattern matching* we connect the outcomes of single atomic matches and use them as premises in order to isolate higher level complex contexts inside an electronic circuit.

In order to illustrate the core idea of this SR strategy we will go back to our NAND example. If we want to match the NAND pattern circuit in the target circuit incrementally, we can divide the process in three steps. At first, we match the simple parallel connection between two PMOS transistors, which is the standard SR, described in the previous section. After this we match the serial connection of two NMOS transistors. If the tool for matching can, after locating the image of the pattern in the target circuit, alter the topology of the target circuit inserting new solid abstraction in place of the recognized topology, we could use now these “intermediate” matches in order to isolate the final context, in our case a NAND. In Figure 2.4-1, we see these three patterns and all places where the matches for them occur in the target circuit marked. Therefore we match at first the parallel connection once. Note that in the target circuit, there are two occurrences of the serial NMOS connection, but only one of them is, together with the matched image of the PMOS parallel connection, forming a proper NAND gate.

The given strategy has a number of advantages. First, if one wants to match difficult contexts that are composed of many elementary devices it is much more natural to first detect smaller functional parts of the given context, and than to match it on the higher level. This approach is easier to understand and to explain and in the end it is easier to write the corresponding patterns. Further, since we match some more complex patterns step by step, employing patterns that have usually not more than two elements, we obtain shallow backtracks, no matter if we are using breadth first or depth first search. This brings faster execution times especially in highly symmetrical circuits.

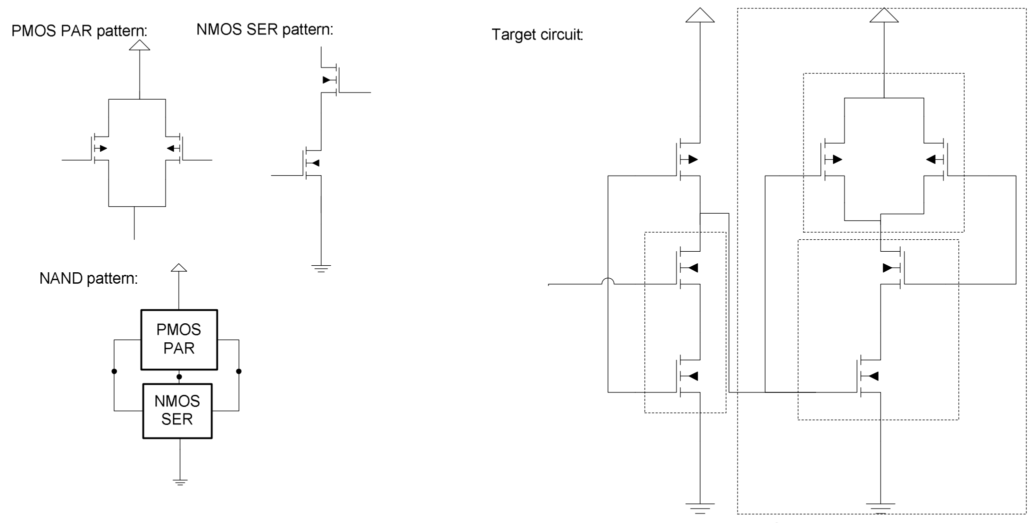


Figure 2.4-1 – Incremental pattern matching. Three patterns are defined: Parallel connection of two PMOS transistors, serial connection between two NMOS transistor and the proper link of these two high-level abstractions forming a NAND pattern. Additionally the example target flat circuit is shown with marked places of the instantiations of the mentioned patterns.

Since we have now specified the fact that a group of matching processes is interesting to us and we want to observe it as a whole, let's try to define the structure of this group and its elements.

An atomic entity of the incremental pattern matching process in SR is a *rule*. A rule corresponds to the single SR process where after the structural *pattern* is matched an *action* is performed. We can therefore say that the rule is built out of the structural pattern and the action which is executed if the image of the pattern is found.

It is useful to represent the pattern as some kind of graph regular expression, to make it templated. In this way a single pattern can match a family of structures. This property elegantly solves some known problems of SR, such as the problem of shorted external nets of the structural pattern. In general it can help matching different generic circuits or circuits done in similar technologies with a single rule. On the other hand allowing the pattern to be rich in templated mechanisms makes the implementation of actions more complex [18]. For this reason it is necessary to carefully choose the set of templated properties that would make the best compromise between the implementation complexity and the ease of application.

There can be several types of actions. Possible actions can include evaluation of the given image of the pattern or modifications of the topology of the target circuit. The modification can, for instance include exchange of the matched star of resistors with a triangle. The special case of modification that is crucial for the incremental pattern matching is the action of *abstraction*. In this case we simply form a solid block that stands for the given pattern image. This block is connected with the rest of the target circuit by the pins that are analogue to the external nets of the pattern.

Once we have defined the rule as the atomic part, we define a rule *sequence*. In our example we have used a sequence of three rules in order to match the NAND logic gate.

In order to make the process of matching more powerful we can introduce also a flow control to control the order of rule execution and make it generic (to react to the outcome of the single rule matches). In this way we combine the pattern matching rule serial sequences with conditional or unconditional loops and branches. By defining flow control we introduce a specific descriptive *rule language*.

This kind of language was for the first time introduced by Chanak in his PhD thesis project [18]. The project of Olbrich/Barke [19] also defines the specific descriptive language, developing the idea of Chanak. Their language that has been named Clarula (classify rules language) will be subject of the following section.

There is nevertheless one known alternative to the language which controls the proper execution of the incremental pattern matching.

Pelz et al. [20, 21] were motivated by the LVS process and have proposed a specific pattern matching algorithm with *hierarchical patterns*. In this approach the pattern has its own hierarchy and the order of matching (executing SR algorithm abbreviation based on depth first search) is determined automatically by the specific algorithm that analyses the hierarchy of the pattern. Pelz introduces the pattern as the generalization of the problem he has analyzed. His goal was to prove if the hierarchical schematic netlist is identical to the flat netlist extracted from the layout. The order of matching is chosen in the bottom-up fashion, logically. At first the most elementary patterns would be recognized and than their results used to recognize further higher level pattern towards the top. Pelz determines the constraints for the hierarchical pattern topology. The preparation algorithm would analyze the hierarchy of the pattern and alter it if it finds the violative properties. These properties are for instance existence of a given topology both as a separate abstraction (lower level pattern) and as a flat topology in the same hierarchical level of the pattern. For instance the hierarchical pattern of the latch has one inverter given as a separate subcircuit and another as the proper connection of CMOS transistor pair. This kind of a pattern would never match. Pelz's preparatory algorithm resolves this violation in the given hierarchical level of the pattern by flattening the given abstracted topology or abstracting its flat version. Another constraint is the existence of flat match of the pattern that is distributed between two hierarchical levels of the input circuit. This is being checked by flattening the pattern and than trying to match it. Pelz further identifies the problem of reordered pins of the pattern abstraction and the problem of technological difference of two functionally identical parts of the extracted netlist. He solves the latter problem by introducing a specific library of patterns that are of the equal type, but have the different, alternative implementations.

All of these problems still exist in the case when we define the flow control of the incremental pattern matching by the descriptive language. They are left to the user to avoid them. In this sense the programming with such a descriptive language becomes also creative and a sort of art. Not just because of possible flaws, but mostly because of powerful possibilities for matching complex contexts that the language gives to the rule writer. Note that the algorithms of Pelz can be still combined with the language and serve as some kind of syntax check once the rules written by the expert are being compiled. The syntax check can issue warnings and errors pinpointing inconsistent rules, that for instance are impossible to be matched.

We will, in further text, describe the language defined by Olbrich and Barke and its algorithmic solutions together with the unique concepts devised for this approach. This approach has, in its later development stages, shown stable and accurate industrial application. By describing that project as an example we want to introduce the reader to the Clarula language and properly set up the context for the explanation of the contribution of this thesis.

2.5 Classify project – Clarula descriptive language

We will present here one realisation of the concept of the incremental pattern matching strategy. This realisation defines a specific language (Clarula) which implements basic flow control constructs and certain template mechanisms for the structural pattern. This research project was realised by Olbrich/Barke for the application in the real industrial environment. The goal of the project was to achieve the general purpose SR tool based on incremental pattern matching. The industrial version of this tool is named *classify*. The flow of the tool could be drafted as in Figure 2.5-1.

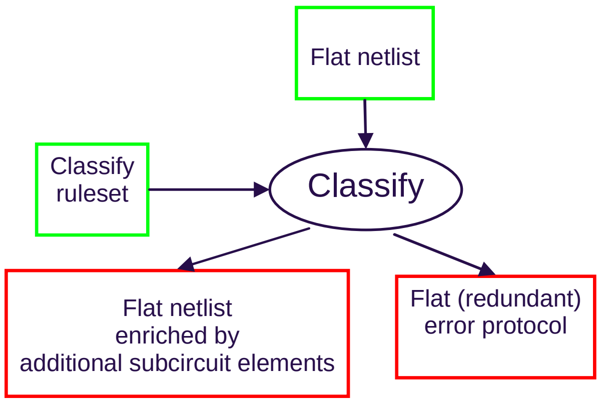

Flat netlist
Classify ruleset
Classify
Flat netlist enriched by additional subcircuit elements
Flat (redundant) error protocol

Figure 2.5-1 – Pattern matching tool cClassify – execution flow.

The flat input netlist is compiled into an in-memory bipartite graph model of the circuit, together with the rule set. The ruleset is an instance (a program) of the descriptive language - Clarula. Therefore, the memory representation of the ruleset consists of the framework to lead the program execution (flow control) with the number of specific “pattern side” graphs that represent the graph regular expressions. The output of the tool is the partitioned netlist and a specific error report file. This ASCII file has a syntax which enables it to be used together with third party graphical user interfaces that represent the hierarchical designs (Cadence Composer®). Therefore, the idea is to integrate the pattern matching tool with a specific rulset (which performs specific electrical rule check of the design) into Composer® to be able to graphically specify exactly the places where some violation has occurred.

The rules language program has a clearly defined structure. It starts with the type definitions that are followed by the net and block predefinition assignments. This

```
 #comment
 # rules block
 .rules
 < type definitions>
 <net and block predefinitions>
 <rule definitions>
 < protocol comment>
.endrules
```

Figure 2.5-2 – Clarula language structure

section is followed by the rules. Keywords and commands start with a dot (.). The program is bounded as a structure that begins with a ***.rules*** keyword and ends with a ***.endrules*** keyword. If the first character of a line is '#', that line is treated as a comment. No inline comment is allowed. The conceptual structure of the Clarula program is given in Figure 2.5-2.

The block <type definitions> strongly declares the set of types that are going to be used throughout the program. The types are assigned to devices (blocks), nets and ports. The assignment keywords are ***.blocktypes***, ***.nettypes*** and ***.porttypes*** respectively. One example of the type set is:

.nettypes *signal power ground tobedriven pdrive ndrive fixedV pMultCon nMultCon bidirekt .*

This statement declares ten different user defined types that give semantics to the nets used throughout the given Clarula program.

The patterns specified in the rules can demand a certain type for a net in order to match. The rules can assign types after successful matches. However Clarula uses the naming convention of the nets in the design to assign the initial typing of the target circuit. This is done in the *<net and block predefinitions>* section. With the statement ***.netpredef*** the user can assign a type to a target net according to the string regular expression. If the target circuit net name matches the string regular expression appropriate type is assigned. For instance:

.netpredef *'vblh*':power*

would assign the type *power* to any net in the target circuit that has a name which starts with a string "vblh". Arbitrary number of lines of this type is allowed, meaning that we can define arbitrary number of rules to assign types according to the string patterns.

The initial types of the atomic blocks of the target circuit can be read directly from the design models. Each device in the electronic circuit design has a clearly specified type.

In this way we have specified the vocabulary for typing the patterns and assigning the initial types of the electronic circuit. Further program structure represents the collection of rules that are combined with the flow control statements.

Clarula defines three types of rules:

- Block rules
- Adjacency rules
- Net rules.

The most general and in the same time mostly used rule type are the block rules. This rule type can match the arbitrary structural pattern and apply a certain operation on the pattern image in the target circuit. Block rules have the following typical syntax:

.blockrule *<name> <port list>*

Pattern: $\left\{\begin{array}{l} \\ \\ \\ \end{array}\right.$ *<element 1>*
[<element 2>]
…

Actions: $\left\{\begin{array}{l} \\ \\ \\ \end{array}\right.$ *[**.gets** <assignments>]*
*[**.flatten** <element list>]*
*[**.param** <parameter definitions>]*

*[**.check** <check list>]*
.endblockrule

The block rule has its name and the list of ports. The ports specified in *<port list>* correspond to external pattern ports and they are as well defining the pins of the instance of the given abstraction that can be inserted by the appropriate action. This header line is followed by the structural pattern definition and a mixture of possible actions that can be performed on the matched image of the pattern inside the target circuit. We can recognise that this realisation of the rule follows the standard structure of the rule for incremental pattern matching, discussed in the previous section.

The pattern is specified in a syntax that resembles the SPICE netlist format. The SPICE netlist format represents data as a list of devices whose types are determined and whose mutual connections are specified by referencing the net names in a device terminal list. This kind of textual representation is natural and already known to the designer, a possible user. The syntax is actually enriched with several concepts needed for pattern matching. First, elements of the netlist can have defined types. The type is assigned to a net or a block by writing its name followed by a colon and the specific type name (e.g. *x1:inv a b c:pwr d:gnd*). The element of the pattern can also define specific parameter values that are required to be identical to the candidate element in the target circuit (PARDEF) in order to match it with the pattern element. We can formally write down the syntax of the single element of the pattern:

<name>[:<type list>] <port list> [PARDEF <parameter definition>]

Additionally, the language defines one possible abbreviation of templated properties. Clarula defines the concept of *optional ports*. By employing this concept one can match a family of circuits by a single pattern. Optional ports allow the pattern to have generic connections (that appear in some instantiation cases and in some not). The language however does not allow the generic number of devices that are the members of the single pattern. This is rather achieved by applying the rule recurrently, employing the flow control. The example of this strategy will be given together with the definition of the flow control structures.

The ports can be divided into three classes. *Mandatory* ports, are the terminals that have to exist in the pattern image in order to have the correct match. On the other hand, *optional* ports can be left unmatched and the pattern as a whole can still be successful. Special kind of optional ports are *multiple* optional ports. The semantics of optional ports differs in the port list that is attached to the element in the pattern list and in the port list which denotes the list of external ports. We will illustrate both strategies employing the examples in Figure 2.5-3. First, let’s analyse the meaning of the optional port in the external port list. Three similar rules are specified in our example under a, b and c. The first rule allows the pattern to have two external ports, the second requires three while the third specifies the port b as optional (written [b]). That means that the port b is allowed to have external connections but it is not required to. As a consequence if we apply these three rules on the target circuits shown in the figure under d and e, the first pattern would match only the example in the circuit d, the second pattern would match only the circuit under e and the third pattern would match both circuits. To conclude, the optional external port has allowed us to “compress” two similar patterns, with respect to the external port configuration, into a single rule.

```
.blockrule serres1 a c
r1  a b
r2  b c
.gets serres:res
.endblockrule
```

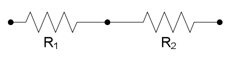
a)

```
.blockrule serres2 a b c
r1  a b
r2  b c
.gets serres:res
.endblockrule
```

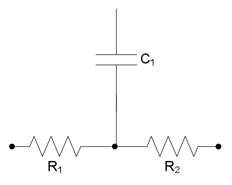
b)

```
.blockrule serres3 a [b] c
r1  a b
r2  b c
.gets serres:res
.endblockrule
```

c)



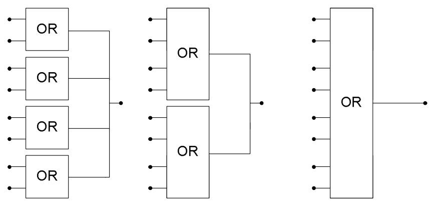
d)



e)

```
.blockrule or1 a B c d {e} {f} x
x1:or a b {e} x
x2:or c d {f} x
.gets gate:or
.endblockrule
```

f)



g)

Figure 2.5-3 – optional port usage examples. a, b and c) three variants of block simple block rule to match the serial connection of two resistors with and without optional ports. d, e) target circuit for the patterns defined. f) rule to match the generic number of inputs or gate. g) Recognition sequence.

Multiple ports have similar semantics, but in addition they allow multiple target side matches for a single pattern side multiple optional port. The net can have a number of ports that connect to it and not just exclusively 0 or 1.This concept is very useful for recognising the circuits implemented in different, similar technologies. For instance the same logical circuits can have two different power supply solutions can be matched with the single rule using this strategy. We can abstract all power (or ground) nets with a single multiple port .This concept is as well useful for matching the generic circuits. In our example figure, under f is specified the pattern that can detect an OR gate circuit with the generic number of inputs. Additionally the target circuit shows one implementation of 8 input or circuit realized with four two input or circuits and the recognition sequence that leads our target circuit into a topology where it has one or gate with eight inputs. The multiple optional ports, as it can be seen in example, are denoted by curly brackets ({<port name>}).

Similar to this example we can now notice that the optional port concept can trivially solve Subgeminie's problem of the shorted external ports. Any external port pair that can be shorted can be defined as a pair of mandatory and the optional external port.

To conclude, this actual application of the concepts for the structural pattern as a structural regular expression witnesses the correctness of the analysis of Chanak [18] where he predicts that due to the implementation complexity just a carefully chosen mixture of the possible graph regular expression concepts should be used. Note that an alternative for optional ports would be writing exhaustively separate rules for each of the combinations of these generic element occurrences. In the case of the OR circuit example this number is (theoretically) infinite!

Once the pattern is matched Clarula can execute a number of different actions. There are four action classes. The tool can (re)assign parameters to certain blocks in the circuit, using a keyword ***.param***. Further, it can by issuing a command ***.gets*** insert a new block, exchanging it for a topology that the rule has matched. The block is precisely connected by the pins specified in the rule. Note that the algorithm determines the proper usage of optional ports. In order to make the tool more powerful it is possible also to collapse some blocks which were abstracted in some previous rule. For this reason the keyword ***.flatten*** is used followed by the name of the block that is got in context and which is supposed to be flattened. One example of this concept is when after the recognition of latches, the rule writer wants to keep just their weaks and re-use forwards for some other purpose (they can be a part of a driver driving the next latch that is connected in cascade). In the end, the ***.check*** action is issuing an entry for the error protocol file if an arbitrary test condition (which can be for instance a test on specific parameter values of the devices that are matched) succeeds.

The other two types of rules have the similar syntax but a slightly different pattern and action specifying concept. We sketch them briefly just for completeness. The work related to this thesis is in the scope of block rules.

The adjacency rules concentrate on devices. They group interconnected devices of the kinds that are specified in the rule instance with no respect to the topology they build. The only criterion is that they are adjacent. Additionally to the kinds of the device types that are to be gathered *cutnets*, the nets that define the stopping criteria and after which no matching is further performed. This kind of rules is especially useful for isolating parasitic networks. By employing this rule type RLCK networks of the arbitrary topology can be easily found and highlighted.

The third rule type matches exclusively nets. The net is matched and appropriate conclusion is applied on it according to its type(s) and the type and the number of ports which are attached to it. They are useful for fast signal propagation.

As we have already mentioned the sequence of rule execution can be controlled by the simple flow control. Apart from the sequence, that is defined by simple applying the rules one after another in the program listing, for and while loops are defined. Their semantics is however different to the loops with identical names defined in procedural programming language.

The for loop executes a group of rules as long as any of these rules match. Its execution is as well optimised. For instance, if we have two rules in the for loop:

```
.for
  {
  Rule1
  Rule2
  }
```

, if the first rule matches and the second doesn't, the for loop executes again the first rule. If it in this new attempt doesn't match, since the target circuit was not altered, we can be sure that neither the second rule will be matched. Therefore, the for loop exits after the matching process of the first rule, skipping the second.

This kind of loop is very useful for the recurrent matches. For instance the matching process from our OR gate example would be executed by the simple program in Clarula specifying the for loop and the single rule we have already defined.

```
.for
  {
  .blockrule or1 a b c d {e} {f} x
  x1:or a b {e} x
  x2:or c d {f} x
  .gets gate:or
```

```
        .endblockrule
        }
```

As we have stated in the example the syntax for the for loop is composed of the ***.for*** keyword and the curly brackets.

Another loop type, the *while loop* is similar to the for loop. This kind of the loop executes a set of rules if another additional set of the rules matches (if any rule from this set matches).

The language which was specified here serves as a powerful and flexible tool to run the context driven netlist processing. In Clarula, it is possible to write tools which perform important static rule checks or make the optimization preparatory steps for the simulation. Although powerfully conceived, the potential of this strategy is not being fully used due to different issues related to the real realisation of the concept for general hierarchical designs. The concept was implemented to work with the flat input netlists. Their size is extensive. First consequence is that the tool can be run just on a certain blocks separately and not on the full chip.

Further, the engine for the SR is the depth first search algorithm that is constrained only by the circuit element types. Since patterns have a complex structure, once the starting device is matched, the algorithm recursively approaches other devices following the current device's connections. Connections between the elements are ordered. This is naturally important as every connection has a different semantics. For example the first port of a transistor element represents the drain terminal, the second gate and the third source terminal. After following one of the ports of the circuit element, in the same order as they are defined, the algorithm approaches the net which can have connecting ports to an arbitrary number of neighbouring elements. In order to confirm or reject a match all possible paths from the given net, in the worst case, need to be checked. Of course, in case that the true match is found the search is terminated. This heuristic approach creates however sometimes inacceptable runtimes. We have tried to optimize the execution of the search algorithm by a greedy approach where one chooses always the path for the depth first search through the "best looking" net (the net with the smallest number of neighbours). This enhancement of the depth first recursive search although trivial for the application in the non templated graphs becomes much more difficult in collocation with the concept of optional ports that is one of the most important mechanisms in Clarula.

2.6 Treating big nets in the incremental pattern matching algorithm

In this section we are going to discuss the efficiency problem of the engine SR algorithm of the Clarula language and propose the algorithmic solution for this problem. The experiments which witness the benefits of the applied solution are discussed together with other contributions of this thesis in Chapter 6.

There are nets in the circuit that have an exceptionally high number of neighbouring devices, up to the order of 10^5. Those are usually supply (power and ground) nets, signal nets surrounding logical abstractions which have big fan-in/fan-out or reference voltage nets. If a large net is considered, the algorithm tests all of the possible connections in order to make a conclusion about a match which always has, unsurprisingly, a greater possibility to be false. Therefore, observed from a given net, this operation includes an exhaustive linear search. Not the whole structure of the pattern has to be analyzed until the algorithm concludes that it is attempting to match the false candidate place for pattern instantiation. We can choose to use a different order when following device interconnections, so that the examination of large nets is postponed. Then, there is a high possibility that, for these examples, these nets won't be processed at all. This depends on the similarity between the pattern and the false match instantiation and therefore, how early the algorithm can make the conclusion that determines the current match attempt. In case of a true match, while applying different ordering of the recursion, it is also possible to skip the processing of some of the large nets. If there is more than one path to test the graph's topology, we can approach the same device in different ways. Thus, we can close the loop path directly on the large net, without examining it.

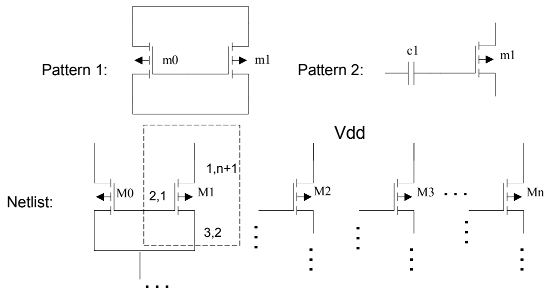


Figure 2.6-1 – Example of the matching process

This new method will be illustrated with the example in Figure 2.6-1. Two example patterns are matched against the netlist. The first pattern represents two transistors which are connected in parallel. This is a very common pattern which would merge two parallel transistors in the netlist. The second pattern is a simple conglomerate of one transistor and one capacitor which is connected to its gate. We will try to match both of the patterns to the given netlist, starting from the candidate place in the netlist which is marked by the dotted rectangle. In addition to patterns, the figure shows an example netlist. Candidate element connections are depicted with a pair of numbers. The first one represents the definition order of the terminal and the second its weight. While matching both patterns, starting from the candidate element, we would ideally proceed with the recursion by first following terminal 2, then terminal 3 and in the end terminal 1.

In the case of pattern 1 we have a true match. The algorithm will start with M1, assume a preliminary match with m1 and follow its gate net. It will proceed with the assumption of a match between m0 and M0. Then it checks the net connected to drain of m0 on the pattern side and finds that it is connected to the already matched m1. It verifies that on the circuit side drain of M1 and M0 are also connected to the same net. The same process is applied to the source terminals. The algorithm returns a match without examination of the other devices connected to Vdd.

If the algorithm first tries to match terminal 1 of m1/M1 and follows the Vdd net on the circuit side, it has to examine all devices attached to Vdd as candidates to match m0, which in the end also works but is expensive. That is exactly what the original algorithm does, it just picks the first terminal of an element as the net to be followed. As a result, the performance of this algorithm is not perfect.

For Pattern 2 the algorithm should ideally try to check the gate connection of M1 and conclude that the match is false as the type of the device connected to its gate is inappropriate. In the case of the original implementation of the depth first search algorithm the matching process also includes a linear search over the power net Vdd. The algorithm always starts with terminal 1 and searches over all n possible neighbours.

Therefore, an intelligent depth first search algorithm would always attempt to choose the next possibility for recursion which connects to the net that has the lowest number of neighbours and then approaches large nets only if necessary.

The realization of this idea is straightforward in the case where patterns have only mandatory ports. Simply from one exact point in the graph, the algorithm can pick the appropriate pair of pattern/circuit ports, which have a one to one correspondence. The underlying statistical data can be collected while building the memory representation of the circuit and maintained later on.

The pattern syntax in our classification method includes the very important and powerful concept of optional ports, as mentioned. This concept is making the formulation of Best Path First (BPF), depth first search algorithm much more sophisticated. The algorithm will be explained in continuation.

This greedy approach is witnessed to give good results by Chanak [18]. Since the look ahead is just one, this greedy solution might not bring us to the best path for the search. Therefore, after a net with a small number of neighbours can stand a device whose all other terminals are connected to big nets. However having in mind the sparcity of the VLSI designs this is not likely to happen even during the searches for big patterns. In the case of incremental pattern match, when the patterns are small, most often containing only two devices, the greedy approach is optimal.

Proposal of Best path first algorithm

Our solution modifies the depth first search algorithm and allows the arbitrary (cheapest) approach to the different correct pairs for the ordered pattern-circuit ports before entering the next recursion level.

The solution, naturally, has to support the usage of optional ports in the pattern. Support of optional ports implies a very complex way of distributing port pairs. The determination of the corresponding pattern port for arbitrarily accessed circuit port depends on the distribution of previously approached circuit ports. In the example in Figure 2.6-2.a the circuit side device, X1, has 6 ports which connect it to the rest of the graph. All ports are connected to nets that have a potentially different number of neighbours. We attempt to match X1 to a pattern device P1, which has 5 mandatory ports pA, pB, pD, pG and pH as well as 3 optional ports pC, pE and pF as shown. Optional ports are marked with square brackets. Lets follow the strategy to proceed with the recursion by first following the path through the net which has the least number of neighbours.

```
X1  c1 c2 c3 c4 c5 c6 element_type
P1 pA pB [pC] pD [pE] [pF] pG pH
```

a) circuit vs. pattern element

```
3  4  2  5  6  1
```

b) optimal order of approaching circuit ports

```
 3-C  4-D  2-B  5-G   6-H  1-A
      4-E  2-B  5-G   6-H  1-A
 3-D  4-E  2-B  5-G   6-H  1-A
```

c) optimal searching path

Figure 2.6-2 – Example of BPF ordering for port pairs

A	B	[C]	D	[E]	[F]	G	H
1	2	3	4			5	6
1	2		3	4		5	6
1	2		3		4	5	6

Table 2.6-1 Port Pair Distribution

Let the surrounding of the device be such that the order of visiting from the Figure 2.6-2.b is optimal, satisfying the look ahead 1 optimum. The best search path for our example is then shown in Figure Figure 2.6-2.c. The path starts with the third circuit port and picks its first possible pairing port from the pattern side, port C. This pair determines a context and influences possible positions and pair forming of the ports that will be approached later. For instance when circuit port 3 is paired with pattern port C, circuit port 4 can only be paired with D. On the other hand if we pair circuit port 3 with the pattern port D, possible ports that 4 can be paired with either E or F, once it is approached on the different recursion level of the modified depth first algorithm.

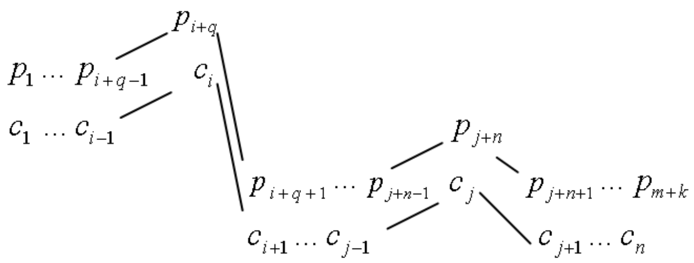


Figure 2.6-3 – BPF vector partitioning

For illustration, Table 2.6-1, shows all possible distributions of the vector of circuit ports to the vector with pattern ports. In any case all mandatory ports have to be matched. Matching of optional ports is, of course, not obligatory. Therefore we can make the conclusion that two circuit elements can be matched only if the number of circuit ports (n_{cp}) is greater or equal to the number of mandatory ports (n_{mp}), and on the other hand smaller or equal to the total number of pattern ports, the sum of mandatory ports and optional ports (n_{op}), or:

(1) $$n_{mp} \leq n_{cp} \leq n_{op} + n_{mp} .$$

The solution for the context driven distribution of port pairs lies in combing this simple inequality with recursion.

Let C be a vector of elements $c_1,\ c_2, \ldots ,\ c_n$, which represent pointers to the ports of the circuit element that is being processed, where *n* is the number of these ports and P the vector of elements $p_1, p_2, \ldots, p_{m+k}$, which represent pointers to the appropriate pattern optional and mandatory ports, where *m* and *k* are the numbers of members of mandatory and optional ports, respectively. Above vectors are shown in
.

The algorithm first attempts to place element c_i, where $1 \leq i \leq n$, together with the element p_{i+q}, where at first q equals zero. This attempted place is accepted, and the pair is formed if the number of circuit ports before the c_i (*i-1*) satisfies the inequality (1) matched to the first *i+q-1* pattern vector elements. Additionally, it is required that the number of remaining circuit ports, that follow the port c_i (*n-i*), should satisfy the same inequality, matched to the number of remaining pattern vector elements, *i+1,…,m+k*.

If (1) is satisfied for both parts of the vector, then the pair p_i - c_i is taken. Otherwise the algorithm attempts to check by the same test iteratively, while incrementing q, all possible pairings of c_i, continuing with c_i and p_{i+1} (q=1) and on.

Further iteration on this level is done if later, during deeper stages of the recursion, the algorithm concludes that the proposed distribution of pairs is not leading to the true match. After exhausting all of the pairing possibilities for the given circuit port, the algorithm terminates the present stage of recursion and returns false as match to the earlier stage. Finding the match before this simply means that, at every stage up to the top, the algorithm would return true.

On the next recursion level the same algorithm steps are performed on the local portion of both vectors. This is illustrated also in Figure 2.6-3 as the pairing of port c_j.

The solution is implemented in C++ programming language using mentioned and additional, auxiliary data structures, such as a set of indexes of circuit pointers, which is ordered according to the number of neighbouring elements of the net that the port is connected to and the tree which saves the recursion context. The module which was developed is smoothly inserted into the old algorithm leaving most of the code intact. The places where the old algorithm was determining (trivially) the next pair of ports before diving into the next recursion step could be clearly identified and isolated.

The algorithm has shown stable and reliable industrial application for already more than two years. The runtime improvement reaches, for some big examples, the factor of 60! We will discuss the experimental results for this algorithm in Chapter 6, together with the other results achieved in this thesis.

2.7 Inexact pattern matching applied to subcircuit recognition

For completeness of this work, although these approaches are not used in our methodology for hierarchical pattern matching, it is important to give a brief overview of inexact pattern matching algorithms that are developed for the application of SR.

Several inexact subcircuit recognition algorithms are known in literature. They are based on different classical pattern matching optimization based approaches. One of the central places and astonishing results are achieved by the application of graduated assignment algorithm, the optimization algorithm that combines iterative optimization approach with probabilistic physics. This approach has been applied and refined in order to get the fast and robust matching algorithm for subcircuit extraction by Nicolay Rubanov [22-24]. In the work of Rubanov, he, at first, defines a labelling algorithm [25] that offers good discriminative properties as a preparation for the application of graduated assignment [7]. Further, the algorithm is carefully tuned and altered in order to be able to isolate all instances of the given pattern from the target circtcuit (represented by a matrix) in almost all application cases. In order to fight the main problem of inexact algorithms and that is that, despite their speed, they are not completely accurate, Rubanov uses two other known approaches in pattern matching theory. He employs error back-propagation and postponed decision making techniques to refine the output of the incremental optimization process. The negative side of this algorithm is the fact that the input target circuit is expected to be flattened. In the industrial realistic application, this can be a considerable problem.

Other approaches apply known pattern matching techniques with bigger or smaller success. Fuzzy attributed graph approach is studied by Zhung et al. [26]. It was however used in the pure university environment; therefore the implementation is not so powerful. Hence, the pattern which has been used for testing is hard-coded in the algorithm. The pattern was the flat implementation of the NAND gate, consisting of 10 vertices (4 devices and 6 nodes). Further examples include SUBGEN algorithm [27], that follows the genetic algorithm approach and other exotic approaches [28-30].

2.8 Addressing designs with extensive size by employing hierarchy

In the realistic usage of algorithms for pattern matching the common problem that stays unsolved is the size of the input circuit. Today’s designs have often more than a billion atomic elements in the flat representation. Therefore, they are not able to be treated at first because their extensive size which is far bigger than the typically available resources. Apart from that, matching the identical patterns that are instantiated a number of times leads to unacceptable runtimes.

In this chapter so far we have intended to present the importance of SR for the chip design verification. The applications and the benefits are numerous. Unfortunately, to achieve the developed level of the initial vision where one has a chance to intelligently control the verification process is not trivial. We have pointed out various runtime enhancements for the matching process, frequent problems and their solutions. Still, one additional problem that stays common and without an appropriate answer for all the demonstrated algorithms is general pattern matching for hierarchical input netlists! In all of the mentioned approaches we have the input netlist of a specific class - a flat netlist. As the chip designs are typically hierarchical an additional and expensive (runtime and memory requirement) process of flattening is a must before any analysis. As we have mentioned, this leads to inability to perform checks on full-chip designs at all, due to inappropriate resources of today’s computers. What makes the matching of hierarchical netlists so hard?

The main problem that the algorithm which has to work on the hierarchical input netlist has to solve is matching across the subcircuit boundaries. For instance, if the netlist abstracts the definition of even an elementary transistor device, it can be referenced in a thousand of places building complex structures. This example is trivial but in general the modules that the designers build often have “unfinished” contexts that get their semantics only once the block is properly placed in its instantiation environment. One additional example is, for illustration, a driver of a latch that is abstracted as a separate subcircuit. This requirement renders the trivial hierarchical pattern matching solution, where no matches across the cell boundaries are allowed, not very useful. There are nevertheless some attempts of the academic environment or even some commercial tools which employ the simplified approaches and which can achieve results in some special cases.

For instance, interesting is the algorithm for hierarchical netlist comparison (LVS comparison) – Hcompare [31]. This algorithm relies on the identity between the subcircuits and if the identity is proven the given subcircuits are kept. On contrary this algorithm flattens the subcircuits having differences and performs conventional pattern matching on the given hierarchical level. If this current "master" level is identical in two netlists it is kept, if not it is collapsed anytime the level is referenced. The algorithm execution strategy follows the bottom-up approach. To conclude, this algorithm is suitable for comparing two hierarchical netlist with similar topologies. If the topologies are different the algorithm performs full flattening of both netlists.

Terem at al. [32] developed the specific approach that employs selective flattening down to the "interesting elements" (members of the pattern), exclusively. This enables them to match patterns orthogonal to the subcircuit boundaries. Still they stress that this algorithm is just for very high-level pattern matching. This limitation is crucial for the feasibility of their approach. Think of choosing a transistor as an "interesting element".

If we had the general pattern matching algorithm for hierarchical input netlists, we could employ the SR with its full power. The full-chip pattern matching driven analysis would be possible. The tool that employs such concept would be also be able to partition the hierarchical netlist in much more efficient way, achieving the ability to highlight or alter some critical topologies non-redundantly, directly on the hierarchical netlist. Apart from pure qualitative enhancement that came as the full-chip ability, the runtime and memory requirements of the SR process would, due to ability to work directly on proper definitions be optimal and non-redundant.

All of these reasons give us a strong motivation to develop the general solution that enables SR to reach its mature and more appropriate version employing directly hierarchical designs that are a realm of the industrial application.

3 Hierarchy

The hierarchical organisation is a concept that appears very often in science, society and nature[33, 34]. This is a common way to fight the complexity and enable understanding, as well as functioning of different complex systems. The hierarchy is also more than a way of organisation, as it has its own semantics. Any digital electronic circuit can be an example of the fact that the function which it implements is not dependent on the technology in which it is realised, but completely orthogonal to it. In this sense, the hierarchical organisation appears as a completely independent abstract layer.

3.1 Hierarchical abstraction in VLSI

3.1.1 Introduction

Apart from other fields hierarchical concepts find wide application in VLSI design to address the extensive complexity of chips which are being shaped. Moreover the concept of hierarchical abstraction is embedded into the methodology of designing the IC and in the process of their verification.

In further text of this chapter, we will draw attention to the role of the hierarchy in VLSI design. In order to do that, we will start with the formal definition of the folded hierarchical data model in (3.1.2). Further, we will give an overview of state of the art EDA databases that implement the folded hierarchical data model (3.2). In a simple case study we will present common algorithms and data-structures that are used to explore the hierarchically organised IC designs (3.3, 3.4). The chapter is concluded with a vision of hierarchical views that serves as a starting point of the realisation of this thesis's contribution (3.5).

3.1.2 Folded hierarchical model

Like in other complex systems, hierarchy is exploited as one of the essential mechanisms to develop and store electronic circuit designs. The concept of hierarchy helps IC design process in many ways and it became a part of the methodology of custom digital or analogue design. By employing hierarchy, the designer typically works on a certain functional block, a part of the design, which once finished represents an element (hierarchical level) that is a verified and correct building unit. This building unit can be further applied in different contexts.

In order to illustrate this concept and explain the benefits of hierarchy, we will consider the example of the 2-bit adder electronic circuit. We see the flat version of the mentioned circuit in Figure 3.1-1.

This flat design is built out of ten elements. If we analyse its structure we can conclude that it contains a topology of a full adder which repeats two times in the circuit. One of the full adders is highlighted in the figure by the rectangle with the sharp corners. Further, in the composition of the full adder one can isolate another topology which repeats twice – the topology of the half adder. One of the half adders that appear in this design is highlighted by the oval rectangle. In total we have four topologies that are isomorphic to the half adder and two that form the full adder.

Let’s now take advantage of these reoccurring patterns and describe (store) the circuit *hierarchically*. Actually, the hierarchical organisation of the example circuit that we will create would in reality come spontaneously. The designer who would make this occurrence of a generic n-bit full adder would have the bottom up approach and he would first create the topology of the half adder, continuing with further design using the half adder as the building element for more complex contexts.

The half adder is composed of two logic gates that have mutually shorted inputs and in total this circuit communicates with the rest of the design by four terminals (two input and two output). Using the finished and correct topology of half adder, the designer than builds a full adder out of it and further the required n-bit adder. The natural outcome of this process is the hierarchical design.

In Figure 3.1-2 we see realisation of a half adder as a part of the hierarchically described circuit. In the hierarchical representation given in the figure, we can isolate three distinct hierarchical levels that define also clear functional contexts. The deepest level which represents the half adder is defined by two logic gates, XOR and AND.

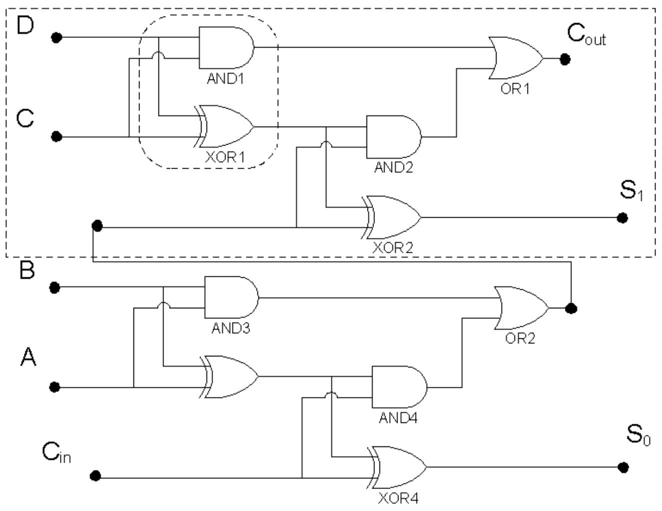


Figure 3.1-1 – Flat representation of the 2-bit adder. Full adder circuit that is a part of 2-bit adder is highlighted by the rectangle with the sharp corners, while half adder circuit that belongs to the full adder is highlighted by the oval rectangle.

This topology is encapsulated in the hierarchical level (or a subcircuit) called HALF ADDER. We have used this subcircuit as an opaque block to define a higher hierarchical level which combines it with other elements. Our example design has, thus, the level FULL ADDER that forms the electronic circuit of the full adder by interconnecting properly two instances of the HALF ADDER subcircuit, using them as circuit building elements together with another atomic element (OR gate). This is a nice example of a powerful mechanism where one can define complex elements that are further smoothly used with other complex elements or atomic elements in order to build any arbitrary circuit. In order to enable the subcircuit that we create to correctly communicate with its environment and to look and feel like a proper atomic element we define specific terminal connections. In the top level we have two instances of FULL ADDER subcircuit forming the circuit that is equivalent to the flat design from Figure 3.1-1.

What are the advantages of this representation? First, we can see that in order to form this circuit we have clearly focused our attention to three different semantic levels, at first we have created the half adder thinking in the world of logic gates. Than we go one level up and use already more complex circuits and interconnect

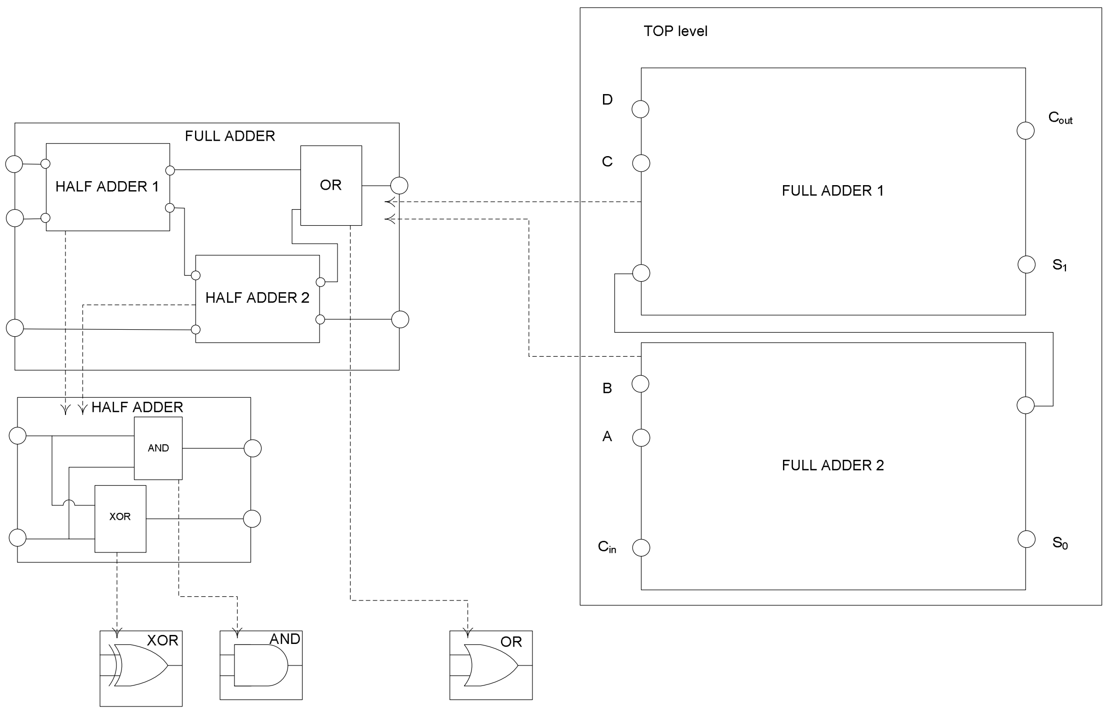


Figure 3.1-2 – Hierarchical representation of the 2-bit full adder. The Hierarchical levels and connections between the elements are given with solid rectangles/lines, while the references between instantiations of hierarchical levels and their definitions are given by the dashed lines.

them to get the functionality of the full adder. Note that the designer can always "tune" his design to get the proper functionality using any elements. In this case, in order to detect the carry we simply use an OR logic gate. In the end when we have the full adder encapsulated we can easily choose the size of the n-bit adder we want to create, not thinking of the inner implementations that reoccur for each bit we add. This architecture is also good for someone that should understand the design. One can immediately see that the top level is composed of two properly interconnected full adders. For an experienced designer this can be enough. He doesn't have to look at how the full adders are realized. Here lies another advantage of the hierarchical design representation: once we have abstracted the functional unit, we can exchange its implementation. For instance, we can redefine the full adders, or the half adders to be composed of exclusively NAND gates and still keep the rest of the design being sure that the functionality of the circuit won't change. Further, this design is also technology independent. We have defined the functionality strictly using gate elements. We can add another level of hierarchy seeing the gates that are here shown as atomic elements (with no further hierarchy and inner structure) as complex topologies of CMOS transistors. In this way we would just have to define these atomic elements and inherit the whole further design and still achieve wanted functionality.

Apart from flexibility and the ease of understanding, this representation is more efficient, as well. The hierarchical representation allows that the definitions of given separate levels can be referenced many times in the design. This concept is known as *folding*. We also say that the design is than folded. One shouldn't confuse this term with time folding, where one uses time multiplexing to reduce the given design's size or share an expensive resource.

As we can see in the example, we have just once defined the full adder and used (referenced, instantiated) it twice. Further, full adder has two occurrences of the half adder which is again defined only once. The same principle is valid for the logic gates, although we have them referenced and used directly only once in this example. We can say that one of the benefits of the hierarchical representation lies in the fact that it is non-redundant. We have managed to represent a design that has 10 elements by directly employing just three of them or each of the atomic elements once. Something that is in the same time advantage and the disadvantage of folded hierarchical representation comes in the domain of the tools that should analyze the designs. For some operations this concept is welcome, for instance for counting the number of atomic devices in the design, or for checking some of the attributes of each of the devices alone. On the other hand hierarchy (folding) represents the problem for some other group of tools that prefer seeing the design as a whole, for instance a simulator that needs to propagate the signal through the circuit from its input terminals towards its outputs.

It is often required by tools to characterise some instances of the subcircuits, too. For instance, one might want to attach specific parasitic elements to an instance of a given subcircuit. This is known as a problem of personalisation. We will come back to these problems and known solutions for them later in this chapter, in section 3.4 .

In order to formalise the described concept we will use and adapt the definition of hierarchical encapsulated graphs. As flat graphs haven't met the requirements in many application areas of computer science, Engels et al. [35] have proposed the model of the graph that includes the hierarchical concept. We will adapt this concept in order to formally represent folded hierarchies that are widely used in EDA. As a type of the graph that is typically used to represent the netlist is a bipartite graph, we will extend this kind of graph notation to enable hierarchical relations.

In order to achieve this goal we define *complex vertices*, as an extension to the standard (atomic) vertex concept. The complex vertex is a part of the graph and can be equally used together with atomic vertices. The difference between the atomic and complex vertex is such that the complex vertex defines the inner structure, as well. The inner structure of the complex vertex is again a graph that can contain any kind of vertices, including other complex vertices. We can, therefore, say that the definition of the encapsulated hierarchical graph is recursive. The set of atomic vertices is denoted by *N*, while the set of complex nodes is denoted by *Y*. As each complex vertex is a graph itself we write that the top complex vertex equals to encapsulated hierarchical graph G(N,Y).

For each complex vertex we formally write:

$$CV = (V, E, KE),$$

where:

(1) V is the set of vertices of CV,
(2) E is the set of edges that belong to CV and
(3) KE is a set of known edges in CV, where $KE \subseteq E$.

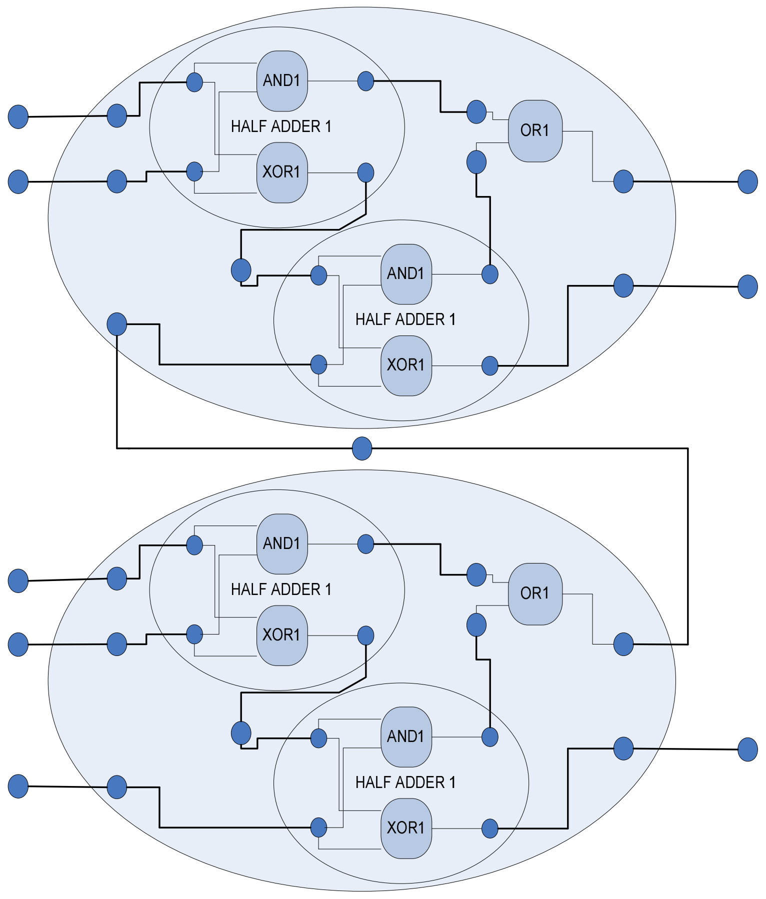


Figure 3.1-3 – Encapsulated Hierarchical Graph Example - The complex vertices are given by light blue ellipses, while the atomic elements as blue squares with oval edges. The node verticies (another vertex class) are given as dark blue circles. Encapsulated Hierarchial graph can clearly separate and define hierarchical levels, but is redundant.

In order to identify the relations between these sets, we will write down further definitions:

(1) HE = E \ KE is a set of hidden (private) edges of the complex vertex CV.
(2) HV = V is a set of hidden vertices (private vertices).

The known edges are the edges that are incident to the complex vertex. Since we have the case of the bipartite graph, V (HV) can be further split into two sets V = X U Y. These sets have the semantics of devices and nets, respectively, for graphs that model hierarchical electronic circuits. Further X contains, in general, two kinds of elements: complex vertices and atomic vertices. For this reason we split this set into two subsets X = CX U AX. In the end, we say that the complex node defines a *level of the hierarchy*.

In order to illustrate this concept, we will represent the full adder circuit from our example with the encapsulated hierarchical (bipartite) graph. Vertices are divided in two groups. The first group, which is drawn with dark blue circles, represents the circuit nets. Another group represents the devices, split again in two subgroups, instances (complex vertices) and atomic devices (vertices). The edges are given as lines. The known edges are highlighted as they are drawn with ticker lines. This hierarchical graph has 41 vertex of a class "net". Further, it has 10 atomic elements. This is a nice illustration of the fact that that the unfolded hierarchy doesn't bring us any advantage concerning the number of elements needed in the model. On the contrary we have some of the elements duplicated in different hierarchical levels (net vertices). The established formal representation can, thus, define hierarchy and clearly specify the borders between the complex entities it models, but we can not represent the folding as a property with it. In order to do this, we will extend the notation of encapsulated hierarchical graph with several additional definitions that enable folding.

The complex node has two sets (HE and HV) that are internal to it. If the set HV = HX U HY, we will observe the set HY (hidden nets) as union of two subsets (HY = EY U IY). Further we will define the one to one mapping relation A, between the elements of the sets KE and EY.

Let $A \subseteq KE \times EY$ is such that:

$$(\forall a \in KE, \exists! b \in EY : (a,b) \in A) \wedge (\forall b \in EY, \exists! a \in KE : (a,b) \in A).$$

By specifying this new mapping we can now say that d = (HV, EY, HE) is a complex node *definition*. We can further write that the complex node represents a tuple:

FV = (d, KE, A).

By separating the definition from the complex node and than referencing it we achieve that the multiple complex vertices are able to “share” the definitions. We say that d and KE are compatible if the set of vertices EY (of d) has the same cardinality as the ordered set of edges KE.

We can now come back to the example and use the new concept to alter the hierarchical graph representing the full adder circuit (Figure 3.1-4).

It is obvious that in this case the number of needed elements to model the identical topology is irredundant and optimal.

Each folded encapsulated hierarchical graph has a number of definitions. We specify the ordered set (Δ) of definitions (d_i) namely:

$$\Delta = [d_0, d_1, \ldots, d_n].$$

Let all complex vertices of the folded hierarchical graph be aggregated in a set Y. The operation *I*:Y->Δ represents the *instance of* relationship. This relation assigns exactly one definition to each member of the set Y (each complex vertex). For example if :

Δ = {TOP, FULLADDER, HALFADDER},

we can write that *I*(HALFADDER1) = HALFADDER. This relation is represented in our folded graph by the lines that end with arrows. We say that y^d is the instance of the definition d.

Further, the membership of a complex vertex, (y) while i(y)=d_j into the definition d_i is denoted as a composition relationship between d_i and d_j.We say that d_j is *referenced* in d_i. For example FULLADDER = {HALFADDER, OR}

With respect to the set Δ and the operation *I* we define the referenced definitions graph. The definition graph is the graph with ordered edges. Further, we say that the graph G(*N*) is a holder for the set Δ.

The elements of the set are ordered such that:

$$\forall y_i^{d_j} ((y_i^{d_j} \in d_k) \Rightarrow j > k).$$

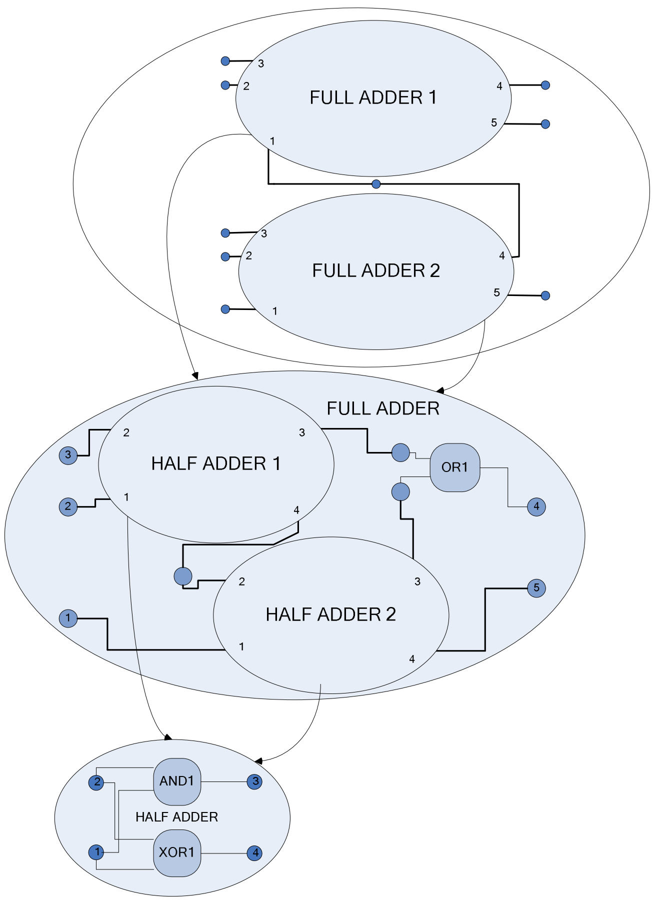


Figure 3.1-4 – Folded Encapsulated Hierarchical Graph Example – The way to represent the hierarchy non-redundantly.

This ordering relation prevents infinitely nested definitions and is actually the natural constraint of the well formed, finite hierarchies.

The element d_0 is known as *the root* definition. That is the definition that is not referenced by any cell.

In addition, we will define several parameters that quantify the hierarchy [36].

h : *height* of the complex hierarchical graph represents the maximal number of levels (or the longest path) between the top hierarchical level and an arbitrary atomic vertex.

l : defines the number of definitions of the folded hierarchical graphs. This is actually the cardinality of Δ.

d : *density* of the complex hierarchical graph. This parameter gives the average number of instances (complex and atomic nodes) in hierarchical levels of the complex hierarchical graph.

f : represents the *number of atomic elements* in the similar flat graph. Having in mind the semantics of the parameters h and d, we can write that : $f \approx d^h$.

n : defines the approximate number of elements (complex and atomic) in the hierarchical design model. We can define it as $n = d*l$.

In the end we will define the gain factor from the fact that we have used the folded hierarchical model as :

$$g = \frac{n}{f} = \frac{l}{d^{h-1}}$$

This result is important for the domain where no flattening of the design is possible.

For instance, the height of the folded encapsulated hierarchical graph in Figure 3.1-4 is : h = 3. It defines three different cells (l = 3). The density of the graph is: $d = \frac{7}{3} = 2.33$, the projected flat graph size is than calculated to equal f = 12.64. Having these values we can calculate the approximate number of elements in the hierarchical graph and the gain factor: n = 7 and g = 0.55.

We will use the quantities defined above to value the graphs during the evaluation of the hierarchical pattern matching algorithm performed on realistic industrial example hierarchical design.

The formal model which was described in this section enables one to store and evaluate any hierarchical design. Through history of EDA there were a numerous implementations and abbreviations of this concept in different program languages. These representations have through time evolved into modern EDA databases that stand behind it and enable persistent storage of the designs together with other important concepts that enhance the employment of this handy methodology.

3.2 EDA databases

The EDA databases implement the hierarchical model and they adapt it so it can be used in different specific purposes [37]. Throughout the history of EDA various database implementations were offered. In such heterogonous environment the interoperability between different tools built on various databases has emerged as a problem. In order to achieve the interoperability the long coordinated standardisation process has been conducted by the VLSI community that in the end coined the proposals for the standards for the EDA database concept. This enables tools from different various producers to work incrementally together coherently in the complex design verification flow.

3.2.1 History

In the history we had many teams working on the topic and they have been resolving and reinventing numerous similar solutions for the standard problems which had to be addressed and implemented into the tools for EDA [38]. Depending on historical period and its trends we had design databases implemented in different program languages [39]. Once the area became more serious and diverse, more and more companies became specialised for the development of diverse EDA solutions. These solutions were step by step accepted and they replaced and complemented a number of solutions of EDA teams of different semiconductor companies. Apart from benefits this brought some problems, as well.

In parallel to useful tools, different databases that hold and model the data handled by those tools were developed. The databases typically employed the relevant hierarchical and other needed concepts. Although similar, they were inevitably in-compatible. The common weakness of these databases was that they were not allowing transparent interoperability between the tools which use their services.

The problem is to make the tools' inputs and outputs compatible and to allow the database to store the results that separate tools produce incrementally (to allow the follower tools to see the changes of theirs predecessors) the proper outcome [40, 41]. The tasks of integration were far from trivial and in literature one can find the introduction of a job description of "EDA tool integrator", or personifications such that the tasks of tool intercommunication are of a calibre of a doctoral thesis. In the environment where the increasing number of companies started offering EDA tools which brought both attractive fast and thoroughly designed solutions and in the same time repellent increasing complexity of the flow integration the most common way of the integrations were loose tool coupling through external ASCII formats for representing hierarchical designs: SPICE, SPEF, GDS, etc.

The ever growing problem amplified with ever increasing design complexities demanded a systematic solution. All these facts have led the top EDA and semiconductor companies to think of and find a solution for the identified problems. The council has been formed to search for the standard for EDA Databases.

3.2.2 Standardization

The standardization attempts started in late '80 when the business analysis confirmed that the investments into tool integration reached more than twice of the sum of investments into the separate application development process. The council named CAD Framework Initiative (CFI) was formed. Their goals were standardisation of the data model that describes electronic circuits and providing the standard API (Application Program Interface) declaration that was written in C language. This first data model has supported exclusively schematics (logical model), while there were plans to extend it towards modelling physical properties of the design layouts.

For different business and political reasons this data model hasn't reached wide usage.

Nevertheless, as the need for the standardised EDA database still existed, second attempt with a slightly changed strategy has occurred starting from 1995., sponsored by SEMANTECH: Chip Hierarchical Design System: Technical Data (CHDStd). This time one of the industrial solutions was solicited and the new standardised model was based on IDM (Integrated Design Model) from IBM. As this second standardisation attempt had, like its predecessor just a document as a deliverable it stayed just on paper as well.

The third attempt that managed to get much bigger interest of the community, because of its availability in both industry and academic domains, its modern design and thorough planning was conduced by SI2. SI2 council proposed a standard for EDA databases: *Open Access*. Open access offers solutions for applications that work both on schematic data and physical data, it is fully written following object oriented concepts which helps its flexibility and understanding. This solution was provided by the reference implementation. This was one of the key reasons for its growing success in both important user domains (industrial and academic). The strategy where the member companies and institutes contributed both financially and by working power was important for the transition of the Open Access project from the vision to the realised database.

The key concepts of Open Access are:

- Standardised object oriented data model and API
- API available to anyone at low or no cost
- Available reference implementation ready for experimental use or industrial application
- Flexible usage of different data domains by the client tools
- Standardised API that includes object-oriented concepts and enables easy interoperability between the tools thus achieving the incremental flow

We will analyse the main concepts of the Open Access database in the next section. After that we continue focussed on the API that is provided by the database to support the applications working with the schematic data representations.

3.2.3 OpenAccess

Open access standardises the data model and the corresponding API for EDA tools [42]. They are capable of storing and presenting folded hierarchical data. The formal representation of the folded hierarchical data model is given in (3.1.2). This model is stored in a persistent store and is accessible by the API which is written in C++. Thus, the API is object-oriented and ready for use in modern EDA tools. For reasons of efficiency, during the application execution a runtime model of the data which was originally stored in the persistent store is built. This is happening transparently to the application and the object oriented API is everything the given application sees. The conceptual architecture that we explain is given in Figure 3.2-1.

An important property of the API is that for each database entity (one instantiation in the persistent store) three different API domains and corresponding objects can be created. Therefore, each entity of the database can be seen through a triade of objects on the application side. In connection with this we have three characteristic domains of the overall API. The domains are:

- Module Domain
- Block Domain and
- Occurrence Domain.

The module domain defines a set of objects and the appropriate models to manage the underlying database data as schematics. Therefore, we see only the logical network, also called a *netlist*, without any physical properties like coordinates, spacing between the objects etc.

Block domain is responsible for the physical side of the design. All objects which model database entities in the module domain can be also seen with their twin objects from the block domain. The difference is mirrored in characteristics of the block domain and the interface of the classes that appear here have somehow different semantics. These objects store the dimensions and all other specific properties of the geometric shapes that form the proper devices, in fact the layout of the design. The hierarchical interface is in this domain a bit different, but equivalent. In the block domain, the connectivity between the levels is modelled in a more simple way as the connections are determined implicitly, by the geometrical position of a given net in the design. Note that in this domain we don’t have hierarchical nodes (3.3.2).

In the end, in the occurrence domain, we have the design represented as a fully unfolded hierarchical database. The type of the model is in this case also logical model, like in the module domain. This domain objects and the appropriate interface is used in cases when the given application needs to personalize the data in different instances of the same subcircuit. The reference implementation of the OA database optimizes the occurrence domain. The objects that represent the occurrences are created on demand, hence only if one traverses the whole instance tree (3.3.2) the corresponding occurrence domain objects would get created. These objects secure the object ID consistency and their size depends on the personalized data they store. This

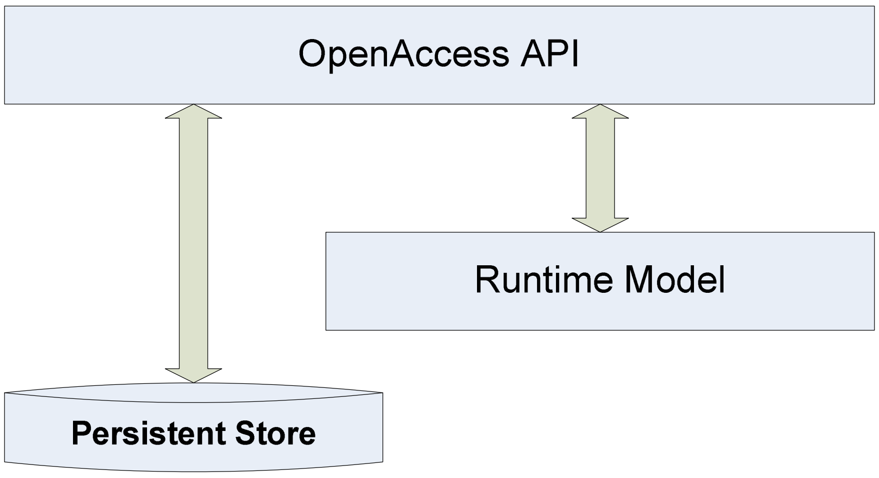


Figure 3.2-1 – The conceptual diagram of the Open Access Database

means that if the two instances of a given subcell are identical their occurrence domain description can be, from the angle of the needed memory requirement neglected. The authors of the reference implementation of OA claim that the typical size of the occurrence domain model is introducing up to the factor of 100 to the original folded model size [38].

This is of course valid for the offered implementation of the database and any optimisation that is being done behind the API would make a difference in performance of the application that is written to the standardised API.

An important property of the OA is the fact that its evolution and further adaptation to the needs of the state of the art EDA application is secured and carefully discussed. The special team called Open Evolution exists. It is led by the engineers from leading EDA companies or the academic world [43]. Any research done in this direction can be discussed with them and possibly affect the standard API or the reference database implementation.

We will, further (3.3), concentrate on the API of the module domain. We will define a simplified case study API which exposes the elements and mechanisms of the object oriented model that are important in order to explain the solution we propose in this thesis.

3.3 NLDB

Let us now define a simple, still functional, hierarchical data model which can store electronic designs that we are going to use further in this thesis. The definition will be given as the UML class diagram. After proper definition of the case study folded hierarchical model, we will make the short overview of the common hierarchical concepts that the tools typically employ to traverse the hierarchical designs.

3.3.1 Object-oriented folded hierarchical model API

In our example designs we will allow fully the concept of hierarchy and folding and for simplicity we will introduce just three atomic elements: MOS transistor (further classified by its model as PMOS and NMOS), the resistor and the capacitor. The API we propose is analogue to the model domain API of Open Access. As they are not directly necessary for the implementations of the concepts we introduce later in the thesis, we will abstract complex parameter mechanisms and the relations of this API to other possible domains (block and occurrence domain in Open Access). We have to stress that for the purpose of our experiments we have used the industrial API model with its full complexity. This gives additional quality to the results we have achieved through tests presented in chapter 6.

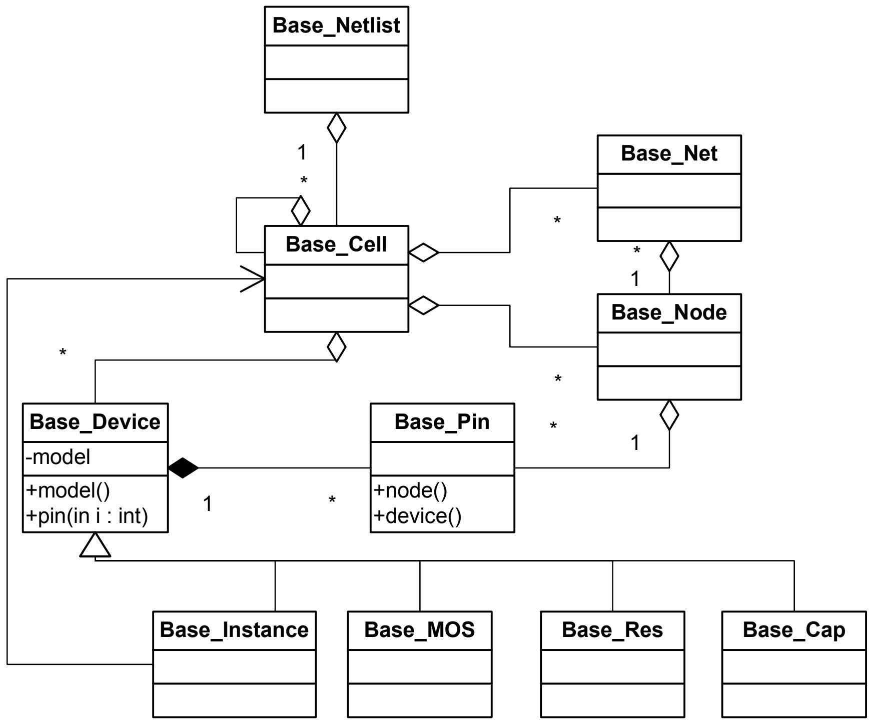


Figure 3.3-1 – UML model of the NLDB database.

The model we propose here as a case study is given in the structural UML (Unified Modelling Language) class diagram in Figure 3.3-1. The UML notation is a common way to grasp different static and dynamic aspects of complex software systems which employ object-oriented concepts. The reader is encouraged to refer to [44, 45] for details about this common notation.

The simple model we define is rooted at the object of the class `Base_Netlist`. `Base_Netlist` is thus a holder class that defines the root cell (root level) of the design, the class of a type `Base_Cell`. This cell is referenced in `Base_Netlist` as a NominalCell. The rest of the cells follow the root cell in the order that corresponds to the order of referencing cells in the design. The order of cells in this vector assures that no cell is presented in this list before any cell that references it. The interface which enables this functionality will be given in 3.3.2. This is the realization of the principle defined in Section 3.1.2, the section that formally defines the folded hierarchical model. The `Base_Cell` cell class object aggregates also some other cells that are defined in its scope, following the SPICE standard. Each cell can aggregate devices (`Base_Device`). Base_Device is an abstract class. Devices can be atomic or, again, complex. The atomic devices that are allowed in our model are Base_MOS, Base_Res and Base_Cap for the transistor, resistor and capacitor devices, respectively. As the model allows the hierarchical organisation, any cell can be referenced in another higher level cell by instantiating the object of `Base_Instance` class, that inherits abstract class `Base_Device` in the equivalent way as other atomic devices do. This is a nice application of object oriented principle of inheritance and polymorphism to handle the concept of vertices that can be complex and atomic, from our formal model of folded encapsulated hierarchical graphs. In order to define which cell (`Base_Cell`) is referenced by the given instance (`Base_Instance`) a link (association) between these two classes is required. Note that, logically, auto-referencing (when the cell references itself) is forbidden. The hierarchy is thus well defined, without loops and finite. Each device has an appropriate number of pins (terminals). They connect the device to the rest of the design. The pin is modelled as a class (`Base_Pin`) that is in the composition relation with the `Base_Device`. The number of pins of the device is precisely defined according to the given device semantics. For instance, a resistor has two terminal pins. The devices have an uniform interface to access the relevant pin by specifying its index. This is achieved via the `pin(int i)` method. `Base_Pin` is on the other side connected to a node. As we have already stressed, the node can aggregate arbitrary number of pins. The node is modelled by the class `Base_Node`. It is defined in such a way that it represents the container of pins, defining the appropriate iterator and specific interface to traverse the set of pins that are attached to it. Therefore we have methods `pin_begin()` and `pin_end()` that return the iterators of the type `pin_iterator`. Pin iterator is, for simplicity not shown on the class diagram in Figure 3.3-1.

Another model entity that we give as a class `Base_Net` is the aggregation of nodes which enables forming of parasitic interconnect networks. This interface is widely used for different applications that include work on parasitic nets and for that reason we include it into our model although it is not present in SPICE. SPICE format has specific extensions SPEF and DSPF that can annotate the original SPICE netlist design with modules that refer to it and enrich it with the data about parasitic elements.

We can conclude that this model is the object-oriented realisation of the formal concept of folded encapsulated hierarchical (bipartite) graphs. We recognize `Base_Device` as the vertex of one sort. It can be further divided into atomic and complex vertices. The second bipartite vertex sort is modelled by `Base_Node` class. We will further analyse different hierarchical concepts that occurred in this model and define the proper interface for them.

3.3.2 Hierarchical concepts in NLDB

The hierarchical model offers one to see each hierarchical level as the proper bipartite graph. If one looks the relation between the levels, the situation gets slightly different while between the levels the constraint that two subgroups of vertices are exclusively interconnected is not relevant anymore. The entities that connect different hierarchical levels are the nodes.

Hierarchical node

In a hierarchical model we can distinguish, semantically, three different types of nodes:

- **Local Nodes**, that have only connections to devices, inside one subcircuit,
- **Root nodes**, that have, apart from local connections, connections down the hierarchy, over the instance pins and
- **Ports**, which are part of the pin list of the given cell, and enable its connection with the contexts in which it is instantiated (up the hierarchy).

Note that the ports can also have properties of the root nodes (connections down the hierarchy), or local nodes. More precisely, the properties of the local node are a subset of the properties of the root node, which are again, in general, a subset of the properties of a port. This classification can be illustrated with an example hierarchy represented in Figure 3.3-2.

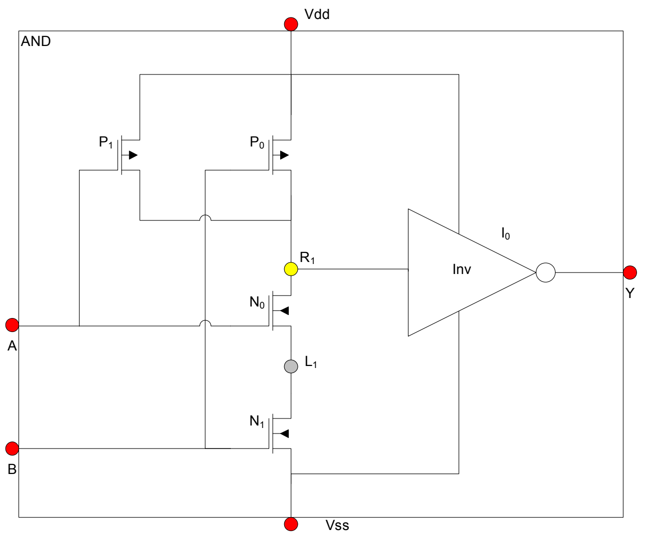


Figure 3.3-2 – Logical AND gate cell, composed of the standard NAND gate and an inverter which is represented as a hierarchical abstraction. Ports are denoted in red, root node in yellow, while the local node is given as a grey circle.

The circuit that is shown represents a logical AND gate. The design given here is hierarchical as the inverter is abstracted in a separate cell. Its definition is, therefore, given independently from the definition of the context in which the mentioned circuit is instantiated. In this example we can distinguish all kinds of nodes given above. Nodes *A*, *B*, *Vdd*, *Vss* and *Y* represent ports, whereas node R_1 represents a root node. L_1 is a local node, which models the connection between transistors N0 and N1.

Ports and root nodes form a structure that we call a *hierarchical node*. This concept thus appears as the consequence of the hierarchical data representation.

The hierarchical node aggregates several atomic nodes (*subnodes*). The nodes are exclusively inter-level connected. It starts with a root node which is its top subnode and that is connected down the hierarchy with a family of other subnodes (that are of the type port). By employing this criterion we form a tree structure that is equivalent to one flat node which would be formed if the hierarchical representation was transformed to its flat version. Note that in the folded model several hierarchical nodes are overlapped. We will use this fact to upgrade the semantics of the standard hierarchical node later, in Chapter 5.

In further text of this section we will point out important structures and algorithms that are standard for the applications that work on (folded) hierarchical models.

Definition tree and the definition walk

Definition tree is a structure that orders the definition levels (subcircuit) of the hierarchical design as a tree topology where each definition (cell) represents a node of the mentioned tree. One node is identified as the root, it corresponds to the top hierarchical level. All subcirucits that are defined inside the given level appear as children of its corresponding node of the tree. This process is recursively repeated depicting the relations between all subcircuits (the way they are defined) of the given hierarchical design. This data model is useful for dumping the hierarchical model, for instance into an ASCII file (following the specific ASCII file format, hence SPICE). Note that this model does not verify if the defined subcircuits were also instantiated, or they just exist as pure definitions.

For a definition walk, we define the templated algorithm which recursively traverses the definition tree and performs generic functions before and after recursion. By defining this traversing algorithm generic, one promotes the walk as a standard API algorithm that can be defined as a friend function in the world of object-oriented languages.

Instance tree and the hierarchical Instance walk

The instance tree has a structure that is similar to the definition tree. In this case we nevertheless present each instantiation of any definition. This unfolded structure therefore has the given definitions repeated as many times as they were instantiated. It is not always possible to create statically the whole instance tree. Of course, an alternative to its static creation is performing a recursive algorithm where, while traversing the instance tree, it collects all the relevant personalised data (relevant just for a given instance) that is used while analysing the given instance of some cell. This approach is known as the instance walk. The instance walk can be extremely time demanding and thus unacceptable.

Instance walk is the simple trade between the hierarchical and flat algorithms. One can upgrade flat algorithms to hierarchical in the most trivial way using this tree/walk. The reason for that is that all extrinsic details and attributes that are defined by the path in which a specific device or the whole instance is given are there resolved. The application can be wide, but the efficiency is not big as although the work is being done hierarchically which demands solving some of the issues concerning the communication between the levels and although the results that are generated by the tool employing instance walk are aware of the hierarchy (original folded hierarchy) this approach is even less efficient than flat algorithms.

Referenced cells tree and graph

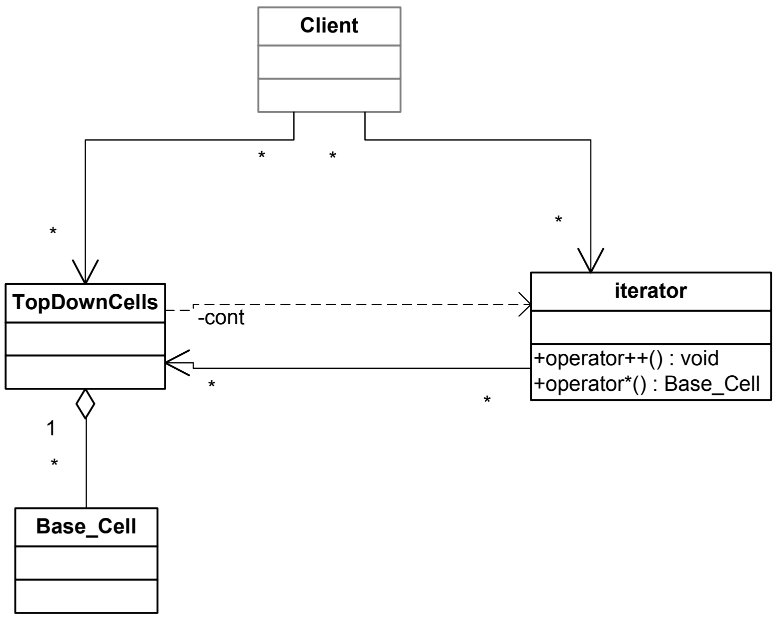


Figure 3.3-3 –Top-Down cells container, the container and the iterator that allow one to iterate all cells of the design top -down and bottom-up.

If we collapse all instances of the given definition inside a given cell into just one representative connection (which than loses path information) we obtain the Referenced Cells tree. In this structure we can non-redundantly access all different definitions of the cells that were instantiated in a given cell. This structure is welcome to perform the operations such as determining the hierarchy height or for algorithms that work on all root-nets (nodes).

If we add the information which determines in which cells a given cell is instantiated, we upgrade the referenced cells tree to the new abstract structure – referenced cells graph. This data structure allows also looking “up the hierarchy” from each of the cells defined in the design. It is very useful for different algorithms that need to take into account several hierarchical levels in the same time while calculating their relevant results.

Top-Down Cells

TopDownCells represents an alternative way to approach the defined cells of the given designs. In this case we introduce the ordered vector that offers a bidirectional iterator that can traverse all cells that are referenced inside the design. The order of iteration is analogue to the ordering of cells given in section (3.1.2). For this purpose we define a class TopDownCells to serve as a container of the ordered references to the different cells defined in the given design. The object oriented architecture of this container is given in Figure 3.3-3.

The iterator can be set-up to give the cells top-down and bottom-up. These walks are used in different hierarchical algorithms for which the information is being passed over the referenced cells graphs.

3.4 Personalization

As we have stated, some algorithms prefer the style of just traversing definitions while some demand either instance tree or fully flattened netlist. Those are typically the applications that need to change some of the instances and contexts in the hierarchical nelist just locally, valid exclusively for a single instance path, or one occurrence in the netlist. In general it is further possible that some of the changes that originally belonged to the same definition are both identical. Thus, the optimal way to present this concept would be to regroup instances and introduce new definition for the two which have left the prior group and leave all other instances linked to the first group. This problem is known as a problem of personalization and since industrial hierarchical folded designs include parameters as well, it is really essential to have a solution for these problems.

These problems were recognized by OA development team and the personalization problem is addressed by introducing the occurrence domain [38]. The occurrence domain is some kind of optimized instance tree that is created on demand. The way the instance tree is stored is also optimized and the new definitions are stored only in a case where some differences between the master objects and their clones exist. It is claimed that this occurrence domain introduces memory requirement overhead that is not bigger than two orders of magnitude. However, this overhead depends on the task a given tool using occurrence domain is performing.

Another known research that appears in literature and addresses the problems of the personalization is done by Jones et al. [36, 46]. In the conference paper they consider various strategies to perform the personalization. First trivial strategy is full development of the data into the instance tree (unfolding), second is employing a dictionary that stores the personalized data and the third is done via partial unfolding where each changed definition occurrence gets an appropriate copy in the referenced cells graph.

An alternative to these approaches, the concept of variants that was used to support our contribution is given in Appendix A.

3.5 Polymorphic hierarchy

Hierarchical representation of a given design is not unique. It is in some sense polymorphic (associative). We can group elements of a complex system in different ways and achieve different hierarchical interconnected levels. This can be illustrated by the famous Indian face picture given in Figure 3.5-1. Is it actually a face of an Indian, or is it an Eskimo entering the cave? This depends on the way we interpret this very same picture hierarchically.

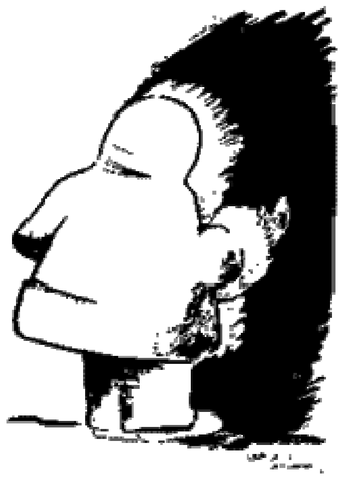


Figure 3.5-1 – An Indian or an Eskimo?

If we link the neck, the mouth, the nose, the eyes and the forehead into a face, we see an Indian with all his other attributes. If we on contrary in our mind link the

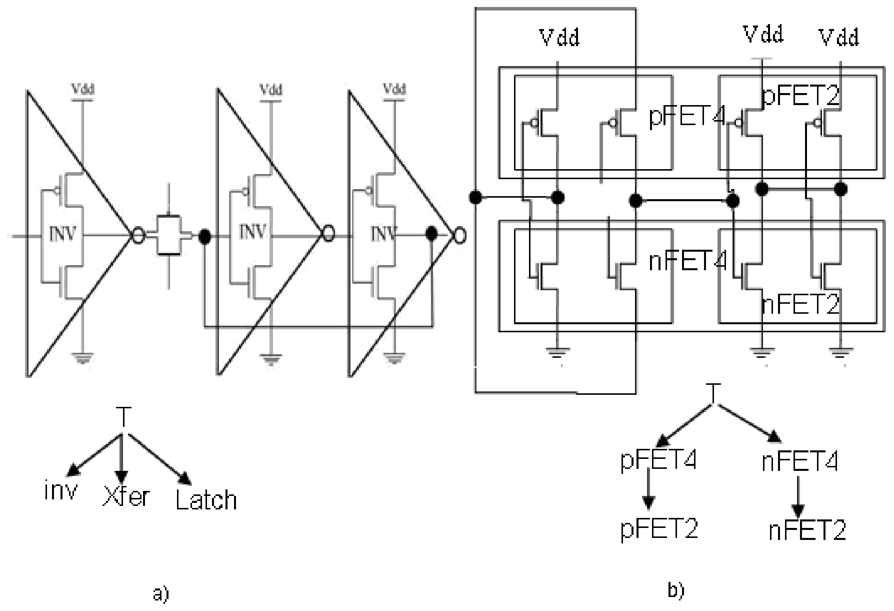


Figure 3.5-2 – Identical driver and latch circuit that has two different hierarchical layouts. (a) shows the hierarchical layout that is more close to thefunctional side of the circuit, while (b) shows that hierarchy that groups the devices in the fashion that is shows some technological, physical properties.

legs, the wrinkle on the coat, the elbow and the head forming the back of Eskimo, we see him entering the cave. The only difference in the picture is in the way we hierarchically interpret it.

In order to make this example closer to our topic we show also two different hierarchical interpretations of the identical circuit – a latch with the corresponding transfer gate and the driver. In Figure 3.5-2, under (a) we show the hierarchical organisation of the circuit that is close to its functional characteristics. When CMOS electronic designs are printed into silicon wafer, usually the layout is organised in a specific way that all PMOS transistors are printed in a line and all NMOS transistors are printed in a parallel analogue line. After this they get properly interconnected in the repetitive step of applying and developing resist layers and etching. With respect to that we, just for illustration, organise the elements of the identical circuit in this other more "layout like" way. This is shown in Figure 3.5-2 (b). Although the circuit is the same, we form completely different hierarchical topology. The definition trees of both hierarchies are different, while the flat circuit they represent is identical.

We can use this ambiguity in the hierarchy and adapt it to the tool that is supposed to use it. If we have the way to flexibly represent the hierarchy we can solve some common problems that the tools typically face and make the tools much more

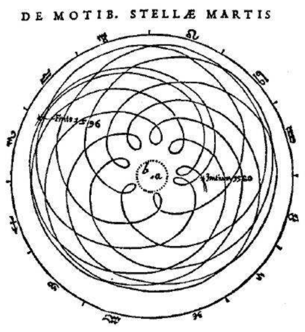


Figure 3.5-3 – The path of planet Mars in the geocentric system. The analogy with the difference in algorithm complexity, according to the hierarchical layout.

comprehensive. In this sense, the we make the hierarchy friendly to the user application and by preparing the data the application builder can relay on certain constraints and solve the specific problem with much easier algorithm. We can compare the complexity of algorithms that use hierarchical data to the problems astronomers were facing up to XVII century. Figure 3.5-3 shows the path of the planet Mars seen from the heliocentric system. Although they were also right, one can imagine how much unnecessary efforts were spent in order to track and predict so complicated path.

Similarly by changing the hierarchical layout of a given design, we want to provide the application with the right “glasses” so the data is seen in the best way. Therefore, we want to populate hierarchical levels of the given design flexibly, group and regroup different elements together using exclusively standard API methods and entities. We achieve this goal using advanced object-oriented concepts defining the framework that utilizes the presented concept vision in the following chapter.

4 Hierarchical Multilayer Views

4.1 Introduction

In the previous chapter we have presented the model of hierarchical abstraction whose advantages are employed in order to efficiently store electronic designs. We have formally presented the hierarchical model as a folded encapsulated hierarchical graph. In addition, we have given an overview of the development of the databases that implement the formal encapsulated hierarchical graph model. These databases include a variety of advanced concepts that help the interoperability between the design tools that are shared by the design process. Modern EDA databases are object oriented and they offer a specific API that can be directly used in design tools. These databases are also turning to a growing standard - Open Access (OA). We have, further, analyzed the API and shown the common algorithms and data structures that are suitable to explore the folded hierarchical designs. In the end we have pointed out that the hierarchical layout of a single design is not unique, but polymorphic. Hence, a given flat design can have a number of different hierarchical representations that are synonymous.

This serves us as an idea to extend the standardized API and adapt it in order to support the different views on hierarchical data. By employing the concept of views we want to group (regroup) different hierarchical entities and see the design with the changed hierarchical layout. This concept considers the extension to the API for the module domain (3.2.3), the domain of the standardized OA that represents the schematics (logical design). We propose this extension as a possible upgrade of the OA standard. We will demonstrate new concepts on the object oriented API analogous to OA standard which is defined in (3.3). The specific architecture of the object oriented API that we will propose further in this chapter allows flexible views on the hierarchy of the schematics (logical designs).

The designs that we will transform are themselves hierarchical. They have the initial hierarchical layout which is changed by specific modules (the implementation of the views) written for the standard API. The concept of the hierarchy groups certain entities together and defines clear borders between different such groups. This is something that can be used as a favourable constraint by the tools written to process the hierarchical data.

We want to employ above sketched mechanism to adapt the actual hierarchy to

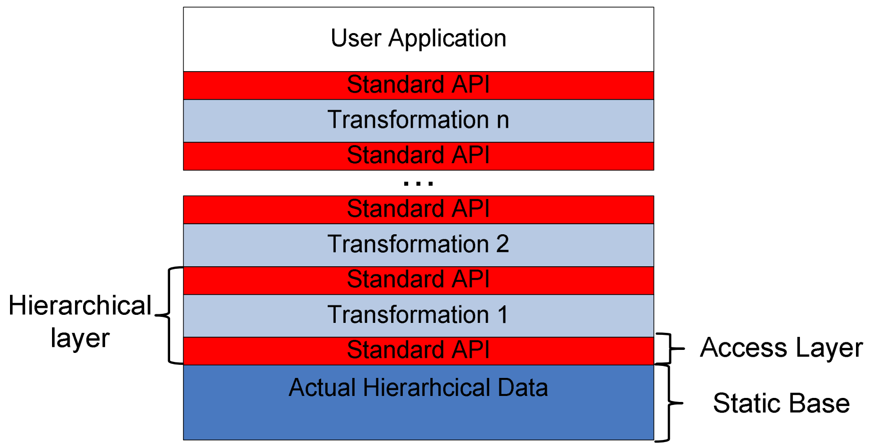


Figure 4.1-1 – Chaining the transformations of the Hierarchy. Arbitrary number of transformations are allowed as they are all compatible with the AL (Access Layer).

the algorithm that is processing it. We can therefore allow the tool to “see” the data in the most favourable way. These conversion steps are, in the current state of the art, usually part of the given tool’s implementation or are realized as the preparatory steps before the tool evaluates the design data and generates the proper outcome. In some cases this transformation is complex, meaning that it can be split in different simpler pre-processing steps. If the data is statically transformed (new equivalent design is created) by the pre-processing step this “chain” of corresponding designs becomes bulky to handle. For instance back-annotating the results that the given tool produces can become a considerable task. In some cases these transformations are even not possible. A typical example is flattening an unbearably big hierarchical design. Flattening is also a hierarchical transformation, as in this case we consider a flat design as the special case of the hierarchical design (that has only one hierarchical level).

Since a number of different (atomic) transformations that are common for the different algorithms can be identified in order to achieve flexible view creation we want to allow another concept for the views, the concept of *layering*. This means that the final, application friendly, hierarchical layout is prepared by employing a number of views, linked one after another. This is illustrated in Figure 4.1-1.

In the figure we see the actual hierarchical data given in the bottom. The data is accessed by the standard API. On top of it we have the first view. This view takes the actual hierarchical data reading it using the standard API, reorganizes it and offers the same methods and entities, populated in a different way for any user algorithm (including another view). Since the vocabulary hasn’t changed and we still have all attributes of the (rearranged) hierarchy given as a standard API, we can immediately apply another view on top of the initial one. The process can be repeated several times and the user application in the end can get the handle to the standard API that populates the hierarchical entities in a specific constrained way.

We realize the requirements by employing object oriented concepts. Thus, we separate standard API as the group of pure abstract classes that is named: *Access Layer* (AL). AL defines the vocabulary to represent the hierarchical design. It consists of entity and method definitions, together with inheritance hierarchy between the entities. Of course, no implementation (hence, no class has any attributes) is offered here. The actual hierarchical data can be defined as a *static base* layer. Here, we have static implementations of the promised interfaces of the AL. By static is meant that all the entities and methods that implement the API are in this case populated with realistic values.

Further we define a section:

standard API - Transformation n - standard API

as a *layered view*. Therefore, the layered view reads the given appearance of the hierarchical data from the standard API and reorganizes it by re-implementing the same standard API.

We will in further text of this chapter, with greater detail, introduce the mentioned entities to the reader. Therefore, section 4.2 describes the Access Layer, section 4.3 static base, while section 4.4 presents the definition and standard architecture of the layered view. We conclude this chapter with several examples of hierarchy transforming layered views (section 4.5). The whole chapter sets the context for the explanation of the Virtually Flattened View (VFV) that is presented in the next chapter. VFV, one possible realization variant of the general concept of layered views on the hierarchical data, is a part of the proposed solution for the problem of hierarchical pattern matching.

4.2 Access layer – pure abstract interface

In order to employ the object-oriented concepts to support interchangeable and combinable view on the hierarchical layout of the design database, we upgrade the overall design of the standard API (NLDB). We introduce Access Layer (AL), which serves as a pure abstract interface to NLDB data. It consists of exclusively pure abstract classes and pure virtual methods. The pure abstract interface includes all necessary inheritances but no implementations of the methods including the references between the entities. For example in this layer the association between the instance object and its definition is not realised, but just promised by the appropriate pure virtual method.

The AL consists of the interface *entities* (building blocks) and interface *methods* (which are distributed over the entities, or defined as friend methods). The availability of different building blocks that form an interface depends on the type of the view which implements the AL. We will call the interface methods also Common Standard Interface (CSI). Of course, the building blocks of the AL represent the complete set of classes and methods capable of describing the folded hierarchical concept.

Note that the implementations of any method come first at realizations of the Access Layer. We have presented the class diagram of the AL in Figure 4.2-1. It is similar to the NLDB example of the API for the folded hierarchical model. As it is obvious, in the AL class diagram compared to NLDB class diagram, all aggregation and association links are missing while the inheritance lines are still present. This is due to the fact that in this layer we exclusively define the vocabulary that gets its proper implementation later.

AL classes define following entities:

- Netlist (`Access_Netlist`)
- Cell (`Access_Cell`)
- Device (`Access_Device`)
- Instance (`Access_Instance` – separately shown because of its special semantics, although it belongs to `Access_Device` class hierarchy)
- Pin (`Access_Pin`)
- Node (`Access_Node`)
- Net (`Access_Net`)

The roles these classes play are analogue to the roles of the relevant classess in NLDB database API. For this reason we will give them just briefly here. Please refer the section (3.3) for further details on element semantics.

`Access_Nelist` is the pure abstract class which plays a role of the holder of the design. We can refer to the top level of the design from it and further access all cells of the given design in the top down and bottom up order. In realistic databases this class stores different global parameters of the design: physical configurations, such as nominal temperature of the chip that is described, special element semantics

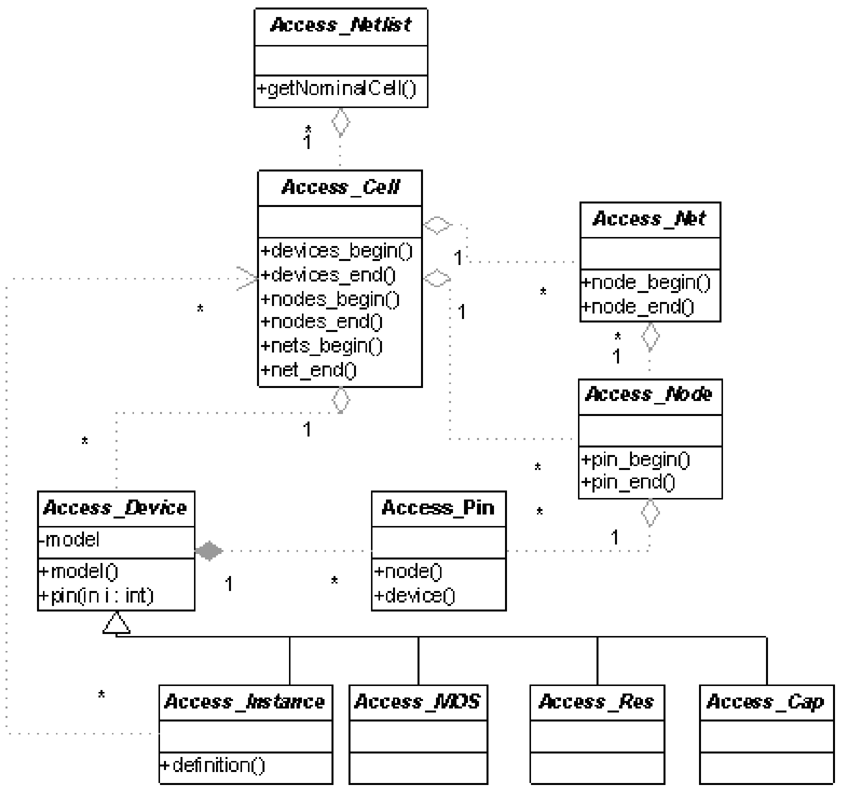


Figure 4.2-1 – Access Layer class diagram. AL contains exclusively pure abstract classes.

(some cells called standard cells come with both structural definition and model information), naming conventions etc. In our case this will be left out from the CSI.

`Access_Cell` models the subcircuit (a hierarchical level) of the given design. This class defines the proper interface to access all devices, nodes and nets in the design.

The `Access_Device` class defines the proper interface for modeling devices. Therefore we have the methods to access its pins and the model. Important specializations of `Access_Device` are present in the AL. Still, none of the methods get realized in these specializations neither. The inheritances are here just to define the necessary entities which any implementer of Access Layer has to realize and of course to add the specific part of the interface, characteristic for the `Access_Instance` class, the `definition()` method. This method is declared to return the pointer to the instance of `Access_Device` class descendent. The methods which return pointer to the pins of the device are declared to have `Access_Pin` as the return value. This class, thus, defines another entity of the AL. It models the terminals which connect any device (instance) to nodes, modeled by the `Access_Node` class. This class allows the interface to iterate over all the pins attached to the given node. Additionally we define the entity net, to model the parasitic networks that agregate a number of nodes that are interconnected with the parasitic resistances, with its class `Access_Net` and the appropriate public method definitions.

We can conclude that the AL defines a proper interface (entities and methods) that can model folded hierarchical designs. The pure abstract classes of this layer exclusively define the interface to program to, but without any implementation. This is the vital design decision in order to allow polymorphism and exclude any overhead in memory layout of the model objects. By isolating AL as a pure abstract layer and programming applications (or views) to it we assure completely transparent usage of any mixture of layered views that prepare the data for the user application. The interface will be equivalent no matter how many layers and which kind of mixture of layers we have applied. Any attribute fields are introduced precisely in places where they are needed, e.g. for purposes of implementing the static base or any specific layered view.

In further text, we will show first the simple architecture of the static base which makes the new architecture functionally equivalent to the NLDB. After that we give the standard architecture of the layer followed by several examples of the hierarchical transformations that the different layers can give.

4.3 Static base

The static base is a fully materialised in-memory representation of the design's actual hierarchy. It is analogue to the standard architecture of the hierarchical database NLDB. All the methods are therefore implemented in place and behind the interface methods we have real data structures storing the attributes of the entities of the database together with their relations. The difference between the static base and any standard EDA database is that it just represents the occurrence of the pure abstract CSI (it is written as the realisation of the AL). We describe this relation in Figure 4.3-1. In it one can see the example inheritance hierarchy of the realisation of the class `Access_MOS`. In the figure one can see the layers of the database separated. All classes that belong to the AL are given in the right diagonal stripe, while the analogue classes of the static base, their mutual relations and the relations with the AL are given in the left diagonal stripe. Both of the layers belong to the NLDB definition.

The hierarchical relations between the classes of the mentioned layers are complex. Multiple inheritance is employed in the definition of the class `Base_MOS`. Let's analyse this class diagram. The classes `Access_Device` and `Access_MOS`

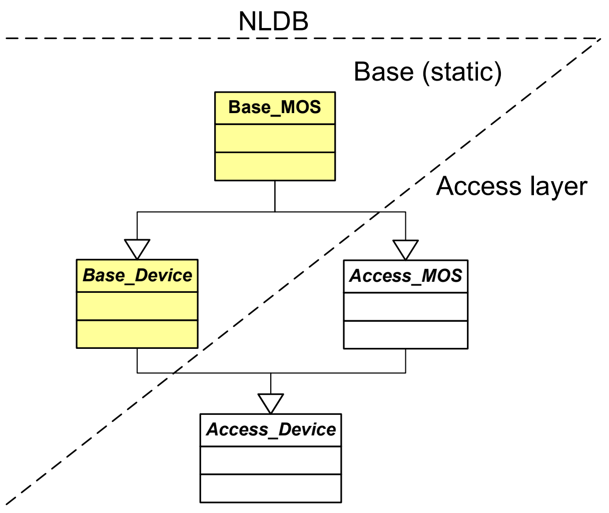


Figure 4.3-1 – static base vs. Access layer

define all necessary interface methods and they themselves are the pure abstract definitions of the corresponding entities of the database API. The `Access_Device` class is inherited by the abstract class `Base_Device`. This class is analogue to the implementation of the `Base_Device` in (section 3.3). In order to be able to keep the inheritance relations and polymorphism between both `Base_Device` and `Base_MOS` and `Access_MOS` (`Access_Device`) and `Base_MOS` and still be sure that all CSI methods of Access_MOS get their implementation, we include the inheritance link from Base_MOS along the other inheritance line to Access_MOS. This inheritance style is typical for different classes of the static base.

Static base behaves equivalently to standard realisations of the object oriented databases and represents therefore pure extension without any negative effects. The only overhead that is present while using such architecture is when one applies the algorithm that is written to the AL interface on static base. The overhead is related to the implementation of the polymorphism in the (C++) object oriented program language. This overhead is however neglectable, especially if we have in mind the importance of the complete transparent usage of static base for any application written for AL. Note that this is also the default implementation of the access layer and that any algorithm that is written to work with the layer entities can be run on static base. The model implemented as static base instance is in the root of any chain of layered views.

4.4 Layered views and their object-oriented architecture

The goal of the hierarchical layered view is to regroup entities that are offered through the standard interface (Access Layer – AL) by re-implementing the CSI methods. The view on the hierarchical data is the group of classes that are placed into the taxonomy hierarchy, which is rooted by Access Layer classes. Each view object plays the role of proxy/decorator for its source object (group of objects, distributed to underlying layers). Therefore each layer is characterized by its *source view*, from which it acquires the available information about the given database element(s) that it decorates. The view implements (realizes or if necessary overrides) the CSI methods. Typically, methods can alter the semantics of already realized CSI methods from previous view layers, just forward the calls of the methods to get the information from deeper layers of the database, or simply use the old implementation of the method by polymorphism.

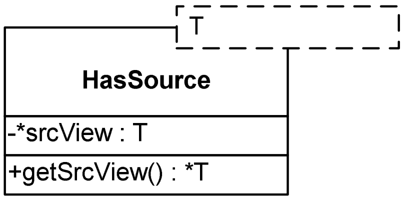


Figure 4.4-1 – Templated HasSource class defines the layering property. The class is typically an ancestor of any layer class.

Every implementation is a unique problem but the overall architecture of the view that serves as some kind of framework for the engineer that is providing a custom view is uniform. We will describe this standard architecture in the further text.

Each hierarchical layered view:

- defines a set of entities inheriting the classes of Access Layer or its descendents,
- has appropriate links to the source objects for each view object
- is written to the AL,
- implements the CSI interface,

- defines and implements the private interface,
- defines private data-structures,
- defines optionally additional public interface.

As we have stated each view defines (and realises) all necessary entities by inheriting them from already existing classes of access layer (or some other layer). Each of these objects is linked to the source view. Through these references the view fetches the data about the hierarchical design whose hierarchical layout it transforms. The mentioned link is modelled as a templated class `HasSource<T>`. This class stores a pointer to the templated type T and defines and implements the appropriate interface method (`T* getSrcView()`) to get the pointer to the underlying object. Therefore, we define a templated class that exclusively models this property leaving the templated type to be decided upon its employment in some usage context. The class diagram of the class HasSource<T> is given in Figure 4.4-1. The implementation of the class is simple. There is an attribute (`sourceView`) of a generic type T* that privately stores the link to the lower level and the public method `getSrcView()` which returns the pointer to T. This method augments the interface of any class that inherits HasSource.

We use above described templated class to build any of the view classes. Typically view classes are inherited multiply. One inheritance link leads from some class of the access layer (or some realization of it) and another privately inherits from `HasSource<T>`. The inheritance is private in order to turn off the polymorphism between `HasSource<T>` and the given `View_<class>` as passing the object that

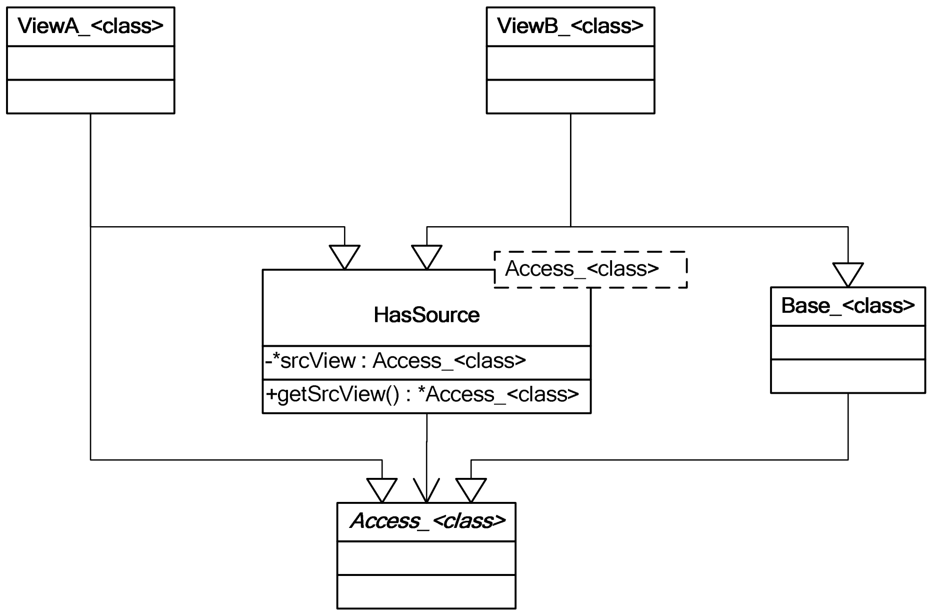


Figure 4.4-2 – View positioning relative to AL layer, static base and other views. View A inherits directly from the relevant pure abstract class, while View B also inherits some properties of the static base.

belongs to the view to the pointer to a type `HasSource<T>` doesn't have any semantic meaning. Furthermore, by applying the private inheritance we save the method `getSrcView()` exclusively for the layer methods as it is not part of the CSI and still is important for the implementation of the view. Note that for the object oriented languages which do not allow multiple inheritance the alternative approach would be to define HasSource as an interface and than provide the links implementing them in every actual class occurrence. This is not as elegant as the solution with private inheritance.

The method `getSrcView()` is a good example of the private view interface member which appears in any layered view. The typical positioning of the view which shows the multiple inheritance is given in Figure 4.4-2. In the figure we see the class diagram which shows two different cases of view positioning. View A is positioned directly above AL. It, therefore, inherits (privately) the class `HasSource`, to get the layering property for the objects of the view and publicly class Access_<class>. By this we gain the polymorphism property and are able to use the object of the view class in any place where a pointer to Access_<class> is expected. Of course it is necessary that the ViewA_<class> properly implements all methods promised in the pure abstract Access_<class>. Note that <class> stands for any appropriate AL entity. We use it in order to allude to the fact that the described architecture is needed for any of the view classes. View A inherits directly form AL, which means that it doesn't need any of some possibly similar view (static base) implementations. On the contrary, exactly this is the case of the example of the positioning for View B. It inherits the class HasSource, equivalently as the View A, but also inherits from static base. In this way when overriding the methods of the static base one can reuse some implementations of it, some attributes that exist in it and also be able to simply use specific methods from static base without overloading them. This view positioning type is used in our implementation of the Virtually Flattened View, which is given in next chapter.

As we have mentioned, each view can implement its private interface and attributes. They are helping structures and methods for the goal of re-implementing (regrouping) the entities of AL that the view offers to the further user.

In this point we can discuss the fact that the view can also expose some new interface and augment the standard CSI. This is for instance good if it is necessary to include extra properties for the NLDB objects for the purposes of the given algorithm. An example would be the interface to store and retrieve the types for nodes of the design. Note that these methods would be visible exclusively if the view which defines them is appearing as the last level, directly under the user algorithm layer. This is the negative issue and it can be an argument for the eventual redefining the CSI where the given methods would be included as a standard.

The described architecture gives a lot of freedom to implement different sorts of transformations. We will present some, as a vision, in the following section.

4.5 Examples of views

So far throughout the current chapter we have defined a new framework which enables hierarchical transformations of the EDA schematic designs. We have specified the pure abstract interface by whose overloading and inheriting its pure abstract classes we can write different views that enable hierarchy layout transformation. In this section we are going to present the motivation and conceptual ideas of implementing various views following the defined framework.

Equivalence class abstractor

The idea behind this view is to group certain elements of the NLDB database as an alternative to subcircuit cells. This is important in various algorithms, which helps treating a group of the devices that are not anyhow explicitly abstracted by the database itself in order to optimize given user algorithm implementation. Equivalence classes are applied in different ERC checks in order to optimize node type propagation. On the other hand this concept can enable building of the nets. More precisely, if an generic equivalence class abstractor is offered, which uses specific generic CSI methods to represent the abstraction, building a net abstractor would be combining this level with an additional which would only serve as an adapter and wrap the generic method calls into specific interface that enables usage of nets in NLDB.

Variant generator

By implementing specific view, variant generation (Appendix A) can be hidden behind the CSI interface, where each variant would be seen as a separate cell definition. For this reason in further text, we will use the terms variant and cell equally. The difference between them is just in the way the given entity is realised: if it is directly defined in the model or isolated as a variant of the cell during the variant creation process.

Virtually flattened view

The size of the hierarchical data can be many times smaller than its unfolded (flat) version. This is especially pronounced in the case of DRAM memories. In this case it is obvious that, as the data is highly folded, algorithms to work on it directly would be extremely complex, in some practical way, impossible, as developing the algorithm for each specific application/task would demand very long periods of time.

As the implementation of some algorithms (pattern matching, for instance) is very difficult for hierarchical netlists, the methodology where one flattens the netlist first and than operates the tool on fully flat netlist was often used. For big examples this method is not efficient. It consumes a lot of memory and time; further the contexts the algorithm works on are highly redundant. Our idea is to provide a specific view on the hierarchical data that can provide flat flexible view on it that is friendly to the user application. This view is the topic of the next chapter.

5 Virtually Flattened View

This chapter brings one possible usage of the general concept of hierarchy managing layered views established in the previous chapter, the object-oriented virtually flattened view (VFV). The vision of this view is given in Section 5.1. The further text of the chapter gives a description of the high-level architecture of the VFV throughout section 5.2. Therefore, the main classes of the view are conceptually discussed, together with the proper explanation of the relation between the view and the Layered NLDB database. Further, section 5.3 gives us an overview of the architecture and semantics of the entities that model the materialised flat data portion itself together with precise description of the typical methods and mechanisms that enable proper flat data portion creation. The discussion of the concept of dynamic iteration over representative devices is left for the section 5.4, followed by the description of the general object building strategies abstracted in a specific builder class in 5.5. The complex concept of the mechanism that assures the consistency between the flat data portion and the hierarchical database and determines the flat netlist space projections of the flat data portion is given throughout sections 5.6, 5.7 and 5.8. Further, the committing process and the mechanism that allows consistent usage of changes of the hierarchical topology together with the original database data which has stayed intact is given in the sections 5.9 and 5.10. The application of the view we define here on the problem of search oriented subcircuit recognition (chapter 2) is given in chapter 6.

5.1 Introduction

The Virtually Flattened View is a type of the hierarchical netlist database layered view. The Hierarchical Layered Views are subject of the previous chapter. The goal of the Virtually Flattened View is to present parts of the hierarchical netlist data in the flat fashion. Therefore, the user (algorithm) accesses the netlist as if it was statically flat. It can iterate over different devices of the design and navigate the local neighbourhood, from once acquired device to arbitrary neighbouring design device, orthogonal to design's hierarchy. In order to achieve this, the algorithm materializes flat data portions that would represent the part of the design which is of interest.

How is this possible, having in mind that the hierarchical concept describes, sometimes, highly redundant flat data, whose materialization (flattening) requires unbearably large memory and whose analysis would require extensively long runtime?

In order to still be able to take advantage of the flat view, some assumptions have to be taken into account. Many algorithms use, typically, local portions of the design data for their calculations. Thus, the algorithm would acquire a handle to a certain device, as the starting point and further examine its local neighbourhood. After evaluating this portion of data, the algorithm would create the conclusion records that represent the tool's output. There are numerous examples for this concept: search oriented pattern matching, parasitic net evaluation and reduction, etc.

We can achieve our goal, if we take this tool preference as a constraint that is not going to handicap the algorithm execution flow anyhow.

Two main constraints are to be established in order to make the concept of the Virtually Flattened View feasible. First, the iteration over the design elements is conceived in a specific way. Hence, the user can iterate only over all context-representative devices, not redundantly over all design devices. This approach is spe-

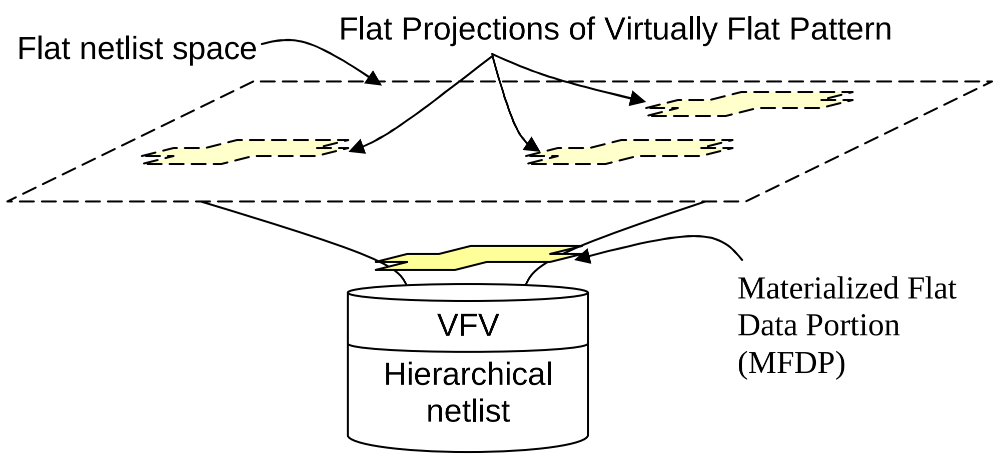

Flat netlist space
Flat Projections of Virtually Flat Pattern
VFV
Hierarchical netlist
Materialized Flat Data Portion (MFDP)

Figure 5.1-1 – The conceptual diagram of the Virtually Flattened View. The local pattern that is being created stands for a number of identical appearances of itself in the flattened netlist space.

cific, however semantically correct. As we have information about the hierarchical properties of the design data, if the user algorithm creates some result, it could be committed in a way that it is valid for all appearances of the given pattern in the flat design version. An additional, important consequence of this approach is a much faster expected execution time, compared to the pure flat approach. By working only on representatives, the algorithm skips all redundant, identical appearances of a given algorithm result. Still we must not forget the overhead that the algorithm that controls the view introduces. A second constraint of the concept we are proposing is that the object identifier consistency is secured only inside an interconnected flat data portion, formed strictly by navigating in the neighbourhood of the starting object, acquired by iteration. If this is taken into account, we have another, implicit, more or less flexible constraint. The size of this local neighbourhood has to be acceptable from the aspect of the available system memory.

Therefore, considering the constraints given above, the implementation of the view must be able to create (materialize) small portions of the hierarchical database data in the flat fashion and to maintain the consistency between this flat data portion and the original hierarchical database data. Each materialized flat data portion (MFDP) corresponds to multiple instantiation places in the flat netlist space, as illustrated in the Figure 5.1-1. This means that physically only one pattern exists, but it is valid for multiple (as example illustrates, three) different contexts in the flat netlist.

Described above is the primary functionality of the Virtually Flattened View.

The limitation that the object identifiers of the locally flattened view are not permanent can be indirectly addressed. As any direct comparisons between the objects of two materializations of the view are not possible, the dependency between two different flat data portions is rather achieved by altering the primary hierarchical data using the corresponding MFDPs that the view has generated. This comes as a second functionality of the view. In every moment, the current group of objects representing the materialised flat data portion can be committed to the hierarchical netlist as an instance that is placed in a given optimal hierarchical level (as deep in the hierarchy as possible). This powerful concept, which enables altering the hierarchical data, by using exclusively standard common interface “vocabulary” (`Access_Cells` and `Access_Instances`) can have a wide application, as it will be shown later.

5.2 Virtually flattened view - high-level architecture

In the previous chapter, we have defined the proper framework to alter the actual hierarchical topology of the hierarchical design by strongly separating the interface entities and their method definitions from any implementation. Therefore, we will now present the concepts of the Virtually Flattened View, with the respect to this framework, employing advanced object-oriented mechanisms. Hence, in continuation we show the high-level architecture of the Virtually Flattened View and it's relation with the Layered NLDB.

The main functional units of the virtually flattened view are given in Figure 5.2-1. The view is inherited from the NLDB static base. Among other advantages that will be pointed out, this gives the opportunity to reuse parts of implementations of the standard netlist hierarchical database. The bearer of the view is the class `Virtual_Netlist`. It inherits the class `Base_Netlist`. This is done, as the virtually flattened view should provide a user the feeling that he is working with a regular flat netlist. Therefore, passing the object of the class `Virtual_Netlist`, instead of the (expected) instantiation of the class `Base_Netlist` allows the user algorithm that is designed for flat NLDB data, to transparently work with the virtually flat data representation, employing polymorphism.

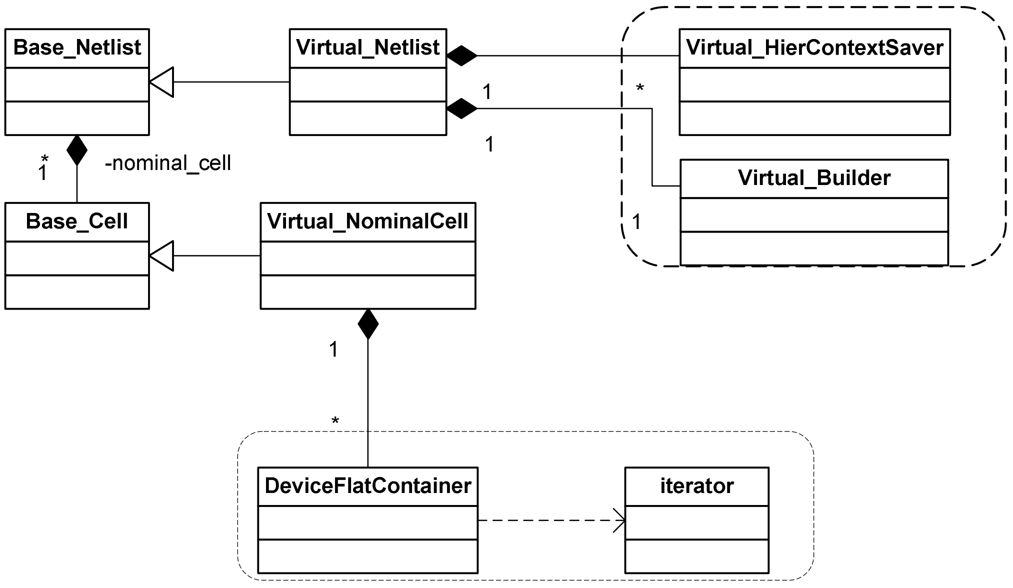


Figure 5.2-1 – High level architecture of the Virtually Flattened View. The view mimics the flat netlist. Thus, It has a Virtual_Netlist, Virtual_NominalCell and the DeviceFlatContainer classes. Virtual_ContextSaver and Virtual_Builder are given also to model the overall VFV creation and exploatation process.

The class `Virtual_Netlist` has its nominal cell, as well. It is, moreover the only cell in this virtually flat netlist. Note that, in general, the nominal cell of the virtually flattened view does not have to be the nominal cell of the hierarchical design. Just a part of the hierarchical design can be, by employing the virtually flattened view, seen as flat. This flexible property can give one a chance to, for instance run a flat algorithm on a part of the hierarchical design that is of relevant interest, or to, by employing the committing functionality, rearrange the hierarchy of a given part of the hierarchical design.

The nominal cell of the Virtually Flattened View contains further an instance of the class `DeviceFlatContainer`. This class models the sophisticated concept of the iteration over irredundant, representative devices of the virtually flattened design. Therefore, the class `DeviceFlatContainer`, as a container collects all different device representatives from each NLDB device container at any hierarchical level of the design. More precisely, it serves in some sense like a façade between the group of containers in the hierarchical database and one simple interface of `DeviceFlatContainer`. The class `DeviceFlatContainer` defines an appropriate iterator, a class that can sequentially access all the elements that the given object of a class `DeviceFlatContainer` aggregates. Note that the number of members of such a container is dynamic and it corresponds to the number of variants of the given hierarchical design. As it was pointed out in the first section of this chapter, the Virtually Flattened View, as well, can alter the hierarchical netlist by inserting new abstractions and rearranging the hierarchical order of the netlist. This includes altering the variant graph structure. An upcoming section will define the usage of the mentioned class pair (container – iterator).

So far elaborated classes in this conceptual hi-level diagram are following the interface of the standard NLDB database and mimic its behaviour.

Additional classes that are part of the general view architecture are `Virtual_HierContextSaver (Virtual_Excluder)` and `Virtual_ElementBuilder`. These parts of the system maintain the hierarchical context of the current Virtually Flattened View materialised flat data portion (MFDP) and control the virtual layer object building process, respectively. The `Virtual_HierContextSaver` defines the relative top hierarchical level for the current state of the Virtually Flat data portion. This level is dynamically chosen by a sophisticated algorithm applied on the specially devised data structure. Hence, for each MFDP that is created this class attempts to place it as deep in the hierarchy as possible. In this way we tie the flat data portion to the maximal number of different contexts. Hence, the MFDP is valid for each instantiation of the relative top level cell.

Two different strategies of the external usage of the class `Virtual_HierContextSaver` are possible. First, it can be used explicitly to get the set of paths for the given virtually flattened pattern. Second, the information about the materialised data portion position (relative to the hierarchical design) can be used implicitly, by altering the primary standard NLDB attributes of the design. More precisely, this happens by introducing new subcircuits and adding their instances to the original hierarchy. In order to implement the first approach, it is necessary to define an extension to the standard interface that is to be used by the user algorithm. In this case, the utility flat algorithm would have to be altered, at least in the phase in which it commits its results. This change would remain, however, local and the main part of the flat algorithm would stay the same. The second approach hides everything in the existing hierarchical database interface. In order to achieve the benefits of this approach it is necessary to allow altering of the primary hierarchy topology and therefore the variant graph of the given hierarchical design, as discussed in the previous section.

The class `Virtual_ElementBuilder` encapsulates the process of the creation of a materialized flat view. This part of the system offers a flexible and updatable architecture, allowing fast adaptations to the specific needs of different user algorithms. For instance if the user algorithm needs some additional interface/variables to be added to the devices of the NLDB design for its proper execution, one would add these functionalities to the relevant database elements by creating more specific type. By abstracting the element creation process into a clearly defined class we can easily setup our view to create these specific objects which it is still able to manage ignoring the specific interface, hence, leaving it to the user algorithm.

The described architecture, therefore, defines a holder for a group of objects that represent a locally flat data portion. The flat data portion materialization starts from the object that is returned from the `DeviceFlatContainer`. In continuation we will explain the implementation architectures, hence, specific data structures and algorithms that allow the approach described with this high-level view on the proposed design. The relation between the objects of (position of) the view schema that `Virtual_ElementBuilder` creates and its holder, together with the taxation hierarchy description of view classes and other parts of NLDB will be presented in the next section.

5.3 Virtually flattened view class representation

The view consists of the collection of classes that upgrade the functionality of NLDB. There are analogue classes for each of the NLDB (base) originals. For instance, the class `Base_MOS` has its view analogue class, `Virtual_MOS`. The relation between these two classes and their position in the overall NLDB class hierarchy is shown in the class diagram in Figure 5-3.2. As it is shown in the diagram, virtual layer classes are not directly inherited from the pure abstract interface of the Access Layer. The reason for this is that the layer requires also some implementation of the static database. After acquiring the starting element, virtually flattened view materializes a small portion of data from the hierarchical database in the flat fashion. All objects that are aggregated into this small view portion are the objects of different view classes. The implementations of interconnections between the database objects are therefore, directly taken over by the layer classes. Apart from being able to use the Virtual layer classes in place where some other base, or more general, Access Layer class is expected, we get the implementation of interconnections of the materialized pattern for free. Just by accessing the interface of the Base Layer inside the Virtual Layer classes, we can access the objects locally, those that are already loaded into materialized view. When the view is augmented, specific overridden interface functions combine the old implementation to e.g. acquire a pin of the device with the functionality that is implemented in overridden virtual layer interface methods. This upgrade reads data from the previous layer in cases where the object, member of the virtual layer still does not have any information about the appropriate connection. Note that the source layer object is defined as `Access_Device`, that means, combining several layers in order to get the corresponding variant of the hierarchical data representation is allowed. Figure 5.3-2 gives the inheritance hierarchy for the view, with its relation with other NLDB classes.

The class diagram shows intentionally the pure abstract class `Acess_Device` at the bottom. This is the root class and each of the classes that are deduced from it have to implement a strongly defined common interface. In this light, all classes of the Virtual layer implement this standard interface in their specialized way. In the first (diagonal) row the primary class hierarchy is shown. The `Access_Device` pure abstract class is specialized by the class `Access_MOS` in order to define augmented interface of the `Access_MOS`, still pure abstract class. Base Layer classes statically implement a hierarchical aggregation of database objects. They take care of implementing numerous references to capture the hierarchical netlist topology and additionally all necessary attributes about a single class that are available through the Standard Common Interface (SCI). In the Figure 5.3-2, the hierarchy of Base_<classes> is shown in the middle diagonal stripe.

The virtual layer classes are all realised by multiple inheritance. The root class of the Virtualy Flattened View hierarchy, the abstract class `Virtual_Device` multiply inherits properties from the `Base_Device`, in order to get the general functionality of creating a topology and the class `HasSource`. Class `HasSource`, as it is mentioned in the previous chapter enables layering. The sourceView of the `HasSource` class interpretes the SCI of the Access Layer. This is depicted by the association line from `HasSource` directly to `Access_Device`. Note that `Virtual_Device` privately inherits `HasSource`. This enables `Virtual_Device` to only privately have the interface of `HasSource` and that it, as well, disables the polymorphism between HasSource and Virtual_Device. Any hypothetical algorithm wouldn't be able to acquire a handle to Virtual_Device as the descendent of `HasSource`.

`Virtual_Device` is an abstract class. Its instantiation is not possible, as it has a set of undefined functions that are implemented in the child classes that again multiple inherit the `Virtual_Device`. This is necessary in order to be able to employ polymorphism and use the Virtual_MOS, Virtul_Res or some other class that is in place of the Base_MOS, Base_Res, etc. It is important to stress that, since this design was implemented in C++, each of multiple inheritance paths are, as well, marked virtual, in order to ensure a single instantiation of each of the parent classes in the object memory layout. For example, `Virtual_MOS` has a `Base_Device` as a parent class through two different inheritance paths. `Base_Device` is the second parent

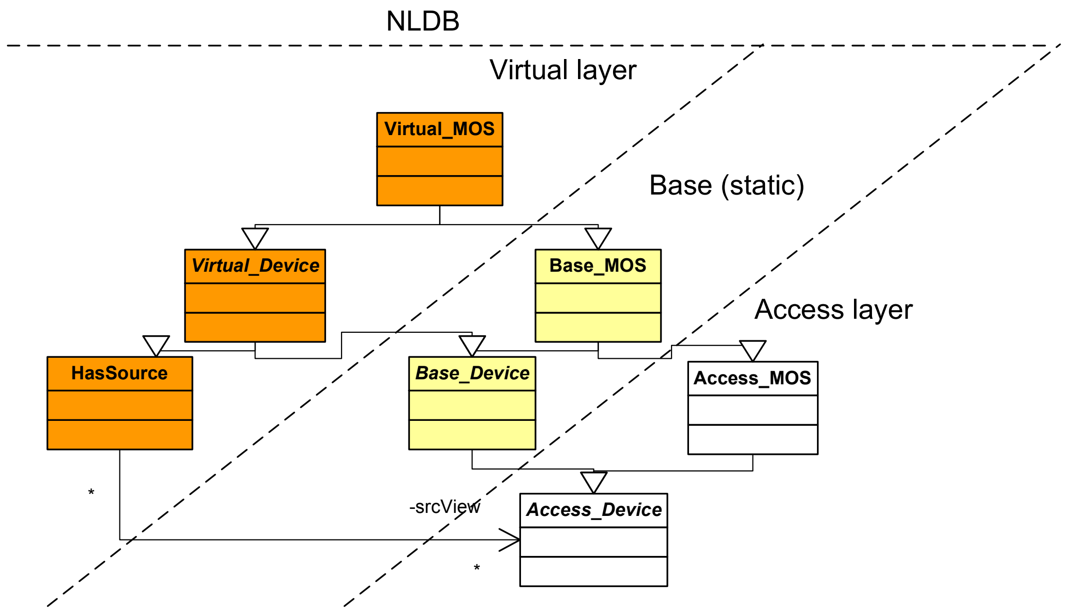


Figure 5.3-2 – Virtually Flattened View layer placement inside NLDB class hierarchy

class, both over `Virtual_Device` and over `Base_MOS`. This configuration is known as a “dreaded diamond” [47]. Its implementation demands pointer address mangling. This results in some overhead when comparing the pointers to the given objects, or accessing the object variables. This runtime penalty is paid in our case in order to achieve very elegant design that requires minimal changes of the static base classes and 100% transparent usage of Virtual Layer objects with the algorithm that was already written to use NLDB API.

In order to enable full functionality of the single virtual class as the part of the virtual layer, overriding of the SCI (see section 4.2) methods is necessary. SCI methods, for this purpose, can be divided in two groups. The first group contains the methods that only forward their calls to the source (Access Layer) object. All these methods directly get information about source object's attributes.

In our simple database that is the function to acquire device model (`model()`) , or a name (`name()`). In realistic EDA databases we would have also the methods to get different device parameters.

A second group of methods of the virtual class are the methods that are part of navigation interface. In our example design that is the method of the class `Virtual_Device, pin(int i)`. Overriding this part of the interface of the virtual class family enables proper view (augmenting) navigation inside the already created MFDP. We will now analyse the algorithm of the function `pin()` of the class `Virtual_MOS`. This function of the SCI, that belongs to any descendent of Access_Device, gets the handle to the device pin which further leads to a given node to which the device's terminal is connected. The function outline is given in Figure 5.3-1.

This function (together with its analogues in different classes of the Virtually Flattened View schema) is responsible for auto-creation of the MFDP. Once the function is called, it first attempts to find the immediate (local to the view) connection to another virtual object, member of the virtually flattened view. If this information is not yet available (the neighbouring object is being referred for the first time) the function will read the data from the previous layer, getting, for instance, the static base object - instantiation of `Base_Pin` object. The algorithm now creates or regains the handle to the virtual object, depending on the fact if the neighbouring object was, potentially, already created using some other path in the view topology. For instance, if we have focus on one device of the parallel connection of two transistors, it is possible to reach the neighbouring device following any of the terminals, gate, source or drain. Therefore, a lookup map is needed in order to know if some object was already used. This is the responsibility of the `Virtual_ContextSaver`. This complex object (aggregation of objects) keeps record on any mapping between the source devices and the view devices. This mechanism will be explained in detail in section 5.6.

The flow of both mentioned scenarios is given in Figure 5.3-2, under (a) and (b). Both of these scenarios require lookup into the hash table and possibly object creation, which makes this usage case the slowest operation in the view navigation. Still look up is the operation with the expected complexity O(1) as a hash map is used. Therefore, as the experiments confirm, no major time was spent on these lookups during the application algorithm execution. Third scenario (c) acquires the virtual object

```
Base_Pin* Virtual_MOS::specific_pin( int i )
{
  Base_Pin* ptr;
  if( NULL == (ptr = Base_MOS::pin(i )))
  {
    ptr =
    Virtual_Netlist::getBuilder()->
                     getVirtual(this->getSrcView()->pin(i), this);
    setTerm(i,ptr);
  }
  return ptr;
}
```

Figure 5.3-1 – specific_pin function code

directly, from view's local references. This is the fastest scenario and in the same time independent (local to the view). There is no direct reference to the source NLDB database in order to acquire the proper object of the view. The described concept reminds of the proxy design pattern [48, 49], where a group of objects serves as a surrogate to the originals.

The concept by which methods, members of the navigation interface, are overridden is given the class `Virtual_Device`. Analogue methods exist for `Virtual_Pin` and `Virtual_Node` classes. They are used to model the bipartite graph by which any electronic circuit can be described (without hierarchy). In our case `Virtual_Device`s belong to one group of vertices, `Virtual_Node`s to the second. `Virtual_Pins` simply model the connections between these two groups. All

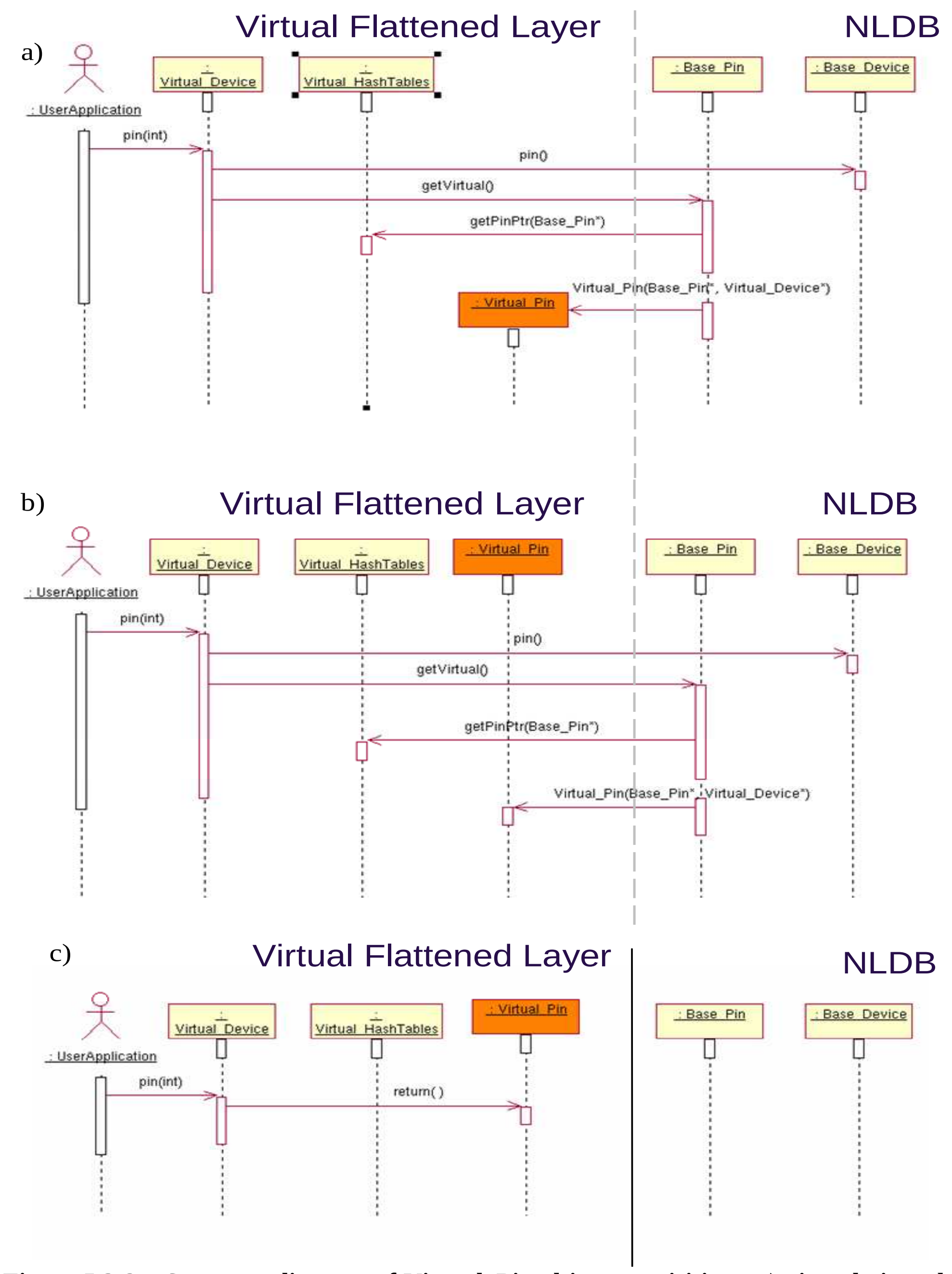


Figure 5.3-2 – Sequence diagram of Virtual_Pin object acquisition. a) virtual view object is created on demand. b) A handle to virtual view object is obtained. c) virtual layer object is directly acquired

tree classes have a similar implementation of the navigation interface.

In the end, it is important to mention that for the virtual class there is also third part of the interface. It is not part of the SCI and is privately defined, to the class. This interface is implementing layering (interface that each virtually flattened view object inherits from the HasSource class).

5.4 DeviceFlatContainer - Iterator

The `DeviceFlatContainer` class is defined in the scope of `Virtual_NominalCell`. It aggregates all devices in all representative contexts. If a given cell is instantiated in two equivalent contexts, regarding a set of parameters, its devices would be represented only once in this container. This is enabled via the personalisation concept given in Appendix A.

Following the container – iterator concept, a container defines an iterator class in its scope. The iterator sequentially acquires all elements that belong to `DeviceFlatContainer`.

In order to achieve this, the iterator has to traverse over all devices of all cells in the cell graph.

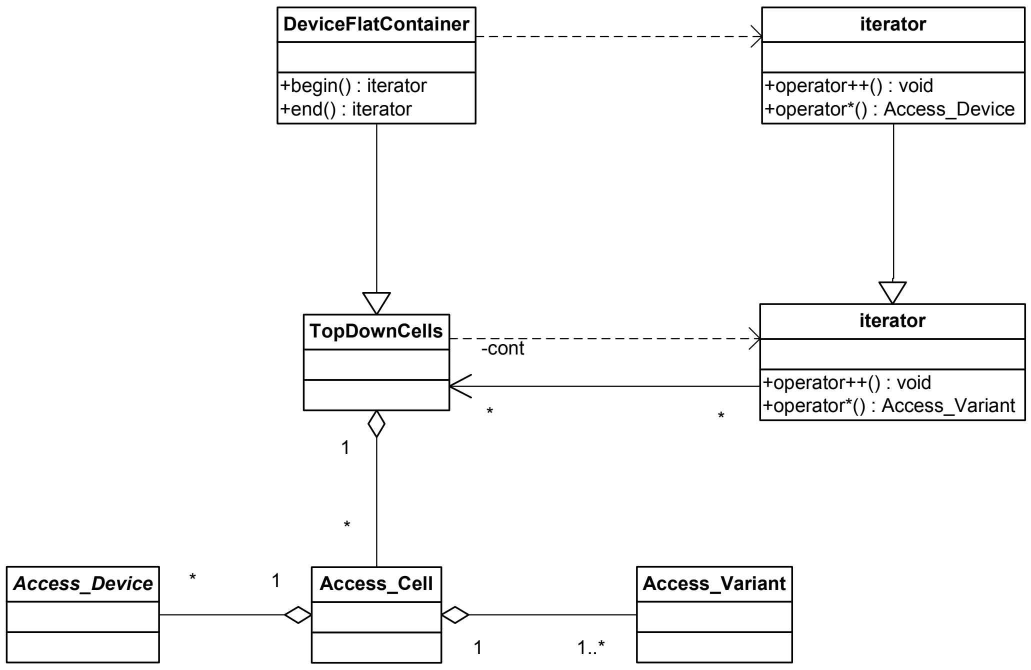


Figure 5.4-1 – Class Diagram of DeviceFlatContainer, facade for the aggregation of Acces_Device objects

Having in mind that the `DeviceFlatContainer` class stands for a set of objects with complex, hierarchical order and interface, groups them together, offering a simple interface (begin(), end() methods and the iterator class with the standard interface), we can notice that this part of the object oriented design follows the façade design pattern [48]. The implementation architecture of the class `DeviceFlatContainer` is shown in Figure 5.4-1.

The class is placed in the inheritance hierarchy of the class `TopDownVariants.` As it was explained in Appendix A, class `TopDownVariants` aggregates all hierarchical design cells in a specific order (top down or bottom up). These cells have some of the entristic parameters resolved (as chosen upon the variant creation). Together with this class, an iterator was defined, that can, sequentially, access all `Access_Cell` objects that are stored in the given design.

Class `DeviceFlatContainer` therefore subclasses `TopDownCells` and its appropriate iterator is subclassed by the iterator of the class `TopDownCells`.

The order of iteration of the `DeviceFlatContainer`'s iterator can be, up to a certain extent, controlled. The user can choose the order in which the design cells are accessed, bottom up or top down, depending on the setup of the `TopDownCells` class. Simple pseudo code to describe the traversal follows:

```
for (all cells)
  for(all variants)
    for(all devices);
        get pointer to the device;
```

Note that the sets that aggregate variants of the design cells and devices are not ordered and in this model, their order is arbitrary.

An additional, important property of the container that we define in this section is that its content is dynamic. If the user algorithm causes a change in the variant graph, e.g. by changing the type of a certain net or a device, or by altering any other parameter that is defined for the variant creation, the container would, as well, change its content. This can be illustrated with the simple example design shown in Figure 5.4-2.

The example design shows a NAND implementation of the XOR logic gate. If we suppose that the variants are being created by the cell pin type, and that all pins of

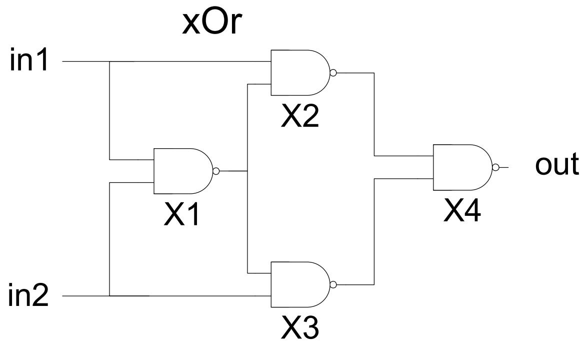


Figure 5.4-2 – Example of dynamic DeviceFlatContainer content. xOr hierarchical representation

the different instantiations of a NAND circuit have the same signal type, our design variant graph would have only two members: top level and one variant of the cell NAND. This would mean that the iterator of the device flat container, if it was setup to iterate bottom-up, would acquire focus on the only variant of the cell NAND, iterate over its devices, than further change the context of the variant to the top level. As in top level, there are no opaque (atomic) devices the iteration comes to an end. If we, for instance, during the user algorithm execution alter this hierarchical design and change the type of the net in_1, the revision of the variant graph would start and instances X_1 and X_2 would be moved to a different variant, as their input terminal one now has a different pin type. After this process our variant graph, apart from the top level variant has an additional variant of the cell NAND and in total two variants of this only cell that is instantiated in the given hierarchical design. The content of the `DeviceFlatContainer` has now changed and we get eight atomic devices during traversal, meaning four from the first variant of the cell NAND and four from the second variant of the cell NAND.

The implementation of the iterator class that enables dynamic traversing is closely related to the way new variants are added during the lifetime of the `Virtual_Netlist` object. For this reason a more detailed implementation of the iterator class will be given later.

5.5 Virtual element builder

In order to materialize the flat data portion, duplicates for each element acquired from the NLDB database are being built. These objects are to serve in different applications. Sometimes, according to the principles of the user algorithm, additional variables (fields) should be added to flat view objects that stand for pure NLDB objects. This can be achieved by subclassing given objects, augmenting their interfaces as needed and adding extra implementation variables. In order to enable this, the view has to support a flexible object building. For this reason, we abstract the building process in a class `Virtual_ElementBuilder`. The definition of a `Virtual_Netlist` holds a handle to the object of this class. In this way we separate the view object building from the rest of the system, enabling better flexibility. The solution that is engineered for the Virtually Flattened View follows the architecture shown in the class diagram Figure 5.5-1.

The `Virtual_ElementBuilder` is given as a combination between the Builder Pattern and the Template Pattern [48]. Thus the product (`Virtual_Device` descendents, `Virtual_Pin` and `Virtual_Node`) building process whose flow is managed by the director object (in our case Virtual_`Netlist`) is delegated to a special builder object (`Virtual_ElementBuilder`). On the other hand, a list of services is declared as a pure virtual interface and further used in the implementation of different higher level algorithms, which is a property of the template design pattern.

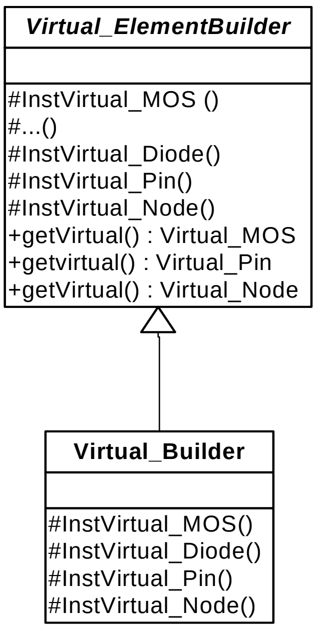


Figure 5.5-1 – Virtual_ElementBuilder architecture

Therefore, `Virtual_ElementBuilder` is the abstract class, as it just offers a family of pure virtual functions that encapsulate object instantiation. These functions form a protected pure abstract interface, as one logical part of the complete `Virtual_ElementBuilder's` interface. These functions are (in our case study model):

- `InstVirtual_MOS(),`
- `InstVirtual_Res() and`
- `InstVirtual_Cap().`

Another part, public interface of this abstract class, is implemented. The implementation of these functions, following the template pattern, uses services that belong to the protected pure abstract interface. All the functions of the public interface of the class `Virtual_ElementBuilder` have the same name, `getVirtual()`. They, however, differ by the argument type. For each type of the object a different function is implemented. The function architecture is standardized, following the pattern shown in the Figure 5.5-2.

```
Virtual_Device* getVirtual(Access_MOS* ptr)
{
  Virtual_Device* vir_mos;
  if( (vir_mos = currentContextSaver()->getElementPtr(ptr)) == NULL)
  {
   vir_mos = InstVirtual_MOS(ptr);
   currentContextSaver()->putElementPtr(ptr, vir_mos);
  }
 return vir_mos;
}
```

Figure 5.5-2 – example method of the getVirtual() family

The example shows that the implementation of the method to acquire the virtual copy of the `Access_MOS` object at first looks up if the appropriate object is already instantiated and if so, it acquires a pointer to it. This is done by looking up the table of existing virtual copies of the database elements at the given hierarchical level. Next chapter explains the data-structures (`Virtual_ContextSaver`) that save these mappings. If no mapping has been found, the `getVirtual()` method would instantiate a new object using an appropriate method from the protected interface. In our example, the method which is called is `InstVirtual_MOS()`.

As an exception to this group of methods, the method `getVirtual(Access_Node*)` has a somehow more complex implementation. The reason for this is the fact that the virtual nodes are distributed over the hierarchy. The explanation of the recursive algorithm of the mentioned method is left for section 5.6, after defining proper environment which helps its understanding.

Note that, as the class `Virtual_ElementBuilder` is abstract, it can not be created. Therefore, we define a class which inherits this abstract class and implements the promised interface. The class `Virtual_Builder` is the default implementation of the builder and its realization of the protected interface simply instantiates pure NLDB objects, members of Virtual Layer. For instance:

```
Virtual_MOS* InstVirtual_MOS(Base_MOS* ptr)
             {return new Virtual_MOS(ptr);}.
```

This function simply wraps the instantiation of the `Virtual_MOS` class.

According to the selected architecture, a `Virtual_Netlist` object has a reference to a single object of the element builder and it gets the handle to a concrete object that subclasses `Virtual_ElementBuilder` already through its constructor. Additionally, access to the builder object and further to its interface that builds the relevant products is defined through a public getBuilder() method of the `Virtual_Netlist` class. As it was chosen, during the lifetime of the `Virtual_Netlist` object, it's not possible to change its builder. In this way we can assure the consistency of the objects which are being built. Note that with a small interface change of the `Virtual_Netlist`, this can be however altered and if it would be necessary for some future use, builder objects can be exchanged during different phases of the user algorithm. In this case, the user algorithm, during its runtime would have to use this additional interface and mange building process consistently.

5.6 Context saving tree

The context saving tree is used to assure the consistency between the materialised flattened data portion and the hierarchical database. It defines all proper mappings between the devices that are represented as the flattened data portion. Additionally, it defines the position of the view in the design hierarchy, relative to appropriate variant that is considered as the context for the materialised flat data portion.

The context saving tree is important also in the process of committing the relevant flat data portion to the primary hierarchical topology of the design. We want to use the very same objects that were tracking the mapping between virtual and source objects thus assuring the consistency of the flat view with the hierarchical database during the process of creation and maintenance the VFV. This time we assign them another semantic role: providing information about hierarchy changes after the committing step where the relevant state of the VFV (current MFDP) is embossed to the primary hierarchy of the input design.

As we have identified two roles of this single object that stores mappings between source and virtual objects, we need the specific interfaces for both usage cases, as well. This makes the overall interface of the given class bloated. Additionally, using the same class to depict two semantically different entities is making the perception of the architecture of the given object oriented solution less understandable. For that reason, to model this part of our system we refer to the concept of Objects with Roles.

5.6.1 Objects with roles

Object oriented concepts tie objects to their types statically. No dynamic type, i.e. morphing of the given object from one type to another, during its lifetime is allowed. The only type changes that are allowed automatically are those along the inheritance hierarchy. This is actually a relation of the more general type to more specific type. On the other hand, it is not exceptional that in different applications, the same object plays more than one role during its lifetime. In each of these roles, the semantic character of the object varies, depending on the context in which it was used. This has sparked a discussion in the object-oriented software development community and various solutions have emerged. Some of them propose new concepts in general object oriented methodology. They are implemented in experimental languages, or just theoretically discussed [50]. Others search for the solution using already available standard mechanisms, creating specific design patterns in order to solve the mentioned problem. Fowler describes a set of approaches in order to solve the role problem and points out their advantages and disadvantages [51]. On the other hand, Bäumer offered a design pattern in which he handles the object roles by instantiating a separate object for each role - Object Role Pattern [52]. By delegation, the core object (which stores relevant information) is accessed from different role objects (that belong to different classes with clearly defined interfaces). This solution offers flexibility and precise definition of separate role entities and interfaces, but suffers from increased complexity of the interface of the object (role maintenance interface) and overhead to implement manipulation, creation/destruction of new or no more active roles. Further, the object identifier consistency is violated. Hence, you can not trivially compare two appearances of the very same objects in two different roles.

We will use a solution that is similar to this one, but is realised, through the object-oriented concepts available in C++: multiple inheritance, friend relation and other standard mechanisms, making it much simpler to use and maintain. The proposed architecture is given in Figure 5.5-3. The diagram describes three abstract classes and one concrete that is possible to be instantiated. At the bottom of the diagram is the pure virtual class `PureAbstractServiceProvider`. It defines a set of protected services, but leaves its implementation undone. The services are equivalent to the services that are defined in order to handle the states of the `Implementer` class object. Both interfaces are defined protected. Thus, the specification of this interface is visible only for the classes that are in the inheritance hierarchy of `PureAbstractServiceProvider`. Those still undefined services are used to implement the public interface of the object with roles, classes `InterfaceA`, …, `InterfaceN.` We can compare this part of the design to the Template Pattern[48]. Each of the abstract classes define their generally different interfaces and expose the declaration of the method `getPointer()`, which enables object passing between different roles. In the end, the class `Implementer` multiply inherits all interface classes and implements (as protected) all undefined interface members. In this way, if the instantiation of the object Implementer is passed to the pointer of any of the role classes, its variables are to be interpreted using a completely different interfaces. Note that this exchange is possible during the lifetime of a single object, therefore allowing the interface methods of different role types to work on the same data in two different contexts. Additionally, implementer class is opaque. It has no public interface defined, which protects the data of this object from misuse. This is achieved by “hiding” the public interfaces of the classes InterfaceA,… ,InterfaceN by making them private, employing the C++ `using` keyword. Further, interface classes (`InterfaceA`, …, `InterfaceN`) privately inherit `PureAbstractServiceProvider.` Thus, polymorphism between `PureAbstractServiceProvider` and any role interface is switched off.

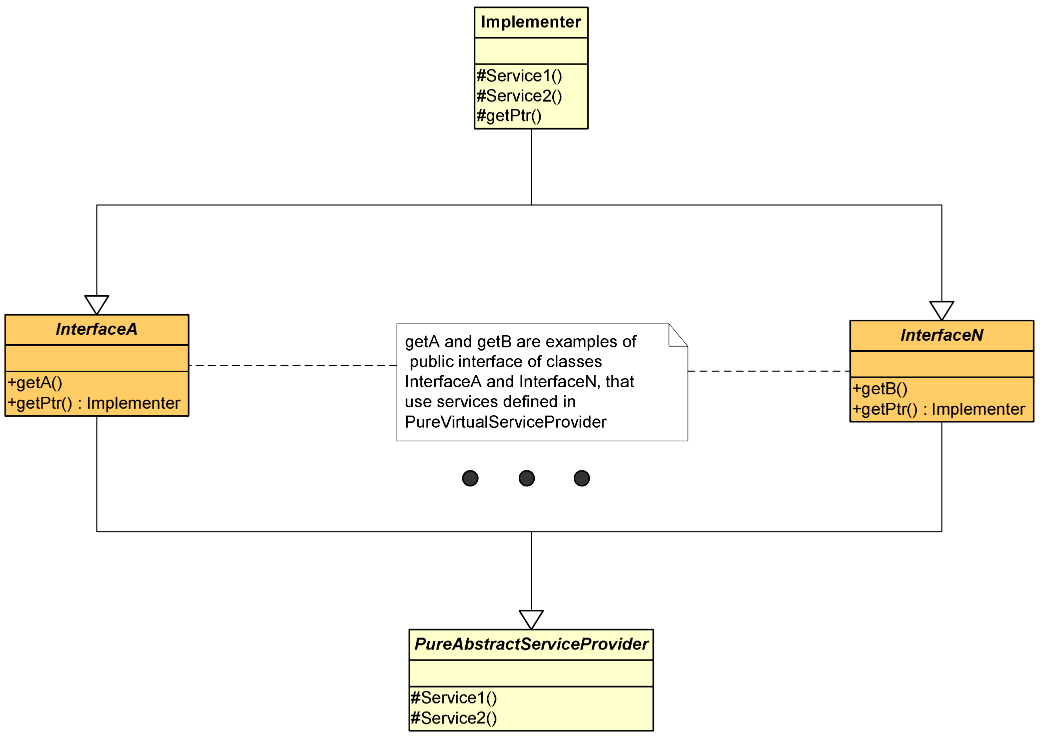

Implementer
#Service1()
#Service2()
#getPtr()
InterfaceA
+getA()
+getPtr() : Implementer
getA and getB are examples of public interface of classes InterfaceA and InterfaceN, that use services defined in PureVirtualServiceProvider
InterfaceN
+getB()
+getPtr() : Implementer
PureAbstractServiceProvider
#Service1()
#Service2()

Figure 5.5-3 - Object with roles – design pattern proposal

By the proposed design pattern we achieve a clear separation of public interfaces for any object that is to be used in different contexts during its lifetime. Moreover, by having an additional type for each of the roles that an object plays in the design, we gain a better understanding and clear applications of the given object data. Further, the implementation of the object itself is exchangeable, as long as it realises the promised services. This gives additional flexibility to our design solution. The described design is employed in order to address the complexity of the implementation architecture of the VFV and make our documenting process of the algorithm more comprehensive, as well.

Therefore, two roles of the context saving objects are going to be defined. In the first role, the objects support the creation and consistency of the materialised flat data portion, maintaining the mapping between source elements and their virtual copies. In the second role, the same object is used in order to change the topology of the primary hierarchy. Therefore, the root hierarchy class, `Virtual_HashServices` defines the protected interface that maintains the state of the multi-role object:

```
 virtual Virtual_Node* getNodePtr(Access_Node* bas) = 0;
 virtual Virtual_Pin* getPinPtr(Access_Pin* bas) = 0;
 virtual Virtual_Device* getElementPtr( Access_Device* bas) = 0;
 virtual void putNodePtr(Access_Node* bas, Virtual_Node* vir) = 0;
 virtual void putPinPtr(Access_Pin* bas, Virtual_Pin* vir) = 0;
 virtual void putElementPtr(Access_Device* bas, Virtual_Device* vir) =
0;
```

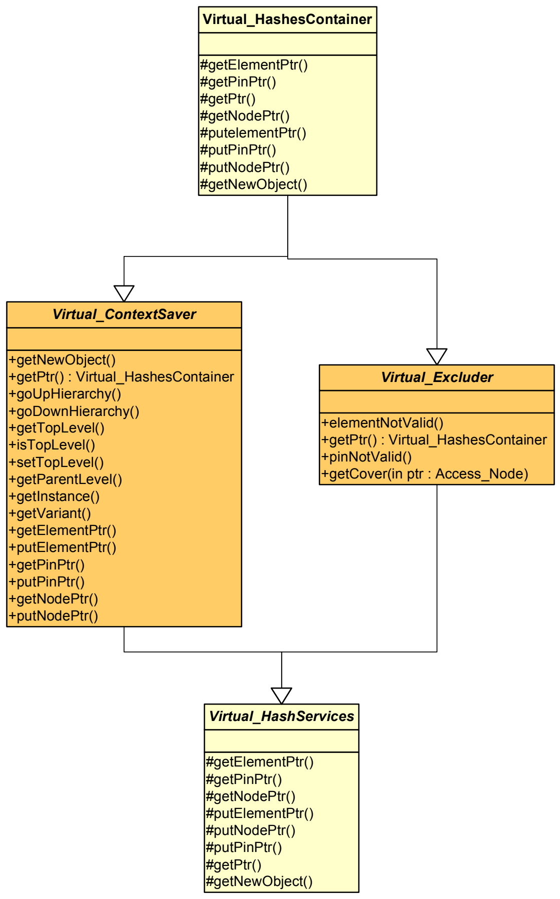


Figure 5.6-5.5-4 – Virtual_ContextSaver and Virtual_Excluder classes

These functions define the processes of assigning and retrieving relevant mappings between the objects of the MFDP and their sources. Note that all the interface methods are pure virtual, any implementation issue is left for later. We are concentrated only on the interface, not on any performance or complexity matter in this moment.

Additional to these application domain methods, two methods to support role switching and concrete object instantiation are defined:

```
Virtual_HashesContainer* getPtr() = 0 ;
Virtual_HashesContainer* getNewInstance() = 0 ;
```

, also as pure virtual.

For our application we need two roles: the first is modeled by the abstract class `Virtual_ContextSaver`, and the second, with the abstract class `Virtual_Excluder`. Their interfaces and semantics are going to be explained in detail in upcoming chapters.

As the implementer class, we have `Virtual_HashesContainer.` This class realises all promised interfaces and hides the public interfaces of the separate role classes. Thus, it has no public interface any more, leaving the object opaque as it was recommended by the proposed design pattern architecture. One can compare this object to a cassette (or a disc) and the roles to the relevant devices that read it.

The implementation of `Virtual_HashesContainer` consists of a set of hash tables. These tables should provide the constant expected complexity for frequent lookups, which are performed by both role public interfaces during the proper algorithm execution.

The methods to retrieve and set mappings are referring than to the hash_tables of the `Virtual_HashesContainer` and the method `getPtr()` implementation simply returns the pointer to `this`, while the method `getNewInstnace()` accepts the appropriate parameters and invokes the privately defined constructor of the implementer object.

By inclusion of Virtual_HashesContainer and realising all proper promised interfaces the architecture we have given above is ready for both contexts of usage.

5.6.2 Consistency of the virtually flat view data portion objects with NLDB database (Virtual_ContextSaver)

As we have stated before in this chapter, the VFV takes an arbitrary device, returned by the `DeviceFlatContainer` iterator as the starting point for generating a flat data portion, arbitrarily according to the needs of the application that navigates in the neighbourhood of the starting device. For each of the acquired original database elements (including devices, pins and nodes), which are distributed over the hierarchy, a virtual copy is created. The virtual copies form together a flat view on the local part of the hierarchical data. It is necessary to maintain the consistency between these materialised objects (members of the given MFDP) and source (original) objects. The consistency between the MFDP objects and its source objects that belong to the hierarchical database is modelled through a class `Virtual_ContextSaver`, more precisely as a complex structure (a tree) of objects of this class. The tree structure is needed in order to be able to properly grasp all mappings between the hierarchically distributed source database elements and the MFDP, allowing it to develop freely crossing hierarchical borders.

The context saving tree is dynamically created and manipulated by the virtual objects that build the MFDP. That enables the MFDP to be self-augmenting hiding all the complex operations concerning consistency maintenance from the user and performing them internally by the VFV.

Every context saving tree starts from the unique `Virtual_ContextSaver` object that defines the context of the key device, which is created upon invoking the star operator of the `DeviceFlatContainer` iterator. The context saver object is tied to a given variant of the cell of the hierarchical model.

If the algorithm tends to develop the MFDP and accesses the neighbours of the virtual copy and if that neighbours are distributed over the hierarchy the context saving tree grows accordingly inserting the relevant context saving object and for each affected hierarchical level and populating it with the relevant mappings. We will define two important concepts of the context saving tree:

- the *active hierarchical level* and
- the *relative top hierarchical level*.

As each context saving object stores mappings between the relevant source objects, that belong to certain hierarchical level and the MFDP objects, it is necessary to choose the proper context saving object to which we store mappings, or from which we acquire mappings. Thus, we always mark an active level that defines the current position of the hierarchy that is in focus. Keeping the active hierarchical level in consistency with the relevant lookups is crucial. The relative top hierarchical level is the hierarchical level to which the root context saving object of the context saving tree is tied. This level is important as it determines to which context of the hierarchical database the overall MDFP belongs.

In order to illustrate this concept, we can consider the example in Figure 5.5-5. The example design is a hierarchical representation of a latch electronic circuit. We show the hierarchy fully unfolded. NMOS and PMOS transistors are encapsulated in separate subcircuits. They form an inverter circuit inside the cell A. Further, on the top level, two instances of the identical cell A are properly interconnected to form the topology of the latch electronic circuit. In the initial stage (a), the algorithm takes in

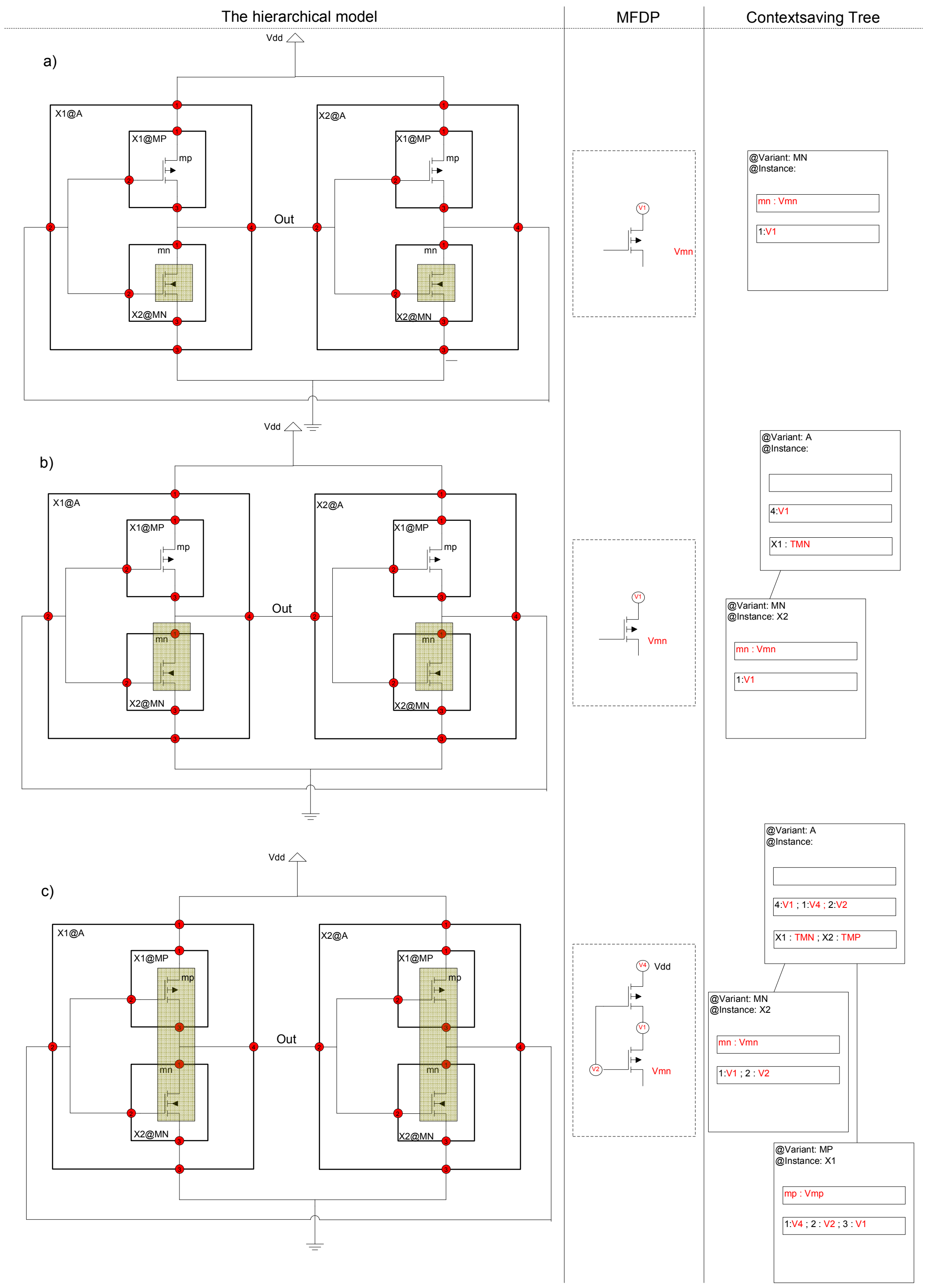


Figure 5.5-5 – The example of the development of the context saving tree, the structure that ensures the consistency between the MFDP and the hierarchical data model.

focus the device mn of the cell MN, returned by the iterator. In this moment, the materialised view consists of the sole instance of the object of a class `Virtual_MOS` (Vmn) and after that, for example, after calling the pin() method of the given device, additionally, `Virtual_Node` object (v1) is instantiated. Note that the implementation of the algorithm uses the specific strategy to instantiate the virtual objects, the members of MFDP, as late as possible, upon direct need for the given object by the user algorithm.

The objects that build the current virtually flattened data portion have their source levels set as objects mn (Access_MOS) and 1 (Access_Node), respectively. These both objects belong to the variant of the cell MN. To grasp these relations an object of the context saving class `Virtual_ContextSaver` is instantiated. This object ties the MFDP to the suitable variant of the cell MN and additionally stores all proper mappings between virtual objects and source objects. The relative top level of the context saving tree is in this moment the only instantiated object, naturally, so is the active hierarchical level. The relative top level defines the position of the MFDP in the hierarchical database. Thus, the MFDP is valid for all instantiations of the cell MN. In relation to that, the relevant context saver object (of the relative top hierarchical level) has its field “@instance” empty. As an illustration, the yellow patch is sketched in all proper places of our example hierarchical design for which the MFDP is valid (inside every instance of the variant of the cell MN).

Let us consider now that the user algorithm navigates away from the starting device (virtual object, Vmn) following the drain terminal and further the node V1. As the source node of the virtual node V1 is the port node 1, of the cell MN, the connection with the levels higher in the hierarchy of the design description exists. In the first row, the immediate parent level is the definition of the cell A. This implies the change of the topology of the context saving tree. The level, which ties the view to the cell MN, becomes the leaf object of the tree, while the root level switches to the newly instantiated context saving object, that is tied to the context of the cell A. The level switch process includes the insertion of the proper mappings in the context switching objects. Thus, the context saving object that is tied to the variant of the cell MN gets the parent object (new context saving object) and the reference to the instance X2 of the cell MN that exists in the hierarchical level A. This context saving object represents now exclusively the instance X2 of the cell MN. The newly instantiated relative top context saving object is initialised with the mapping that links the instance X1 to the context saving object of the level MN and the mapping between the already instantiated virtual node V1 to the source node 4 (of the hierarchical level A). The stage of the Virtually Flattened View and the context saving tree after this step is depicted in (b). It is interesting that the change of the relative top hierarchical level and the augmentation of the context saving tree was done while the actual appearance of the MFDP is still unchanged. We have, by this operation switched the (active) hierarchical level in which the MFDP exists and allowed it to “see” its neighbouring objects in the context of the hierarchical level of the cell A.

Let’s consider now that the user algorithm, seeing the flat version of the design tries to acquire the pointer to the device pin that is connected to the virtual node V1. In the original, hierarchical database, the device pin connection exists in the hierarchical level of the cell MP and connects the given node to the device mp. What does this mean for the context saving tree? VFV will determine that the source node of the V1 is connected to the instance X1 of the cell MP. It will than follow this connection and descend to the hierarchical level of MP. This means that a new context saving object will be created and inserted in the context saving tree. This context saving object is tied to the cell MP and the instance X1. Once the algorithm descends it will create the linking mapping of the virtual node V1 to the local (to the cell MP) source node 1. At this moment the virtual node V1 has three sources, each of three relevant for separate hierarchical levels MN, MP and A. The algorithm further acquires the pin connected to the node 3 and creates the relevant virtual copy of it attaching it to the virtual node V1. Following this pin the MFDP obtains another device, the virtual copy of the device mp, named Vmp. If the algorithm further follows the gate connection of the virtual device Vmp, in the background will the following happened: At first the VFV algorithm will, inside the hierarchical level MP create the virtual copy of the node 2, called V2. Than, the active level of the context saving tree will switch to A. All port node mappings will be propagated to the level A of the hierarchy. This means that the mapping between the node 2 and V2 will be made at the hierarchical level A. After this, the VFV algorithm again switches back to the hierarchical level MN, without port node propagation. It takes in focus the source node 2 and searches for its virtual copy using the function getVirtual(). The implementation of this function is always searching for the relevant mappings recursively from the root node of the source hierarchical node to the current subnode. In the example case, the algorithm will search for the mapping between the node 2 and some virtual node in the level A, find it and than build the mapping between the node 2 and the same node V2 inside the cell MN. The recursive algorithm that we describe here is given in figure Figure 5.5-6.

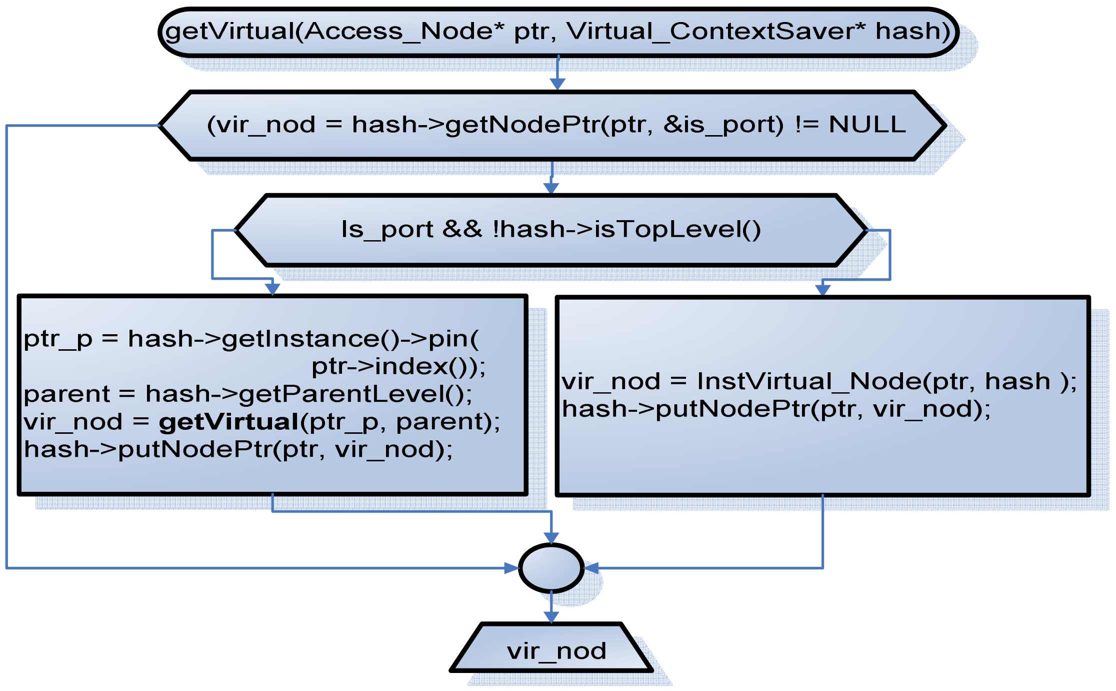
The stated requirements shape the functionality and the interface of every individual context saving object. First, the object is tied to a specific variant of the cell of the given design. If the context saving object is not the root of the context saving tree, it is additionally tied to the given instance. This is modelled through the part of the interface of the `Virtual_ContextSaver`, by methods:

- getVariant()
- setVariant()
- setInstance()
- getInstnace()

As the given context saver object is a member of the complex structure (a tree) formed by the object of that kind there is a specific interface to navigate through and augment this tree:

- goUpHierrarchy()
- goDownHierarhcy()
- getTopLevel()
- isTopLevel()
- setTopLevel()
- getParentLevel()

The first two methods of this group are capable of augmenting the context saving tree (creating new tree nodes). According to the appropriate parameters they instantiate the new context saving objects tying them to the appropriate hierarchical levels and placing them to the appropriate positions in the context saving tree. Apart from creating or switching levels, the function `goUpHierarchy()` is responsible for propagating the mappings for all port connections up the hierarchy. This is important preparation for the algorithm which determines the virtual copy of a source node in the given moment (getVirtual(Access_Node*)), that is a member of the `Virtual_ElementBuilder` class. The rest of the methods can exclusively navigate the already created context saving tree. Both objects groups are used by the VFV in order to properly maintain the consistency between MFDP and the data model. The methods to switch levels are implemented directly in the class `Vir-`



getVirtual(Access_Node* ptr, Virtual_ContextSaver* hash)
(vir_nod = hash->getNodePtr(ptr, &is_port) != NULL
Is_port && !hash->isTopLevel()
ptr_p = hash->getInstance()->pin(ptr->index()); parent = hash->getParentLevel(); vir_nod = getVirtual(ptr_p, parent); hash->putNodePtr(ptr, vir_nod);
vir_nod = InstVirtual_Node(ptr, hash); hash->putNodePtr(ptr, vir_nod);
vir_nod

Figure 5.5-6 – algorithm of the function getVirtual(Access_Node* ptr)

`tual_ContextSaver`, as this is the specific functionality of the role of context saving played by the object of the class `Virtual_HashesContainer`.

Another important group of methods is responsible for storing and retrieving the mappings between the elements of the MFDP and their source objects. These functions take a pointer of the `Access_Device`, as a key, and search for its virtual copy in the appropriate (current active) context saver object. Note that consistency between level switching and device mapping search is here essential. A public interface to retrieve and store mappings from the hash tables is implemented using the template design pattern, as it is already mentioned in the previous section. The services declared in the `Virtual_ServiceProvider` are here used to define the interface and implemented later while defining `Virtual_HashesContainer` class. This flexible architecture offers, apart from level switching ability, also easy experimenting with different types of mapping container implementations. To conclude the interface that stores and retrieves the mappings consists of the following functions:

- getElementPtr(),
- getPinPtr(),
- getNodePtr(),
- putElementPtr(),
- putPinPtr() and
- putNodePtr(),

that are more or less trivially publicly exposing the functionality of protected services of the `Virtual_ServiceProvider`.

In the end, this object role exposes the proper interface to support the object switching and the context saver object instantiation. Thus, two additional methods are implemented:

- getNewCSObject()
- getPtr().

The part of the algorithm that constructs and maintains the context saving tree is not visible to the user algorithm and is hidden behind the virtual node class. Furthermore, the virtual copy of any `Access_Node` object, instance of the `Virtual_Node` class becomes the context-switching object. This will be explained in the following section.

5.7 Context-switching / multi-context nodes

In Chapter 3, we have defined the hierarchical node and the three semantically different types of subnodes that are forming the hierarchical node. The hierarchical node was presented as the consequence of the instance tree. If we have the instance tree in focus, these hierarchical nodes have just one context.

On the other hand, if we observe the referenced cells graph (variant graph), hierarchical nodes become multi-context, as any of the cells that hosts the parts of the hierarchical node has the multiple instantiation paths.

Virtually Flattened View hides the hierarchical node (composed of an arbitrary number of elementary Access_Nodes, depending on the hierarchy) behind a single element – `Virtual_Node`. This node is responsible for context switching. In connection to that, it also controls the creation and navigation through the context saving tree.

Virtual node

As settled above, the virtual node is used to replace a group of nodes, connected through the hierarchy with a single node. This node is a part of the materialised flat data portion of the given circuit. It is modelled as a class that inherits the class `Base_Node`, which describes general properties of a node, Figure 5.7-1. `Virtual_Node` class does not inherit directly from the `Access_Node` abstract interface, as it uses different implementation solutions for the standard interface of the `Base_Node`, as it will be shown in detail later.

`Virtual_Node`, as well as `Base_Node`, can be observed as a container of pins, that connect this class of bipartite graph vertices - *nodes* to the other class which consists of devices. Therefore, pin_begin(), pin_end() operations and appropriate iterator class are defined, following the container/iterator concept [53], as discussed in chapter 3. The nature of the iteration in the case of the `Base_Node` is static and the iteration algorithm is simple. Hence, it is only necessary to traverse the vector that statically aggregates the elements of the type `Base_Pin`. Since `Virtual_Node` represents a group of `Base_Node`s (members of the hierarchical node), traversing the container gets more complicated. Iteration is also not single-context. Several, partially overlapping, sets of neighbours of the given `Virtual_Node`, occur.

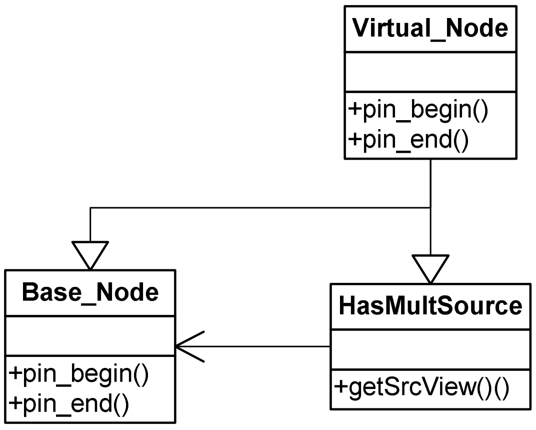


Figure 5.7-1 - Relation between Virtual_Node class and Base_Node class.

As we have mentioned, the virtual node is an object that belongs to the MFDP on the hierarchical data, therefore there must be a *srcView* defined to it. Through the source view the virtual node collects different parameters of the `Access_Node` interface that are invariant to the hierarchy (for instance those are the node type data, etc). Moreover, the virtual node has a relation 1:n to the `Access_Node`, as shown in Figure 5.7-1. The layering functionality for the `Virtual_Node` is modelled by the specific class `HasMultSource`. This class is inherited from the class `HasSource`, adding multiple source property. A single virtual node has in general more than one source view. The current source view is defined by the current hierarchical level that is in focus (current `Virtual_ContextSaver` object, part of the context saving tree), during `Virtual_Node` "container" traversal. A private interface that models this is the `getSrcView()` function, that takes a pointer to the object of the `Virtual_HashTables` class as an argument. The mapping is implemented as a hash table, therefore the average (expected) complexity of fetching the data has O(1) [54]. The operation of storing the data in this hash table has a worst-case linear dependency, but with a wisely chosen hashing function, this case is unlikely to occur in praxis.

The virtual node should feel and appear like an integral node, member of the flattened data portion. In this light, we have to define an iterator for this multi-source virtual object, as well.

The iterator has to traverse all possible neighbours of the given node. The order of iteration can be partially determined by the hierarchy. Hence, the members of several unordered sets can be presented in the order that is adjustable. For instance, it is natural first to iterate over the pins (connections) of the first local node, then to traverse down the hierarchy and then to step up the hierarchy. Traversing is similar to the depth first search, which does not start from a root level.

The order of visiting the parts of the graph, which is formed by the hierarchy of nodes, will be first explained using the example shown on the Figure 5.7-2.

In the figure, the ports are marked with red colour, while the root nodes are in orange. Cell borders are shown only partially, with dark angular lines. The iteration starts with the node 1, the pins connected locally to this node would be accessed in arbitrary order, more precisely, by the order defined by the insertion in the vector of pins. This further depends, e.g., on the implementation of the SPICE netlist parser, if the hierarchical design has been loaded from the netlist external ASCII format. After traversing all the ports, the iteration is exploiting all the choices given in the current (top) context (cell A). That means that the iteration continues with the pins connected locally to the node 2, inside cell B, which is down the hierarchy in comparison to the starting node 1. At that moment, the hierarchical level that is in focus will change and another entry will be added to the context saving tree. Hence that once the hierarchical level that is in focus has changed, the active source view of the `Virtual_Node` switches to the node 2. Note that any switch of the hierarchical level is followed by change in the active level of the context saving tree. In our example case the new object will be inserted into the tree, leaving the context saving object that corresponds the level A as the relative top for the MFDP and setting the active context tree level to the relevant instance of the cell B. Having in mind this process that happens in parallel, we will further concentrate only on the states of the multi-context node.

The next step is the iteration over the neighbours of the node 3. Once this is finished, the context from which the iteration has started is completely analysed. We remind the reader here that all the pins, acquired through this process are actually the source objects for the relevant MFDP copies achieving our goal that the data portion is presented as flat to the user algorithm, which initiates this hypothetical iteration over the neighbours of the virtual node. The iterator can be set to stop the iteration here, after traversing a single context. This is important for certain applications and is completely similar to the flat circuit iteration.

Let's now consider that the set N_e=N_a is the set of all possible neighbours that are traversed so far, the set of neighbours of the context of the cell A. The set N_e defines all neighbours, in a given moment, for the multi-context node. As it can be seen in our example figure, node 1 is a port node that is connected to two nodes up the hierarchy, but sitting in two disjunctive contexts. Therefore we distinguish two disjunctive sets of neighbours that will be added to the original set N_a. $N_e = N_a + N_1$, or $N_e = N_a + N_2$. Therefore if we proceed to node 4, the iteration is performed on its local nodes, and than to all subnodes, in the lower hierarchical levels. It is important to exclude the path that leads to the instance of cell A, back to the same node where we have started our iteration,

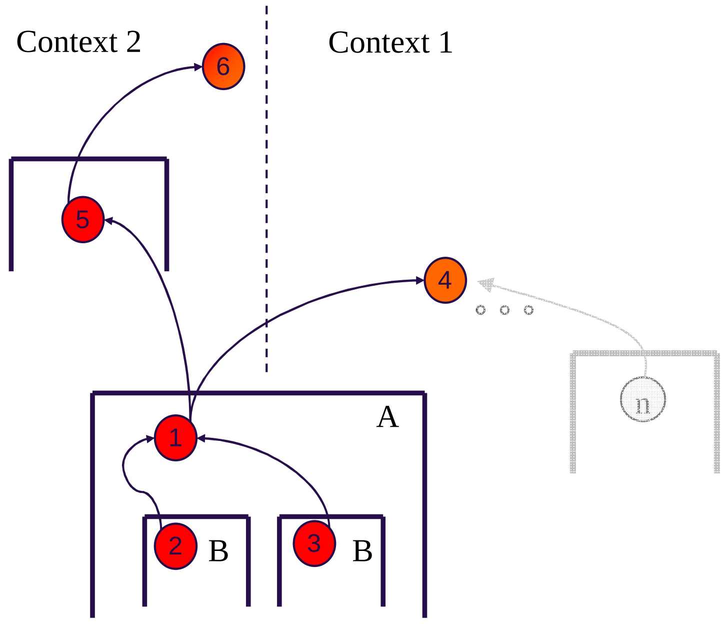


Figure 5.7-2 - Example of a multi-context node. Subnodes, which are ports, are given in red, while root nodes are represented with orange circles. Design hierarchy is given by unclosed angular lines.

in order to avoid an (semantically incorrect) endless loop! The process would recursively repeat up the hierarchy as long as there are ports in the topology of the given

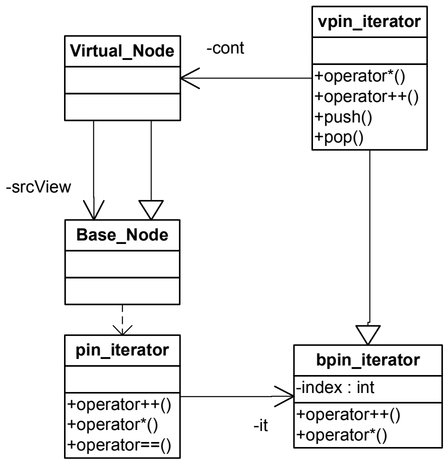


Figure 5.7-3 – Positioning of the vpin_iterator class in the CSI. Polymorphism allowing architecture, where the implementation of the iterator is chosen during the runtime.

hierarchical node. Note that it is possible to have new context "crossroads" further. The set of neighbours would be further augmented. Once all the neighbours of the newly defined context are traversed, in our case the pins of the node 4, the algorithm returns to the previous context, erasing all the invalidated neighbours from the view, the neighbours that belong to the N1 set. This is being done by cutting the top part of the context saving tree which corresponds to the context 1 and all other (eventual) subcells that were part of the iteration, leaving the subtree rooted at hierarchical level A. After this, the algorithm proceeds further to nodes 5 and 6, where the traversing operation for this example finishes.

We have therefore introduced the multi-context node, a switch through different hierarchical contexts and, implicitly, a multi-context MFDP. The latter will be analysed in the following section.

As VFV is a design to be used transparently instead of static base NLDB API, the implementation architecture of the iterator has to satisfy the interface standard requirements. For this reason, we relate the iterator class (vpin_iterator) to the CSI entity pin_iterator as shown in Figure 5.7-3.

The object of the pin_iterator class can get the pointer to either bpin_iterator or vpin_iterator, flexibly. The vpin_iterator is responsible for the iteration type described in the example above. The object of this class is returned by the instantiation of the `Virtual_Node` class, which represents the container of pins that are to be traversed. Note that the destruction of the delegated dynamically instantiated specialisations of bpin_iterator is handeled employing the concept of smart pointers {**...** }. This two level architecture allows even runtime switches between two implementations of the iterator class. Note that the first level employment of polymorphism was not possible as the iterators are in most of normal application cases statically instantiated in the program environment.

The class `vpin_iterator` that is proposed as the implementation architecture for the concept presented in the example above consists of one stack and several types of hierarchy traversing class definitions, whose objects are maintained as stack entries. These objects are introduced for each source node, which is taken in focus during the transversal. There are three basic types of stack elements:

- Context,
- DownTheHierarchy and

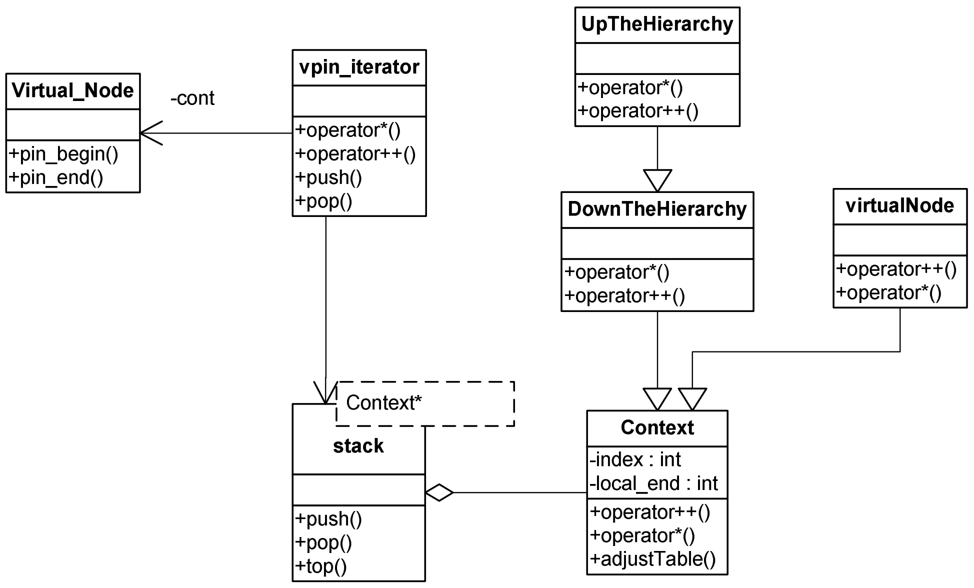


Figure 5.7-4 –Architecture of vpin_iterator and it's relation to Virtual_Node. Vpin iterator defines a stack that is populated by the family of classes that inherit from the clas Context.

- UpTheHierarchy.

They are defining traversing for three semantic types of Access_Nodes, local node, root node and the port, respectively.

Apart from the defined classes, later, together with the introduction of the MFDP committing step that alters the original netlist hierarchy, a specific wrapper class, following the decorator pattern [48] will be introduced. The goal of this class is to group a family of virtual node objects at a single hierarchical level and combine the information they carry. The relation between the stack elements is given in Figure 5.7-4.

The class vpin_iterator holds the stack of context saving entries and it offers the full standard interface of the iterator. From this interface, calls are forwarded to the *top element* in the stack. More precisely, operator++() would forward its calls to the analogue function of the stack current top element, as long as it returns false, or the stack contains elements. The operator++() function is given in the Figure 5.7-5.

```
virtual bool operator++()
      {
       while (!empty())
       {
       setVirtualTable(this->stack.top()->getvTable());
       if(this->stack.top()->operator++())
         return true;
       else
         pop();
       }
      return false;
      }
```

Figure 5.7-5 – operator++() method of the `vpin_iterator` class

Each of the context saving classes defines its own operator++() and operator*() in order to be able to receive the forwarded calls.

The class `Context` is the root class of the hierarchy of context saving stack entries. It is able to iterate over the simple local node, to which it is paired, using its operator function operator++() that has a boolean return value. In case that the iteration leads to a valid (next) object, the function returns value true, after it finishes the iteration over statically assigned node elements, the operator++() function of this object returns false. The return value is taken over by the global operator++() function (defined in the scope of the class `vpin_iterator`) which pops the object from the stack. This strategy is general and is happening for all members of stack.

The class DownTheHierarchy has an additional property. After traversing all statically instantiated device pins, which are linked to the given root node, it traverses over all instance pins, changing the hierarchical level to the definition of the given instance by altering the state of the `Virtual_ContextSaver` context-saving tree. Once the level is switched, an appropriate context saving entry object is created and paired with the new local source view. A new entry object is then added to the stack. The global operator++() calls are forwarded from this moment to the new top of the stack. Note that, in order to avoid loops in the case of the traversing from a higher level to the lower level, instead of the `UpTheHierarchy` entry, a simple context class entry or a (further) DownTheHierarchy node is saved. In the case that the object is created from the lower context, a source_pin is noted in order to skip this path while traversing the instance pin vector, in order to avoid returning to already visited part of the hierarchy.

```
     virtual bool operator++()
     {
      if(!context::operator++())
      {
         while (instPinIterator < instPinIteratorEnd)
         {
           if( local_node->instPin(this->instPinIterator)==
source_pin )
           {
             instPinIterator++;
             if(!(instPinIterator< instPinIteratorEnd))
                break;
           }
           if (produceNewContext())
              return true;
         }
         return false;
      }
      return true;
      }
```

Figure 5.7-6– operator++() of the class DownTheHierarchy

The implementation of the operator functions that belong to the `DownThe-Hierarchy` class first employs part of the algorithm that checks local nodes, defined in the `Context` class and than proceeds with switching contexts along the possible paths down the hierarchy, Figure 5.7-6 .

A further upgrade of the functionality of the class `DownTheHierarchy` is encapsulated in its child class `UpTheHierarchy`. This is the point where the control of context switching is implemented.

The class `UpTheHierarchy` is able to switch the context to the upper hierarchical level. It is performing this operation through its version of operator++(), as shown in the block diagram in Figure 5.7-7. After executing the functionality of simpler methods (father classes in the class hierarchy, `DownTheHierarchy` and through it Context::operator++()), operator searches the next proper context and if it is found, changes the focus of the context saving tree, picking a new source node, for the newly introduced hierarchical level and adds another entry to the stack. Next time operator++() method of the vpin_iterator is called, the call is forwarded to the new top element in the stack. An important property of the `UpTheHierarchy` class is that it maintains only one proper context, deleting all other invalidated neighbours, belonging to other abandoned contexts, like the set N_1 in the example in the Figure Figure 5.7-2.

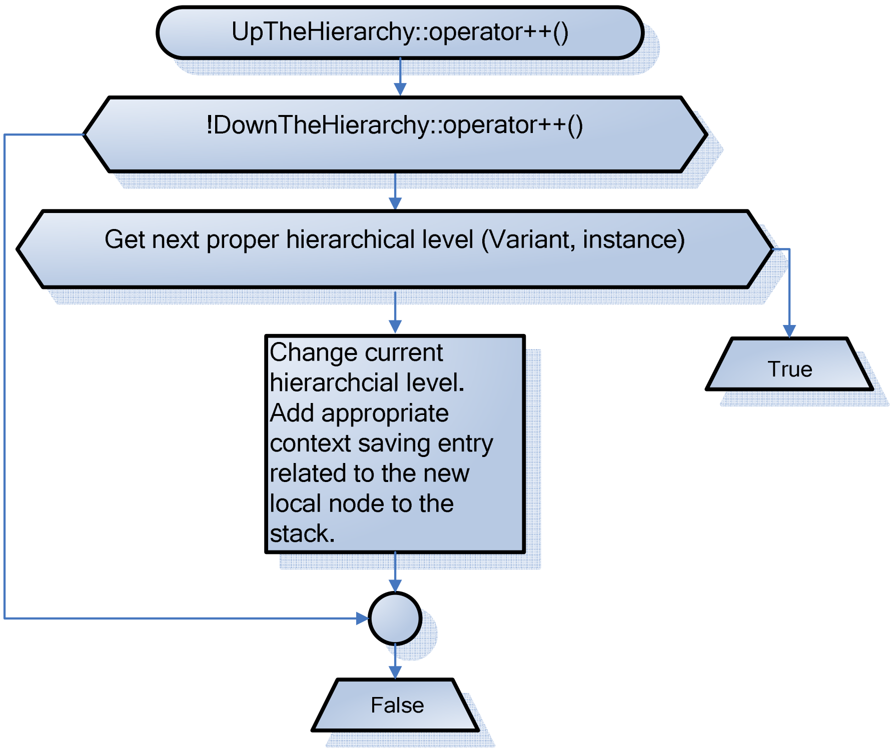

UpTheHierarchy::operator++()
!DownTheHierarchy::operator++()
Get next proper hierarchical level (Variant, instance)
Change current hierarchcial level. Add appropriate context saving entry related to the new local node to the stack.
True
False

Figure 5.7-7 – Block diagram of the operator++() method from UpTheHierarchy class

With these three kinds of context saving stack entries, it’s possible to fully implement traversing the multi-context, hierarchical node.

The explanation of the class VirtualNode is linked with the strategy of committing the MFDP (altering the hierarchical database). Therefore, this will be explained later having in mind the sections to follow.

It is important to stress that the algorithm we have proposed in this section enables us to traverse the hierarchical node potentially using the meta hierarchical data in order to optimize the iteration. As it was mentioned before, hierarchical designs typically have supply nodes with extreme deepness, which connect literally all active elements of the chip (section). For the application of pattern matching, for example, it would be unlikely that some pattern is connected over the node that has a very big deepness, meaning that some of the neighbouring pins of the hierarchical node belong to cells whose placement and semantics are far from the pattern that is being explored, sitting in the original, starting context. For this reason, it is possible as well, to add to our traversing concept principles of Constrained Graph Exploration, such as *tethered robot search* [55]. This means that the distance (rope length) from the starting context is defined. As our iteration changes the hierarchical context, the remaining rope length is decremented until it reaches the edge of the possible exploring radius. The need to implement this was not approved through the test phase of our pattern search algorithms. Note that this approach is approximate, as there is a possibility that some of the semantically important neighbours are skipped and cut of by the introduced “distance” criteria.

5.8 Multi-context (overlapped) flat data portion

In this section we are going to connect the concept of multi-context node, described in the previous section, with the concept of the MFDP that represents the flat view on a small part of the hierarchical database.

Let us consider the following simple example design, Figure 5.8-1. The design consists of two instances of the identical cell/variant (Res), which contains only a single device – the resistor device. This cell is instantiated two times in the top level. Possible flat views that can be created for this design are: res device alone, which sits in the context of the cell `Res`, the lower (leaf) cell of the definition tree of the given design. In the figure, this view is shown under (a). Another MFDP that can be produced, as the augmentation of the previous view, is the serial connection between the resistor device, of the cell Res and the capacitor device that lies in the top context (b). Third possible state of the MFDP is the serial connection of the resistor and inductance (c). All three topologies contain the multi-context node p_1. Its state is, however different. Its state, as described in the previous section, defines the neighbours of the resistor device of the cell Res. Note that the topology where one serial resistor is connected to two devices (d), an inductor or a capacitor is, of course, forbidden! Contexts top/X1 and top/X2, are according to the definition of the multi-context node mutually exclusive.

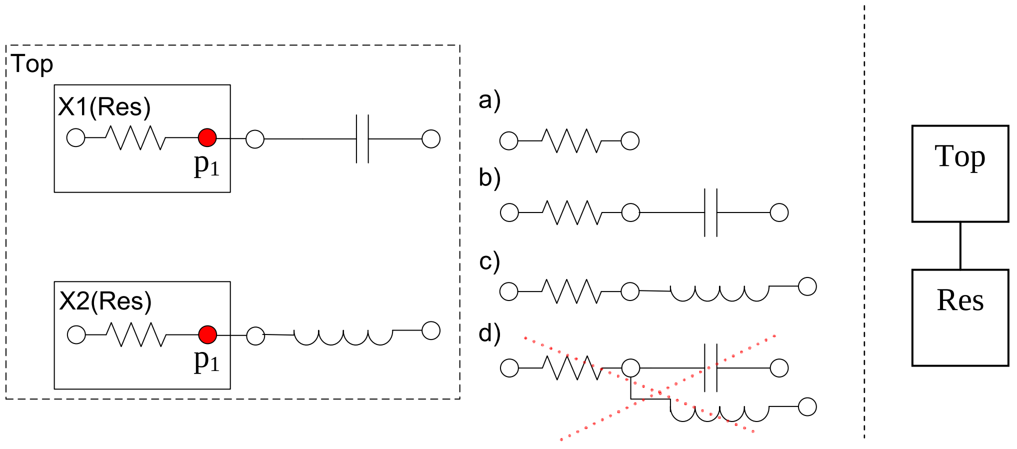


Figure 5.8-1 – Multi-context Topology Example

In this light we can observe the virtually flattened view’s MFDP as a current state of the multi-context topology in, as our example shows, three different discrete time moments. Note that this hypothetical discrete time changes happen after each call to the operator++() method of the `vpin_iterator` class.

Therefore, a topology that contains multi-context nodes represents a multi-context topology. It is defined by the starting point (a device which is selected by the `DeviceFlatContainer::iterator`). The starting device in our example is the resistor device of the cell Res. In case that we have chosen the capacitor as a starting device, this multi-context node would have only two allowed states (and one context): capacitor device alone and the serial connection between capacitor device and a resistor device.

The number of states of the multi-context pattern corresponds to the finite number of different flat data portions that can be built out of the given starting point of the hierarchical design, by navigating. Number of states of the multi-context pattern is strongly dependent on the number of states of each of the multi-context nodes it might contain.

For any algorithm that is using the proposed hierarchical framework, information about the current context of the multi-context pattern is important. Therefore, this, additional, information related to the hierarchical organization has to be handled by the user algorithm. For this purpose, described functionality can be defined as a characteristic interface.

One possible definition of this interface are functions:

- `static int getContextIndex()`,
- `static void lockContext()` and
- `static void unlockContext()`.

These three functions belong to the `Virtual_ContexSaver` class, a context defining class (context carrier) of the VFV. The functions are statically available to the user algorithm.

Functions, `lockContext()` and `unlockContext()` force the algorithm to iterate only over the neighbouring elements (states) inside the current context, and allow multiple contexts, respectively. Note that the navigation can be started with the property lockContext(). In this case multi-context node and MFDP property is switched off.

Function getContextIndex(), simply counts the number of hierarchical levels, from the, starting device. This simple information can be passed to the user algorithm in order to allow the simple test each time the virtually flattened view is to be augmented. With this information the user algorithm can detect context switches and coordinate its execution flow to it.

This functionality can be useful for the algorithms that incrementally collect information from the view and calculate the cumulative results in certain points. For instance, it can be used for the purpose of parasitic networks analysis [56]. The algorithm would start calculating the total (terminal to terminal) resistance of the parasitic network and the total capacitance, in a bottom-up variant walk, for each root net the results are to be stored and intermediate results, for each cell itself can be reused and just augmented for each new value of `getContextIndex()`. Nevertheless, this algorithm would require hierarchical netlist with parasitic information, extracted from the layout, which is currently not common.

One other usage example is the search oriented pattern matching whose example algorithms were represented in the chapter 2. While matching a certain pattern, the algorithm will include the context information in its backtracking. The return value would be flavoured by the context number.

This short pseudo code explains the given situation:

```
match(current device);
{
pick_next_terminal (current device)
return_value = recourse(chosen_terminal);
}
```

while (return_value is zero, or if the context index of the match is bigger than the current context index and there are more terminals);

In this case the hierarchical netlist that is created, for the embossed hierarchy (secondary into the primary) is optimal. Hence, if we are iterating bottom up, starting from a certain cell, the same variant of the cell would be used in any context in which the distributed match appears.

Negative side of this approach is that the algorithm has to be specially written for this purpose, using this simple interface to control context swithing of the hierarhcial data. In this case, simple upgrading of the legacy hierarchical tool and using it as a utility for the underlying hierarchical engine wouldn't be possible.

An alternative to this approach is to hide the interface inside the hierarchical engine (VFV).

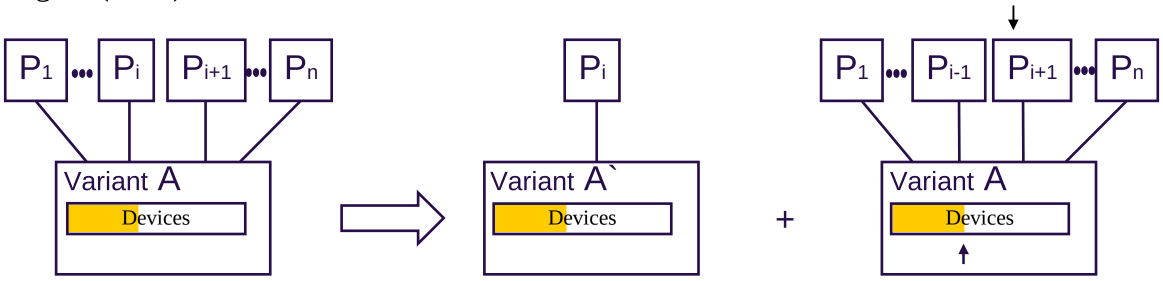


Figure 5.8-2 – Motivation for the introduction of memento. Variant split-up. Child variant is given as a Variant A. The device iterator "progress bar" is indicating that a number of starting points has been already chosen. Child variant has n parents. After split up, the position of the next iterator element is shown.

This is possible for a certain types of algorithms. For the purpose of pattern matching, the algorithm can exhaustively search for the incident devices for a given node, and implicitly switch the contexts.

This solution includes maintaining a memento of the current state of the multi-context flat data portion. Information inside the memento is maintained and used, when the backtracking search process is interrupted, once a successful match is found and committed.

Memento class should save the starting point of the next pattern search (the state of the iterator), and than as a list, each of the alternative iteration starting points of each multi-context node that exists in the given MFDP. In this way we achieve the optimal algorithm execution. The example in Figure 5.8-2 shows the situation where the algorithm iterates over all the devices of the variant (cell) A. The progress of iteration is marked with the yellow ribbon in the device vector. After acquiring a certain device, the MFDP will continue to the parent cell (P_i). If the MFDP gets now committed, the topology of both variants changes and therefore we have to create a new variant of the cell A called A'. This variant has the cell P_i as a parent cell. The memento saves the position of the iterator that locates the current device and also one that picks the right parent cell. The further iteration will continue than over the remaining devices of the new variant A' and than, using the memento information continue with the next device, skipping first n devices.

This strategy was implemented as our solution for structural pattern matching in hierarchical netlists. The approach with memento allowed us to use the original, flat, pattern matching algorithm without further upgrades that control the context switching. This approach uses the flat algorithm transparently, but the referenced cells graph that is created as the result is not completely optimal. By using this approach we can get several identical variants of the design cells which introduces the redundancy.

5.9 Committing of the MFDP (and it’s repetitive use)

In the so far presented text of the current chapter, we have depicted the concept of the Virtually Flattened View (VFV), its architecture and the specific, complex data structures and algorithms that make it feasible. Additionally, we have presented new concepts, such as a multi-context node or the multi-context flat data portion, which have emerged together with the overall idea of the VFV. To make the concept of the VFV more powerful and flexible, we shall define the way to commit the results of the local evaluation to the hierarchical database.

This process can be seen as embossing the topology of the materialised flat data portion (MFDP) into the primary hierarchy. By primary hierarchy, we assume here any “starting” hierarchy on top of which the MFDP has been created. This concept therefore enables the modification of the hierarchical structure of the given design. More precisely, it alters the topology of the variant graph. For instance, in the example circuit in Figure 5.5-2 , if we isolate the inverter whose elements are distributed across the hierarchical levels and we want to commit its topology as the separate cell/instance, we must create the additional variants of the cell MN (and MP), specially for the instantiations in the cell A. These variants will be missing devices mn and mp, respectively. This is done as the devices that previously belonged to the given variants of the cells MN and MP are now moved to a new subcircuit (inv). The instance of the newly defined cell inv is placed at the variant of the cell A. Consequently, committing of the match requires several operations. These operations alter the affected design hierarchy and build the subcircuit and the instance of the given new hierarchical attribute, placing it correctly in its environment. After the committing process the modified hierarchy “looks and feels” like any proper hierarchical design. Thus, it is ready for some future proper usage.

We can now conceptually define the algorithm that commits the given state of the MFDP:

1. Refine the MFDP leaving only the instantiations of the relevant devices.
2. Add the references to the elements (devices and instances) that belong to the MFDP the new subcircuit definition.
3. Add all local nodes of the MFDP to the subcircuit.
4. Create the instance of the new subcircuit and attach it to the appropriate variant
5. Handle the pins of the newly inserted instance, attaching it properly to its environment.

When the user algorithm works on the MFDP, it flattens (creates) also some “noise” - the elements which are neighbouring the relevant data of the MFDP that are important for the algorithm execution. This common scenario happens while, for instance, one performs the pattern matching, or isolates a specific parasitic net that is being evaluated, from its environment. Therefore, some of the elements of the MFDP that are considered as the environment have to be chopped off leaving only the relevant data.

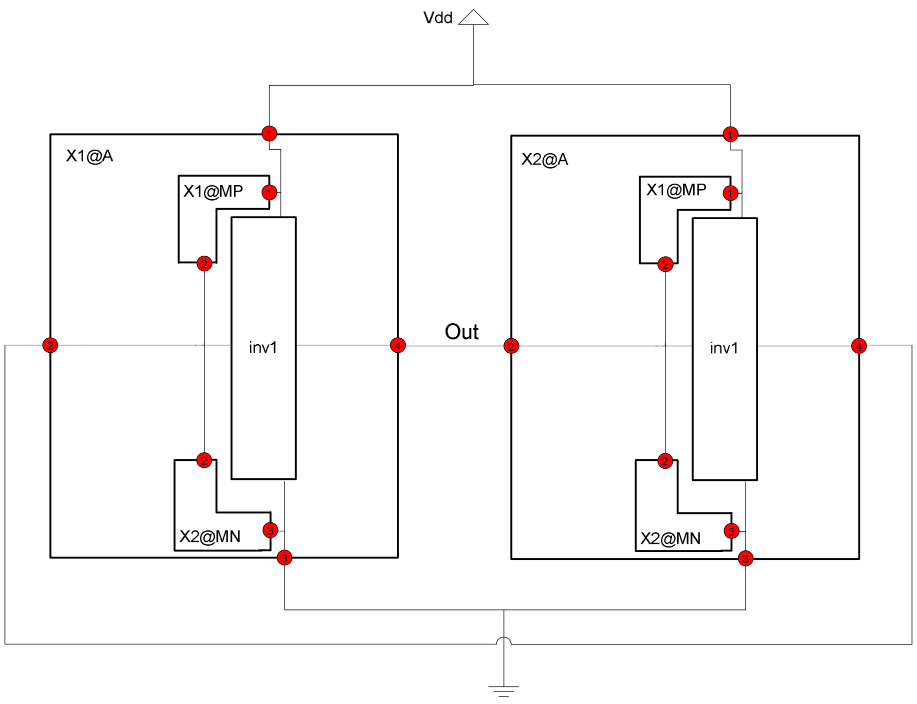


Figure 5.9-1 – Embossing step

We can conclude that the "carving out" algorithm behaviour depends on the specific application domain. Thus, we will provide the template algorithm, which makes the functionality that refines the content of the MFDP interchangeable. A domain specific constraint is than separately defined. Therefore, the right place for the definition of this function is the `Virtual_ElementBuilder` class. Exactly the class whose specialisation is created upon the decision on the application domain for the framework we propose. Hence, the function that refines the MFDP performs a walk over the context saving tree, recursively eliminating all the elements that are considered redundant and keeping the relevant structure. This operation is expected to have no influence on the algorithm complexity, as it just removes the "noise" whose acquiring (materialisation) and analysis is considered as the part of the user algorithm complexity. Therefore, the mentioned strategy is expected to add just a constant to the overall complexity of the application domain algorithm.

Steps 2, 3 and 4 are more or less trivial. Simply, all remaining elements of the MFDP are referenced in the new subcircuit definition. In the case of the nodes, a simple test is performed to check if the node is local and has no additional connections outside of the MFDP. If not (the node is local), its reference is copied to the subcircuit as well. This step forms a proper bipartite graph, together with the device (instance) elements that is placed into the new subcircuit.

When this is done, we add the instance object to the (relative top) variant, which is identified by the context saving tree. This process will than alter the variant's topology in all of its instantiation places. Note that also other variants than the relative top, deeper in the hierarchy, might be affected by the embossing step. In that case, we have to create new variants that have the modified topology (missing the devices that are moved to the new subcircuit). We identify these two processes as processes of "covering" and "splitting" variants. The efficient algorithm to perform this step is explained in section 5.10. The outcome of the process of embossing can be depicted with the example in Figure 5.9-1. In this example, we have embossed the current state of the MFDP from the example in Figure 5.5-5. The example shows the resulting hierarchical design where a new instance is inserted into variant of the cell A. This variant is “covered”, therefore the change is valid for all instances of the given variant of the cell A (instances X1 and X2 in our example). With the collapsed rectangles for the cells MP and MN, we intuitively show that their topology has changed (they have lost the transistors mp and mn, respectively). The change, which was done here, is valid just for the relevant instantiations (in the given variant of cell A) of the variants of cells MP and MN. These variants are, thus, “split” from original variants of the cells MP and MN, respectively.

The step 5 includes another complex algorithm that will prepare the context saving tree and the affected hierarchy for the insertion of the new instance. In this step, it is necessary to remove all redundant information about the node mappings for all pin nodes of the newly inserted instance. Hence, the context saving tree contains redundant mapping between the source subnode and the virtual node at all relevant hierarchy levels. Referring to our example, the node 2 of the level MN is mapped to the virtual node on the appropriate level and further on the level A. The reason why this redundancy was introduced is to enable determination of the proper node mappings in an efficient way, at any current hierarchical level. In a word, the introduced redundancy helps the efficient implementation of the view navigation/augmentation.

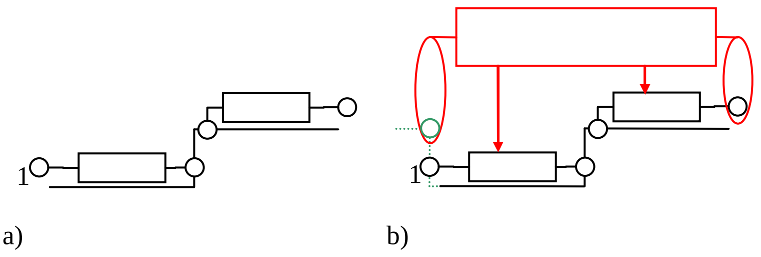


Figure 5.9-2 – Example of port creation. (a)The serial connection of two resistors is distributed over two hierarchical levels. (b) If the pattern that was searched was two resistors in a series, the block that they are abstracted in exists in the higher subcircuit. In order to connect it properly, we insert an additional port node to the lower cell; and the relevant root node in the higher cell.

In addition, it is necessary to “bring up” all the nodes, to which the pins of the newly inserted instance should be connected to, to the relative top level. This process is necessary to provide a proper connection of the new instance with its environment. This process creates some new nodes and to the affected hierarchical levels, if necessary. One situation when the node generation is necessary is depicted in the example in Figure 5.9-2. Under (a), the serial connection between two resistors is shown. The resistors are distributed over two hierarchical levels. That is sketched by slightly shifting one resistor above another. If the user algorithm abstracts the serial connection between two resistors as a subcircuit, the VFV will add the instance of the new abstraction to the relative top level and connect it with two pins. In order to connect the node 1, that was, originally in the lower hierarchical level, it has been transformed into a port and an additional root node in the higher cell is created. This port therefore, alters the variant of the lower cell by adding an additional pin (port node) to it.

In order to give a common and efficient answer to all mentioned operations we create two types of walks over the context saving tree structure. One will prepare all the nodes for the commitment and create necessary additional pins for the relevant hierarchy levels (instances and its definitions), while the other, in a single context saving tree walk performs the committing step.

As the result of the described process the variant graph is altered by a new variant that holds the introduced instance (on the relative top level of the context saving tree) and by additional new variants that have an alternated number of pins (additional pins) and devices (they have lost some devices) than the original variant version. The modified variant structure is valid and ready for further usage/alterations. The technique that is developed for the variant graph alteration enables optimal runtime as all changes are done locally, with the respect to the MFPD and prior variant topology. The technique includes the dynamic variant creation (operations of splitting and covering of the existing variants) and the technique of layering. These two concepts will be given in detail in further text (5.10).

5.10 Distributed variants

Distributed variants are the concept which is developed in order to support the VFV. This concept enables quick and efficient alterations of the hierarchical design (variant graph). The strategy is to represent each variant with a group of objects. Partially, depending on the similarity between the variants different entities share the objects that represent them. This process supports the embossing step of the MFDP making it more efficient.

5.10.1 Technique for the topology adaptation

The main principle of the concept of the distributed variants is the technique of topology adaptation by variant layering. In this concept, one can define new variants by grasping only differences between the current variant and the modified one, hence combining the starting variant with the specially defined layer to get the altered topology.

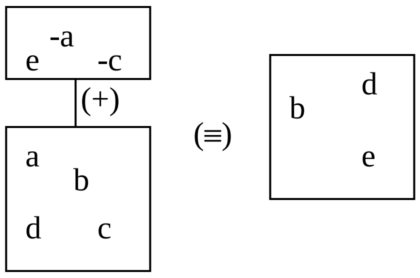


Figure 5.10-1 – Topology adaptation principle example.

Let's consider the set of vertices of the bipartite graph that represents the topology of a given variant: $V = \{X \cup Y\}$. The sets X and Y are the sets of elements (devices) and nodes, respectively. We are for now interested in the set of elements X. We can observe X as a multiset MX, defined as a pair (X, m). m is the multiplicity function defined as

$$m : X \to \{-1,0,1\}.$$

Therefore, we can write that

$$MX = \{(x_1, m(x_1)), (x_2, m(x_2)), \dots , (x_n, m(x_n))\}.$$

If a given instance of such a multiset contains exclusively pairs with nonnegative multiplicity values, we call it a *base*. The *layer* is a multiset that contains pairs with negative multiplicity values.

In addition, we define the operation that performs the topological adaptation (+). This operation can combine a layer with the base. The operation is only possible with the compatible layers and bases in order to get the product of the operation that is again a valid base, containing exclusively positive multiplication values.

As an example (Figure 5.10-1), if we have the valid base B = {a, b, c, d} and a layer L = {(a,-1), (c,-1), e}, B+L = {b, d, e} is another valid base! Thus, we can observe this layering process as a recursive operation, adding an arbitrary number of layers on the top of a single base:

$$B_{i+1} = L_i + B_i.$$

In this way, we have defined a technique for altering the semantics of the topology of the given valid set, by chaining a number of objects. The important property of this structure is that the proper sets can be "seen" from any layer by "looking down" to the atomic base set in the end. Therefore, from a starting set, we can form a family of similar sets, strictly by saving differences between them.

If we map this principle to our object oriented vocabulary, we can identify `Access_Cell` as a base. Special class `Virtual_Variant` is introduced to model the layer. `Virtual_Variant` referes to the `Access_Cell` (base) through a method getSource(). The class has a list of elements that are excluded from the base (formally represented with the negative multiplicity values) and a list of elements that are added to the variant. The list of elements that are to be excluded is delegated and represented by the object of a class `Virtual_Excluder`. This class is actually another role of the context saving tree object, `Virtual_ContextSaver` (see Section 5.6). Therefore, all elements, which belong to the `Acces_Cell` (cell/varint) and are contained in `Virtual_Excluder` are eliminated from the resulting variant. The depicted architecture is presented in Figure 5.10-2.

We can use the defined technique to alter the topology of the cell/variant graph. We will e.g. observe a case where a serial connection between two resistors R_1 and R_2 (that exist in a single variant) is highlighted by the MFDP. The MFPD thus consists of two virtual copies of the source resistors (VR_1 and VR_2). The example is illustrated in Figure 5.10-3. The proper object of the type `Virtual_ContextSaver` assures the consistency of the paired source and virtual objects and properly positions the MFDP relative to the primary hierarchy. In order to

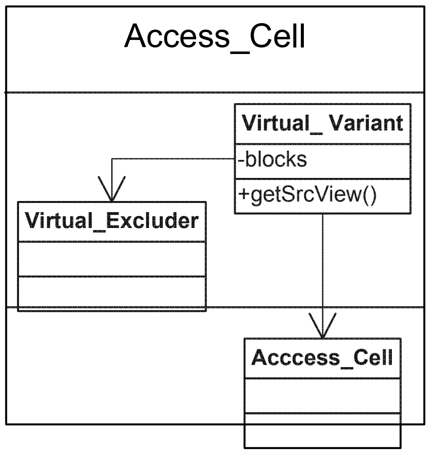


Figure 5.10-2 - Recursive representation of the abstract interface of the class Access_Cell.

emboss the state of the MFDP back to the hierarchical netlist, we employ the technique of topology adaptation. Therefore, we would use the `Virtual_ContextSaver` object (in this case the only member of the context saving tree), seen as the `Virtual_Excluder` to specify the list of the elements that are not any more contained in the variant (the both resistors of the serial connection). Additionally, the instance of the new abstraction (Block_1) is added to the `Virtual_Variant`. We see now the group of the `Virtual_Variant`, `Virtual_Excluder` and the former `Access_Cell` as a new `Access_Cell (new variant of a cell)`.

This kind of architectures enables quick changes / insertions of new variants. If the difference between two variants of a cell is e.g. in one element, the change is done just by specifying the element that determines the difference instead of copying all n-1 elements while forming the new variant definition. Another advantage of this concept is that all information is present. This enables easy undoing or back annotating. It is important to add that by usage of this concept it is possible to perform the concept of *semantic layering*. The semantic layering concept enables partitioning the database according to the complexity of objects that are instantiated in it. We will explain this technique later in this section.

The price that has to be paid for the benefits that are gained lies in the fact that the data that describe variant are distributed over a number of objects. For each reference to a given variant, a lookup operation has to be performed in order to extract the actual data. This is done during the iteration/navigation over the elements of a such variant.

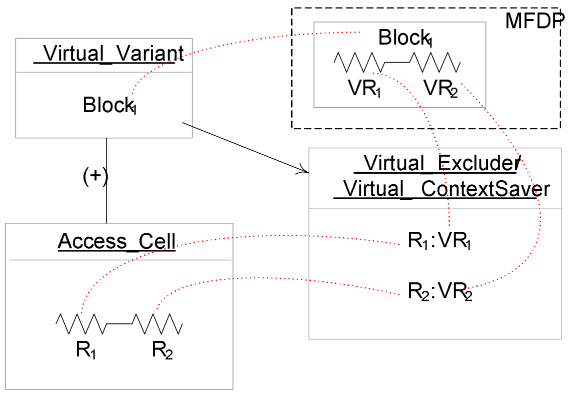


Figure 5.10-3 – The example of the technique of topology adaptation.

5.10.2 Dynamic variant creation

Once we have established the principle of topology adaptation, we are going to apply it to the variant graph. Each element in the variant graph is connected up the hierarchy (this connection is determined by the number of references to a parent variant) and down the hierarchy, by a subvariant vector that links the current variant with all children variants. We define two types of operations that employ topology adaptation principle. The first type is called *covering* and the second *splitting*.

We can cover a variant of a cell by altering its content (excluding some elements from the variant and adding some new elements). Still this change becomes valid for all instantiations of the variant, therefore, the update is done by an additional object, but the paths are read from the original parent vector (of the previous object).

This operation doesn't change the topology of the variant graph. It just alters the content of the variant itself.

The second operation splits the variant from its father variant. In this case, only one path is being separated from the original variant and in general we have two variants available. Note that if there was only one instantiation of the father variant, splitting it will lead to an operation that is somehow similar to covering. The outcome is than, that the father variant is no more accessible, but only its altered semantics through the layered variant (the layer that augments its semantics).

These two operations are being done “on the fly”, while creating the MFDP. Therefore, once the initial context is determined the current relative top variant is covered. In that moment the Virtual_Variant object that represents the layer which can change the semantics of the base variant is void. If the algorithm, from this place wants to go up the hierarchy, or down the hierarchy, the appropriate variant will be than split! In the case of going up the hierarchy the new relative top variant is, again, covered.

In order to ensure very fast operations, the variants that are split and covered during the evaluation phase (the phase where algorithm accesses the data in a flat fashion),are in a “non-validated” state. In this state, the objects that should augment the semantics of the father variant have just a part of the necessary information and they use the `Virtual_ContextSaver` objects to save the relations between them. Therefore, variant graph is still not altered. This is very important as in this way, the VFV algorithm, can instantly detach new objects, if for instance it decides to move the `DeviceFlatContainer´s` iterator to another position. If the embossing step for the current topology of MFDP is evoked specific algorithm alters the variant graph.

5.10.3 Virtual variant tree

As we have defined now the steps of covering and splitting of the variant, by combining them we produce the virtual variant tree. This tree represents the group of variants that are produced out of a single variant from the initial variant graph, created by the standard variant creation algorithm. In the tree we distinguish the root element, which is the identical (trivial) excluding/upgrading of the input variant. By applying a number of “cover” and “split” operations the tree grows and forms the group of leafs. Those are all “visible” elements of the tree, they form “access points” for newly created variants. The path from each leaf of this tree to the root element represents the whole semantics of a single variant, according to the principle of excluding/upgrading. In order to implement this tree we use the `sourceView` reference and the specially ordered vector of active elements. From each of these active elements, it is possible to navigate towards the root, following the `sourceView` references. Once a variant is covered, or if a variant with a single parent is split, it gets deleted from the list and stays in the body of the tree still giving its contribution for the excluding/upgrading technique. In this way, we have achieved to dynamically and implicitly alter the content of any variant, while building the context saving tree.

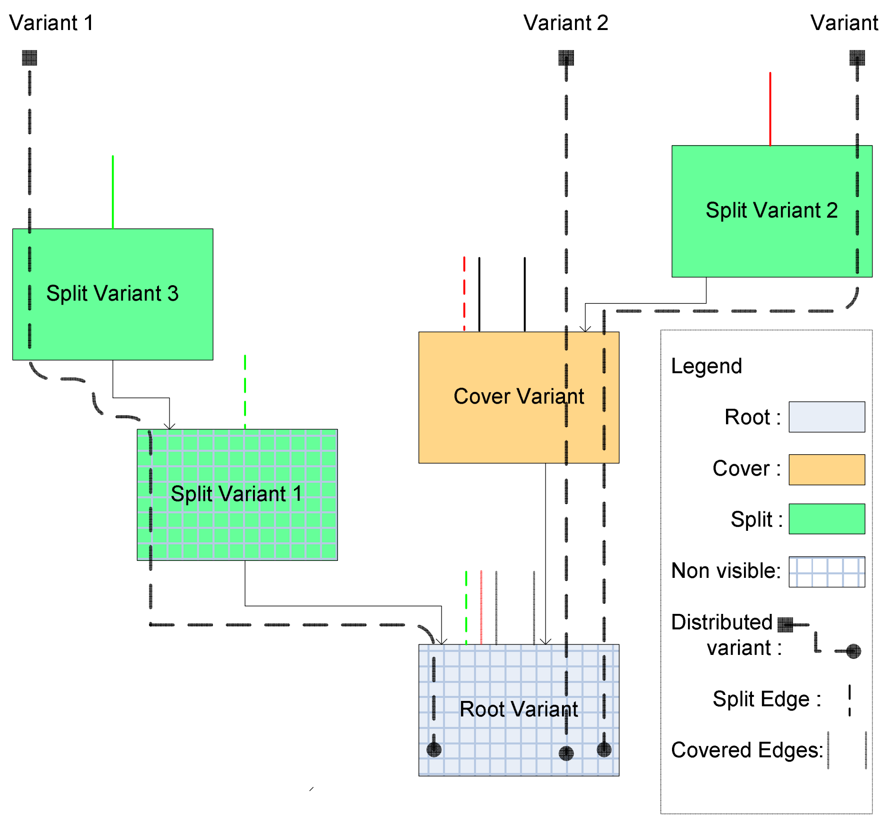


Figure 5.10-4 – Distributed Variant Tree – The root variant has been split in tree, by arbitrary application of cover and split operation.

In order to illustrate the properties of the Virtual Variant Tree we will consider the example given in Figure 5.10-4. In the example we see the state of the distributed variant tree after four operations of splitting or covering. In the beginning we had just a single variant with a number of parents that it refers to. We have sketched the paths to the parents as lines that are given in tree colours. The first operation that was performed on this example structure was the operation of splitting. This has caused the introduction of the variant Split Variant 1, while the relevant single path was moved to the newly introduced variant. After this process, the root variant and the Split Variant 1 have been the members of the leaf variant list. The next operation that was performed is the covering step for the root variant. After this step, the root variant transfers its all parent paths (remaining, not including one that was already transferred to Split Variant 1) and it stops being visible. New virtual variant object is instantiated and its source link is set to Root Variant. Further, after the covering step we have depicted Root Variant has been exchanged with Cover Variant in the leaf variant list. Therefore, in this list we have now Split Variant 1 and Cover Variant. In another action we split one of the paths (red line) of the Cover Variant and form Split Variant 2. In the end, split operation has been performed on Split Variant 1. Since it had only one parent path the variant becomes invisible and replaced by Split Variant 3 in the leaf variant list.

After the outcome of this process, we have three variants and their semantics are acquired by referring to the tree:

Variant 1: Split Variant 3 -> Split Variant 1 -> Root Variant
Variant 2: Cover Variant 1 -> Root Variant
Variant 1: Split Variant 2 -> Cover Variant -> Root Variant

As we can see the variants that are the result of the growing process of the distributed variant tree are spread over it.

We will now in greater detail explain the semantic layering concept. Consider a transistor level design. If we apply rules to abstract all transistor devices into gates, we see hierarchical design at the gate level. Suppose that a part of the design was analogue, hence no digital circuit were isolated. Since all the information about the logic gates (about the abstractions that form logic gates) exist in layers above the atomic variants of cells, after applying the topology adaptation technique, we can define from which layers (and how deep into the distributed variant tree) the information will be read. We can thus exclusively read the data about a given variant that is stored in specific layer. If we classify the layers introducing indices for them, we can specify the complexity of the data they carry. For instance, if we had rules to extract all NANDS we can cast variant layers that are adding those conclusions to belong to specific class that carries index value 1. All root level variants, carrying information about transistors are assigned by value 0. Employing this principle, during the iteration exclusively elements that are stored in layers of the class 1 can be acquired. The algorithm of traversing the variants would recourse deeper to the part of the distributed variant which has more basic elements in it. In this way we can partition the database into concentric shells and pick appropriate shell to see the underlying data on the wanted complexity level.

In the end we will explain briefly how these structures are used to perform dynamic variant creation, algorithmically. If the algorithm starts creating MFDP from some arbitrary variant, it gets covered and all devices that are mapped to their virtual copies (members of the MFDP) are then candidates to be excluded from the current variant. The new variant is in this moment non valid. Note that there are no connections from the valid variant to the variant candidate. That means that, if the embossing command is never called, all new non-valid variants can be easily detached and deleted. The architecture used to implement this process allows sending the non-valid objects directly to the garbage collector which is started in a separate thread. If, on contrary the algorithm decides to commit the variant. It would start the embossing process (5.9), prune the data of the MFDP if necessary and then commit the changes, building new variant layers into the variant graph. After this operation, the variant graph is altered and ready for further use and possible changes.

5.10.4 Layered nodes

In parallel to the process of excluding invalidated elements of a given variant of a cell and adding new elements, the completely analogue process is being done for the device pins, using the described technique. In the case of nodes, the strategy is slightly different.

The virtual node that is linked to the instance pin of some instance (of the committed MFDP) adds this information to the set of pins that is aggregated by the `Access_Node` which is the source of the given `Virtual_Node`. That means that the total number of aggregated pins seen from a given node is distributed over several virtual nodes. In order to see all necessary pins that exist as the neighbours of a node of the distributed variant, one has to acquire the pointer to the top of the nodes. All other nodes will be accessed descending from the top node, following its `source-View` connection. In the example (Figure 5.10-5) we have the base (root) variant and in it we have the node N. This node has five pins that it aggregates. During the user algorithm execution we have committed the corresponding MFDP-s that were containing the given node three times. The first committing process has added the virtual node VN1 which references a pin connection to the abstraction which was committed in this first step. All the information about excluding (of the pins) and layering of the nodes is available automatically, by seeing the relevant `Virtual_ContextSaver` object as the `Virtual_Excluder`. Thus, the mapping between the node N and its virtual copy VP6 will be used to acquire the handle to the virtual node. Once other two layers are added the relevant mappings were inherited from the process of the construction of the relevant MFDP. If this structure is now to be used to build some new MFDP, the node can be approached following the pin P5, from the device already defined at the base level, or following pins VP7 or VP8, that were defined in two different layers. In any of these three cases in order to get the source node for some new virtual copy (of the MFDP that can build the layer 4) VFV assures that the top node is picked as the source. This is done by the recursive function:

- getTopNode()

that is called every time a source node from the database is to be read. If the path to approach the node N was through the pin P5, we would need three look-ups to "climb" to the current top node. First the look-up if performed at the `Virtual_Excluder` object of the Layer 1, where we find the mapping between the node N and the node VN1, further at the layer 2, at the relevant `Virtual_Excluder` another mapping exchanges VN1 to VN2 and in the end while looking-up at the `Virtual_Excluder` object of the layer 3 we get the handle to the top virtual node. Once we have this reference we can iterate over all valid pins of this complex layered node.

Note that the number of look-ups depends on the number of layers employed and also on the fact how deep the entry point to this list of layered nodes is. In order to incorporate this process (the layering of nodes) into the multiple context node concept and to allow the proper functioning of the iteration process of the

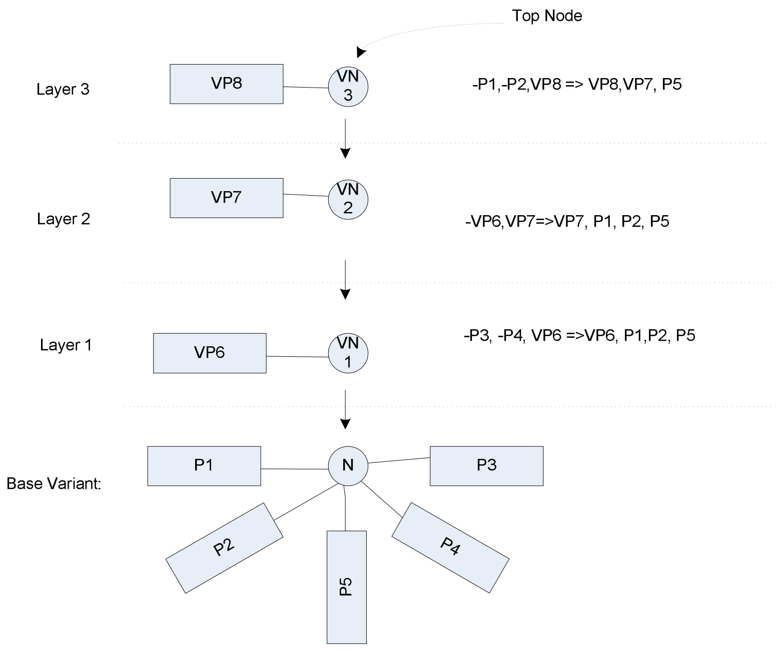

Top Node
Layer 3
VP8
VN 3
-P1,-P2,VP8 => VP8,VP7, P5
VP7
VN 2
Layer 2
-VP6,VP7=>VP7, P1, P2, P5
Layer 1
VP6
VN 1
-P3, -P4, VP6 =>VP6, P1,P2, P5
P1
N
P3
Base Variant:
P2
P5
P4

Figure 5.10-5 – Example of the distributed node. The node is composed of three virtual nodes and one base node.

`vpin_iterator`, we add an additional stack entry class type. We add the new class `VirtualNode` to the inheritance hierarchy of the `context` class (Figure 5.7-4). This class is conceived following the architecture of the decorator design pattern [49].

The strategy here is that any top node, no matter if it was a virtual node or some other instantiation of Access_Node, originally present in the database, once it is approached during the transversal over all pins of the virtual node (using `vpin_iterator`) gets the instance of the class `Context`, `DownTheHierachy` or `UpTheHierarchy` if it is a local, root or the port node, semantically. Additionally, if the node is a virtual node and thus its semantics is distributed over several layers, the object of the decorator class `VirtualNode` is instantiated. The original object that handles the proper iteration is referenced inside the `VirtualNode` class instantiation, together with the source level of the acquired virtual node. Upon the usage of the `operator++()` method of the `VirtualNode` class it is assured that, by using the stack all relevant pins are iterated over recursively. The code which defines this recursive operation follows:

```
bool Virtual_Node::vpin_iterator::virtualNode:: operator++()
     {
      if(!visited)
        {
         if(!decoratedContext->context::operator++())
           {
            context_it->stack.push(decoratedContext);
            visited = true;
            if(srcNode)
              context_it->push(srcNode, source_pin, true);
              return true;
           }
         return this->context_it->operator++();
        }
      return false;
      } .
```

By defining the described set of very complex data structures and the algorithms that are driving them we have managed to realize the idea of the virtually flattened view. This view is now ready for the test application in order to achieve the algorithm for incremental structural pattern matching in hierarchical netlists.

5.11 Summary

In this chapter we have presented the vision and the thorough realisation of the virtually flattened view. The view design allows it to present the data of the hierarchical design locally flat and to commit those local flat data portions back to the hierarchical design as new subcircuits. This functionality is identified to be useful for different applications. The intention is to use the view together with different software projects that were written for the applications exclusively with flat input netelists, neglecting the problems the hierarchical representation includes. For this reason we define the view generically.

For different application scenarios the view has to get some additional properties, mirrored as augmentations of its classes' interfaces, specifically to the given application domain. The design of the view that we have presented allows this to be done with ease and elegancy as it was prepared to be flexible. To achieve this we have applied advanced object-oriented principles. Good performance, in the first row the runtime efficiency, of the operations that are driving the view is allowed by novel complex data structures and novel algorithms performed on them, which are specific to this view.

In the following chapter, we will thoroughly present and value one possible application scenario for the VFV. We will create specific changes to the view and adapt and integrate it to the flat incremental pattern matching project classify (2.5).

6 Application of the VFV to Search Oriented Pattern Matching Methods

6.1 Introduction

Throughout the second chapter we have analysed the problem of subcircuit recognition (SR) in VLSI designs, as a subproblem of graph matching. We have represented among other approaches the incremental SR approach, where one is enabled to flexibly match complex patterns by specifying a set of rules that form a specific descriptive language program. As no algorithms that would perform the hierarchical pattern matching on hierarchical schematic designs are available, we were motivated to search for the solution for this issue and enable the structural pattern matching directly on hierarchical designs/netlists.

The Virtually Flattened View that is defined in the previous chapter with it's functionality:

1. represent any hierarchical netlist using the standard NLDB database objects (following standard object–oriented API, the access layer (AL))
2. create small flat portions (topologies) of netlist data and offer them to the user algorithm
3. commit possibly altered (cleaned up) flat portions of data to the hierarchical database. (commit them to the view)

can be used in order to implement the hierarchical pattern matching algorithm.

In this chapter we will apply the VFV on the already existing project (classify) that implements the incremental pattern matching approach. This project was written to work on exclusively flat input data (flat netlists). In order to accomplish this, we will present the specific set-up of the generic VFV (section 6.2), together with the minimal adaptations of the flat algorithm (section 6.3). Apart from plausibility and functional correctness the hierarchical approach to the structural pattern matching allows obtaining irredundant results. This qualitative enhancement is discussed in section 6.4. The chapter is concluded with the qualitative and quantitative evaluation of the obtained hierarchical algorithm through the extensive tests (Section 6.6).

6.2 Hybrid layer

The application of the classify flat algorithm to the VFV requires a specific set-up of the view. We will flavour generic VFV elements in order to enable the flat algorithm to work on the data it needs in a proper way. We achieve this by specifying a view, whose classes are related to the original VFV classes through the object oriented mechanisms. Particularly, in the case where the interfaces are compatible, or one writes a new algorithm that uses specific NLDB interface as the API (possibly augmented by some additional properties for the NLDB entities), one can achieve this goal by simply inheriting each of the classes with the proper specialisation which provides the objects of the altered view with the proper augmentation of the interface. This task is almost trivial as the view has been designed to be particularly flexible.

Therefore, by adapting the view to the user algorithm we provide the generic view with a specific set-up, creating an application domain specific *hybrid layer*.

In our application case, the goal of the hybrid layer is to connect the VFV to an already existing project, the flat netlist pattern matching tool classify. It connects these two different projects and enables them to work together, serving as some kind of adapter. The application domain project (classify) was written even before the NLDB and its additional LV (Layered Views) mechanism.

Both projects have their own way of representing the electronic circuit. Classify, as it was written to perform pattern matching in flat netlists, has no mechanisms to represent hierarchy, but has the specific interface that enables it to conduct the pattern matching process. On the other hand, NLDB has the classes and the interface (defined by the Access Layer) which are capable of representing also hierarchical data. By "energising" the interface of the Access Layer with the property of virtual flattening we prepare NLDB to provide the "friendly" data layout to the application domain algorithm. To summarise, the hybrid layer has to satisfy following criteria:

1. enable classify the look and feel that it is working with its specific (flat) data model. Particularly, the entities have to be compatible with those used in the algorithm and they have to support the corresponding interface.
2. On the other side, NLDB has to be able to handle the objects, which are provided to the pattern matching algorithm in the flat fashion in its style. That means that the objects have to be compatible with the entities of the VFV in order to be managed in the proper way.

The answer for the above requirements can be found in employing the multiple inheritance. Thus, we position the view on top of both projects (making the relevant projects' classes father classes of the given hybrid layer class).

6.2.1 Positioning of the Hybrid layer

The layer itself is specified as an additional header in the classify project. The classes of the hybrid layer have their analogues in both projects as their ancestor classes. The typical inheritance relation between the classes of the hybrid layer with the classify and NLDB projects is given in Figure 6.2-2. In the figure, in order to explain the standard architecture, we have used the example of the positioning of the class that models the MOS device in the appropriate inheritance graph. On the far left side of the figure we see the domain of the classify project and its entity that models

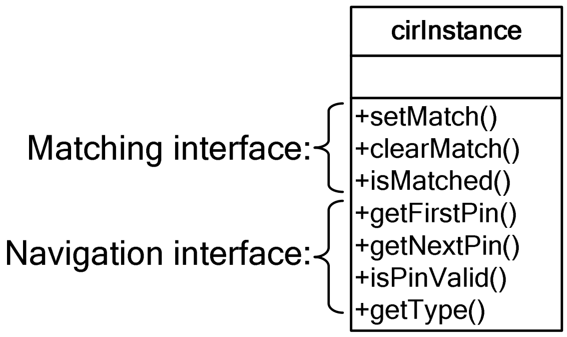


Figure 6.2-1 – Interface of the cirInstance class of the classify project

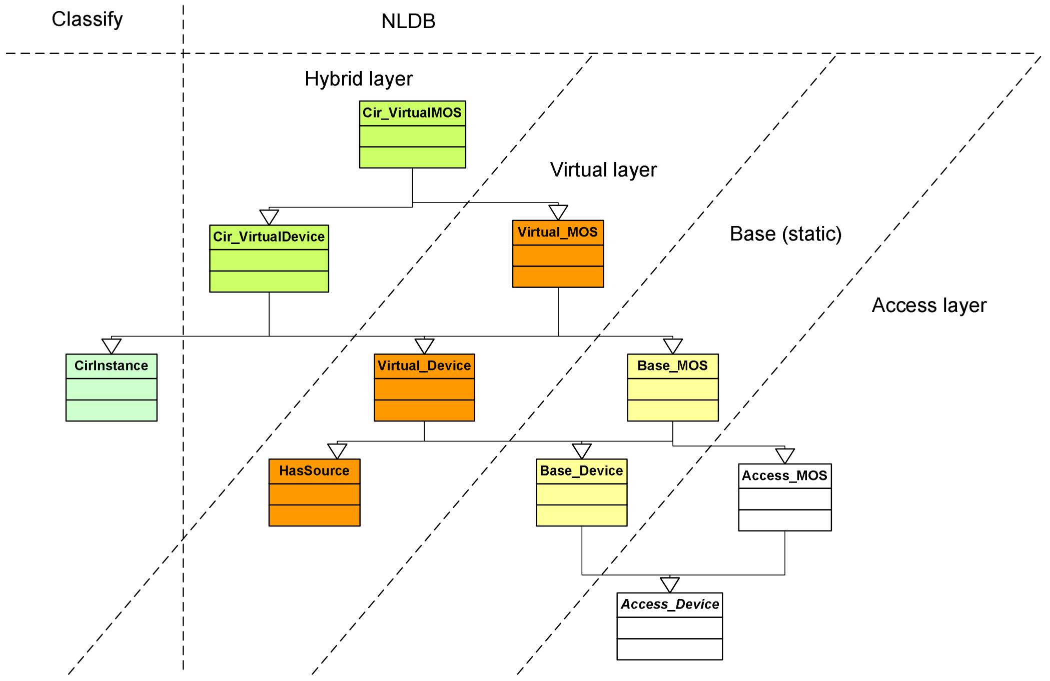


Figure 6.2-2 – Placement of the hybrid layer classes in the Access_Device inheritance diagram

an element (a device/instance) – `cirInstance`. From this class we inherit the class `Cir_VirtualDevice`. Therefore, in any place where the algorithm expects an instance of the class `cirInstance` one can pass, transparently, the instance of some of the realisations of the (abstract) class `Cir_VirtualDevice`.

Looking towards NLDB (right, predominant area of the figure), `Cir_VirtualDevice` is the child class of `Virtual_Device`, as well. Moreover, another class `Cir_VirtualMOS` is written in order to connect the `Virtual_MOS` class of the (generic) Virtual Layer with the classify project. What have we achieved with this (complex) architecture? The classify project models all devices with a single class (cirInstance) and further classifies them according to the value of a method `getType()`, while in NLDB project we have an abstract class `Access_Device` and a family of descendents. By employing such architecture where we have a common hybrid object as a subclass of both `cirInstance` and `Virtual_Device`, we achieve that the hybrid classes that stand for descendents of `Virtual_Device` (`Access_Device`) can be seen from the single class cirInstance in the domain of classify project.

We further have to partially adapt the interface of the class cirInstance, in its descendent `Cir_VirtualDevice`, to the implementation that employs complex algorithms of the VFV. In order to identify exactly the places where this is necessary, we will split the interface of the `cirInstance` into two parts. The first part corresponds to the application domain specific interface. This is the interface which enables the pattern matching algorithm proper execution. Particularly, those are the methods of the first group in Figure 6.2-1:

- `setMatch()`,
- `clearMatch()` and
- `isMatched()`.

The methods that handle the iteration process and the navigation between the elements of the in-memory circuit model of classify project are forwarded to the

NLDB! These methods are identified as navigation methods of the class `cirInstance`. In this way we achieve the property that the objects of the MFDP are presented to classify in 100% adapted way. Classify analyses the objects, navigates their (virtually flattened) neighbourhood and decides if the given topology is equivalent to the pattern or not, than it issues an action of encapsulating the objects which are parts of the MFDP into a separate instance. All the calls are translated to the operations of the VFV which further handles the objects of the MFDP performing the embossing step (5.9). In this way both algorithms manage to see the objects from their "worlds" and to communicate with them using the appropriate methods. Therefore, the whole hybrid view plays a role of a bi-directional adapter.

Analogue to the example of MOS element that we have specified, all VFV relevant classes are connected to the corresponding classes on the classify project side.

In general, the wrapping process can face some interface incompatibility problems. For instance, one project can use the aggregations based on the container-iterator concept, while another can use linked lists. In the first case a special object (iterator) grasps the actual element that is iterated over, while in the second case the object itself has the information about its position in the container. Problems like these can be easily solved by minimal adaptations of the application domain algorithm or employing Adapter Design Pattern [48].

6.2.2 Cir_VirtualBuilder, the concretisation of the Virtual_ElementBuilder

Once we have specified the application domain flavoured classes we have to assure that the objects of this kind are going to be built by the VFV instead of generic NLDB `Virtual_<class>` instantiations. The view design is already prepared for the flexible object creation and all that is necessary to be done is to create an appropriate specialization of the `Virtual_ElementBuilder (5.5)`. To do so, we will specify a class `Cir_VirtualBuilder`. This specialisation class is provided with the implementations for a set of pure virtual functions of the abstract class `Virtual_ElementBuilder`. Therefore, all methods that wrap the object creation are defined here. They are set-up to create new hybrid objects that are derivated from their generic ancestor classes everywhere the VFV wants to create an instance of the `Virtual_<class>` family. For instance in order to build the `Cir_VirtualMOS` class we simply specify:

```
Virtual_MOS* InstVirtual_MOS(netlist::Base_MOS* ptr)
                                { return new Cir_VirtualMOS(ptr);}
```

Logically, the consistency is vital for the proper implementation of this method family. Still, the way the creation process wrapper functions are written, with precisely defined return types, should notify the user about possible errors, (for instance defining the `InstVirtual_MOS()` to return `CirVirtual_Res*`) already in the compilation time.

After specifying this function family we will provide the function implementation for another generic part of the VFV. In Section 5.9, we have isolated and separately defined a templated function to handle the refinement step of the embossing process. This part detaches all elements of the current MFDP content that are considered environment of the structure in focus. For the pattern matching application, the implementation of the templated function will use a part of the interface of the application domain specific side. The function would simply run through each of the hash containers and keep only the mappings of the elements that are marked as "matched". This means that the algorithm has paired them with a corresponding device in the pattern which is being matched. The implementation of the algorithm is simple. We have three loops that iterate over all pin, device and node mappings. The algorithm tests the function: `isMatched()` and if the element is not matched, mapping is removed from the hash, while the virtual copy gets deleted!

Having defined a specific builder class, we have prepared the view for the execution of the incremental pattern matching algorithm. Therefore, upon the VFV creation, we construct the object of the `Virtual_Netlist` class with the instantiation of the class `Cir_VirtualBuilder`.

6.3 Adaptations of the flat algorithm

The main concepts of the incremental pattern matching algorithm work smoothly with the described hybrid view. Still, in order to minimize the execution time of the algorithm, we want to apply the greedy algorithm that has been written to optimize the execution of classify for flat pattern matching. This algorithm has enabled the intelligent path choosing technique for the pattern matching algorithm that is driven by the templated rules, the rules that incorporate the concept of optional ports. In order to achieve that, the best path first algorithm, (section 2.6*)* has used specific global quantities (the number of neighbouring elements for each net). Therefore, the path that appears to be the best, following the net with the least number of neighbours is chosen. In a flat netlist, it is trivial to acquire the quantities about the number of neighbouring elements of a net. This kind of information is then built while forming the flat netlist. Such handy quantities are unfortunately not possible to have in a hierarchical netlist. In the case of hierarchical netlists, we have port nodes that are connected up the hierarchy. Instances of a single variant are connected to different topologies with different number of neighbouring devices.

As an alternative to the flat netlist statistics we have developed an algorithm that labels the nodes in the hierarchical netlist according to their hierarchical properties. We create a factor that determines the node deepness, or, to how many hierarchical levels is the node distributed. In addition we specify also the wideness of the node, meaning to how many instances a node gets connected in a single hierarchical level. This technique would enable us to favour "shallow" hierarchical nodes, those that have as local connectivity as possible. It is natural that such nodes don't have some dramatic number of devices connected to them which would cause the linear search complexity to dominate in the algorithm complexity bringing extensive runtimes. To depict the extremes: on one hand we have a node with deepness 0, this node would be a local node that has just a couple of neighbouring devices connected to it. On the other hand we have the supply node which leads to every subcircuit of the design! An alternative to the best order of execution algorithm is applying the concept of stop nets. We can assign that the nets of a certain type (supply nets, reference voltage nets) are the places where the recursive search stops. Stopping the recursion at stop nets does not handicap the algorithm execution, as no pattern is connected exclusively through supply nets.

The statistics about the nodes are collected in global walks over all variants. First the bottom-up walk over all variants is performed. In this process we perform the iteration over all instances that are referenced in the given variant and than iterate over all pins of the instance in focus. Once the proper pin of the instance is picked, we acquire the handle to the nodes it pairs, hence the node in the domain of the variant which is in focus and the port node inside the definition of the instance. As we are performing the bottom-up walk over the `TopDownVariants` vector, we are sure that the lower node already was processed. Therefore, we perform a check and pass the integer label that is built by the following code to the higher node.

```
if ( nodeDeepnessVector->at(upper_node) < (nodeDeepnessVector->at(lower_node) + 1
                                          + upper_node->instPins()->size()) )
    nodeDeepnessVector->at(upper_node) =
    nodeDeepnessVector->at(lower_node) + 1 + upper_node->instPins()->size();
```

The values are stored in a separate vector for each variant of the hierarchical netlist. Each vector's size corresponds to the count of all nodes in that variant.

After the described step we have the appropriate label for all the different nodes in the variant graph on the root level for each of the hierarchical nodes. Now we pass the acquired values in another similar walk, this time iterating top-down over the variants. Now we know that each higher node was already processed and we pass its label to the lower node.

```
if (nodeDeepnessVector->at(lower_node) < nodeDeepnessVector->at(upper_node))
    nodeDeepnessVector->at(lower_node) = nodeDeepnessVector->at(upper_node);
```

For the proper assignment of the hierarchical node labels we use standard API of NLDB developed for the variant concept.

```
defineNodeDeepness dnd;
ForAllVariantsBottomUp(cellIt, dnd);
passNodeDeepness pnd;
ForAllVariantsTopDown(cellIt, pnd);
```

Classes `defineNodeDeepness` and `passNodeDeepness` are defined as function objects, having an operator function compatible with the global iterating functions `ForAllVariantsBottomUp()` and `ForAllVariantsTopDown()`.

In this way we have achieved to define the alternative strategy to drive the BPF algorithm.

6.4 Hierarchical result reports

In chapter 2, we have already stated that the output of the classify tool is a specific protocol file that lists the contexts which satisfy the specific conditions. That is actually the file that stores the results of the matching process.

The protocol file was developed for the purposes of ERC where it was necessary to point to the specific device that was isolated by the particular algorithmic check. Each occurrence of the protocol error in that case was a single device. The protocol file would be parsed by a specially devised algorithm that than marks the errors directly in the Composer® IDE. Therefore the standard syntax of the file was as following:

<global summary >
<error report 1>
<error report 2>

…

The global summary gives a review of the checks specifying the number of matches per rule or the overall flat and hierarchical number of matches.

The syntax of the each error report consists conceptually of:

```
<paths>
<device(s)>
```

For each hierarchical occurrence of the faulty device the list of paths would be given. The paths get then analyzed by the IDE in order to mark the proper instances along the path, in the end pointing directly to the isolated device which was the target of the check.

The protocol file is written in general to support the hierarchical pointing to the specific devices in the design. Therefore this file marks “errors” that consist of two parts. First, we have the path, followed by the devices which represent the objects that are isolated by the corresponding tool’s algorithm.

The flat classify reports syntax was following the above defined structure, still as no hierarchy was present in the input files, the paths in the reports were hard coded with the statement “--Root Level--“. Thus, all results that are specified reside in the top level of the electronic circuit. Hierarchical classify offers isolation of errors directly in the subcircuit where they are defined. For this reason we have upgraded the algorithms that generate the protocol files of the original classify in order to support the path generation. As the devices of the given pattern can be further distributed through the hierarchy (deeper than the relative top level), we have introduced

Flat:

```
================================
Classify  - Netlist Checks
================================
Summary of errors:
2 violations - NAND
Total number of error classes: 1
Total number of parameter errors: 2
================================
Error 1
Title: Find All Inverters
-----------------------------------------------------
Path: -- Root Level --
-----------------------------------------------------
Device(s):
m/x1/x2/mN0
m/x1/x1/mP0
================================
Error 2
Title: Find All Inverters
-----------------------------------------------------
Path: -- Root Level --
-----------------------------------------------------
Device(s):
m/x2/x2/mN0
m/x2/x1/mP0
================================
```

Hierarchical:

```
====================================
HClassify  - Netlist Checks
====================================
Summary of errors:
2 violations - NAND
Total number of error classes: 1
Total number of parameter errors: 2
Total number of hierarchical parameter errors: 1
====================================
Error 1
Title: Find All Inverters
---------------------------------------------------------
 path: X1 [A ] ( X2)
---------------------------------------------------------
Device(s):
X2/mN0
X1/mP0
====================================
```

Figure 6.4-1 – Flat and hierarchical error protocols. The example shows the output of the inverter search process.

a specific *relative path*. This path is written directly in front of the device's name in the reported devices list.

The above described protocol file can be illustrated by the example given in Figure 6.4-1. In this example we have reported the occurrences of inverters that appear in the hierarchical design in Figure 2.3-3. The flat version of the report (that assumes the flattened input netlist prior to algorithm execution) reports two errors. On the other side, the hierarchical version of the report registers one hierarchical occurrence of the inverter, but specifies the multiple paths of its instantiation. Note that the path which is given in the example is also "condensed". Condensing paths means that the common parts of two different paths are grouped together while the different instantiation paths of the given hierarchical level are listed in brackets. The example in the Figure 6.4-1 is simple; it contains two paths of one hierarchical level that are combined. We read the path in the example as: "in instance X1 of the cell A and also in instance X2". In the example we have just one hierarchical level, if the hierarchy is deeper, we can get very compact paths and consequently shorter, more readable output file.

The hierarchical error protocol reports aggregate non-redundant results. This is very important in order to suppress the time needed for their analysis. In the trivial example that we have specified, the counts of the reported errors in the flat and the hierarchical results differ by the factor of 2. In realistic examples, as we will show in the next section through our experiments, this factor is two orders of magnitude in average. This achievement clearly points out the benefits of the hierarchical algorithm and is one of the key results of our research project.

In order to compare two versions of the protocol file, thus to prove the functional equivalence between the flat and the hierarchical algorithm, we have written a specific Perl script. The script "flattens" the hierarchical report by connecting the paths of the reported devices of each error with their names (combined with relative paths) as a prefix. Thus, in the example, from the single error report with two paths, we would get two pairs of CMOS transistors. Thus, we obtain the redundant flat list of errors that is comparable to the originally flat error report.

6.5 Example of the matching process by incremental hierarchical structural pattern matching

Once we have prepared the VFV and connected it to the pattern matching tool classify, we can perform incremental structural pattern matching directly on hierarchical designs. In order to illustrate this process we include a simple matching example where we match all latches in a given hierarchical design in two incremental steps. First we will match all inverters and than all latches that are instantiated in the given design. The example that we give in ***Appendix C*** emphasises the background actions of VFV during the matching process.

6.6 Case study

In order to provide evidence of the functional correctness, qualitative benefits and to measure the typical runtime and memory consumption of the application of the VFV on an search oriented incremental pattern matching algorithm, we have employed a number of tests.

The runtime performance tests were computed on a machine with AMD Opteron processor at 2.6 GHz and 64 GB operating memory, running Linux RHE 4. Throughout tests, the memory consumption was measured externally, by evaluating the relevant process size using the third party tool - *massif*, from the *valgrind* package [57]. The authors of the tool claim the precision within 1% for the obtained results (the peak memory usage of the process).

We have tested two flat classify versions and four hierarchical classify versions. They are result of the algorithm evolution, depicted in Figure 6.6-1.The program version named *c42* is the initial version of the incremental pattern matching tool classify that sequentially picks the terminals of each device while proceeding into the DFS. The version that implements the enhancement where the flat algorithm picks the best path, by the greedy approach is denoted as *c44*.

Further, we have the default hierarchical classify version is *c52*, which is the *c42* flat algorithm ran on VFV. The version *c54* is the flat algorithm version *c44* ran on VFV. In addition a specific optimisation (fingerprint verification) of the iteration directly inside the VFV in order to test how this fact influences the execution of the hierarchical pattern matching is implemented. This version is denoted as *c52f* and *c54f*, when applied on *c42* flat algorithm and *c44* flat algorithm, respectively. The fingerprint verification principle is given in (Appendix B). Note that the implementation of this principle is just preliminary and approximate and that it, in some cases, misses the matches of highly distributed contexts. This can be eliminated by further work on the implementation of the fingerprint verification principle. Thus, we include this version of the algorithm just to prove if the potential benefits it brings are worth the time to implement such an enhancement in order to make the specific application of the VFV on structural pattern matching more robust.

Throughout the experiments, the behaviour of the mentioned program versions against five different rule sets was analysed. Four rule sets were written in order to recognize the elementary circuit elements:

- All inverters
- All flipflops (as a single, flat rule)
- All inverters and than all flipflops
- All NANDs.

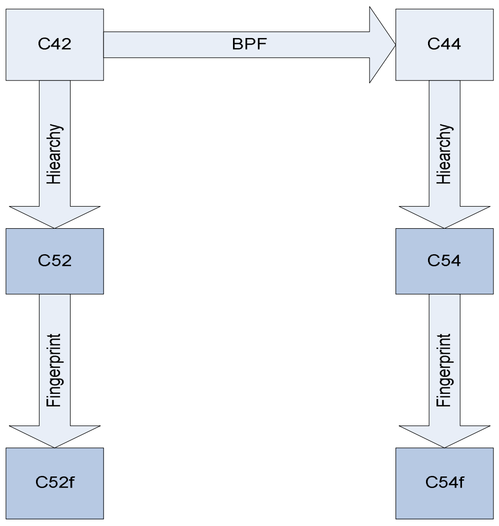


Figure 6.6-1 – Structural pattern matching tool – Classify - algorithm evolution and available versions.

The rules have in common the net type predefinitions, where all standard names for the supply and constant voltage nets are marked. This part is followed by the individual definitions of the block rules(s) to recognise above given simple contexts. The contexts that were recognised are then reported in the protocol file without any parameter evaluation. The evaluation of these examples can clearly show how various recognition rules behave and contribute to the global runtimes and memory consumptions of the more complex programs written for classify. This will be discussed together with the results we have obtained.

As an additional rule set we have included the test that isolates the realistic industrial contexts needed in order to

- detect the slow nodes, that are driven with weak drivers (loadcheck).

Loadcheck is a typical industrial check that is also given as the example of structural pattern matching application in VLSI, in the first chapter.

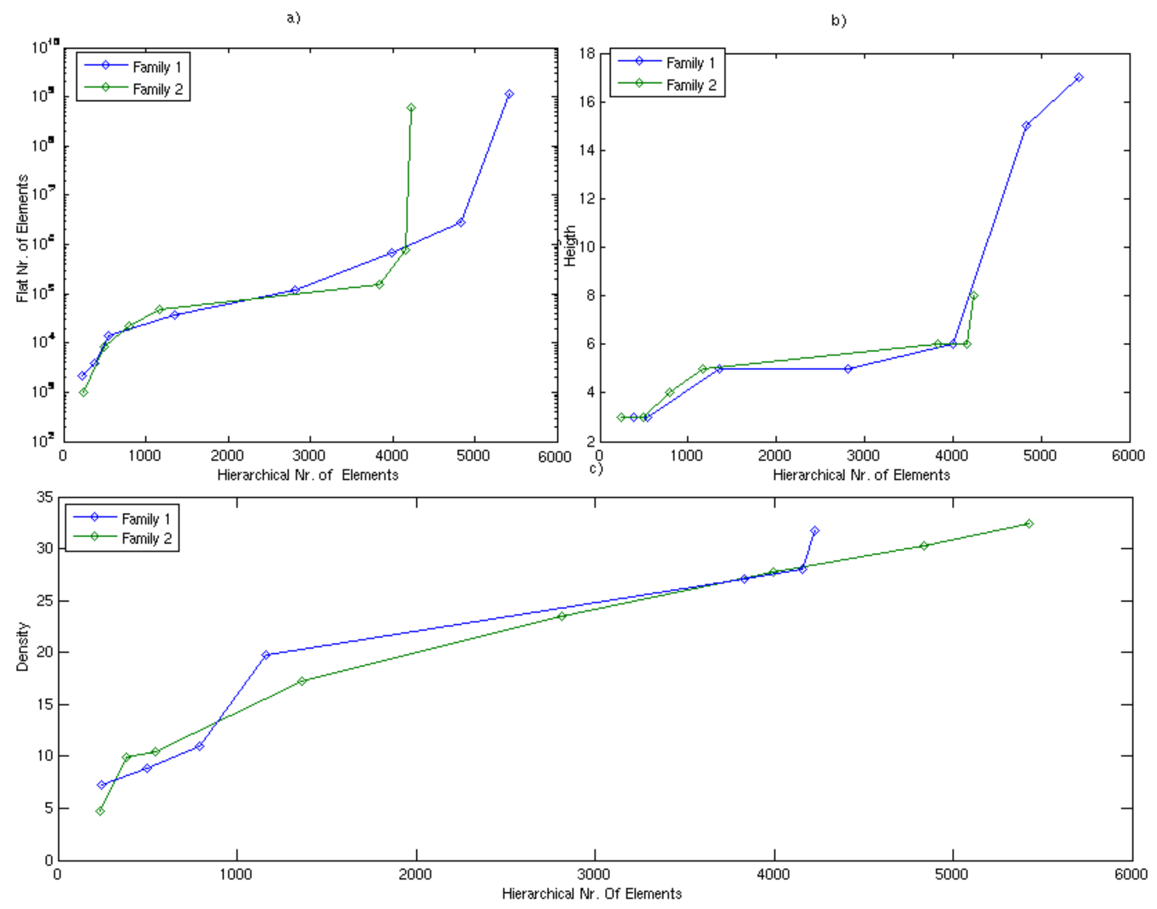


Figure 6.6-2 – Quantities of the example hierarchical netlist families. a) Semilog graph that shows the relation between the flat and the relevant hierarchical element count. b) Graph that depicts the measured hierarchical design height. c) Hierarchical design density for the example families of netlists.

These five rule sets were run against two families of hierarchical netlists. The family of the given hierarchical netlist is obtained by gradually increasing the size of the netlist. This is achieved by allowing that the previous example netlist is one subcircuits instantiated in the context of the next example netlist in the given family. In other words, the successive netlists of a given family are always contained one into another. In this way we can get the opportunity to observe the scaling of different algorithm parameters that we have measured across the developing size of the hierarchical netlist. As a measure of complexity of the netlist we take the equivalent flat netlist's number of elements, or alternatively the number of atomic elements in the hierarchical netlist. Of course, as the connectivity of the elements is important and the hierarchies have different quantities we can not guarantee perfect scaling. In order to depict the quality of scaling the relation between the hierarchical and the flat element count, the height and the density of the example hierarchical netlists of both families are given in Figure 6.6-2. The figure shows constant increase of both parameter values. The two families are generated out of two realistic industrial hierarchical netlist examples. The example designs represent DRAM memories and thus contain a highly redundant and enormous in size array circuit. The array contains the memory cells. This subricruit of the hierarchical design is contained as the last example netlist of each of the families. As we can see in the figure, the growth of the flat element count compared to the hierarchical element count of both example netlist is very steep. Note that the scale of the graph is semi-logarithmic, having the logarithmic axis that quantifies the number of flat elements of the given netlist.

With respect to the flat netlist size, we have two domains of hierarchical netlists, the lower and the upper domain. In the lower domain the flattening of the netlist and application of the flat algorithm is still possible, using typical available computer resources. The border between these two domains is than flexible and depends on the hardware that is used to execute the algorithm. In the lower domain we have compared the execution behaviour of two algorithm strategies (flat and hierarchical). The measurements were done with flat netlists that contain up to 2 500 000 elements. In the higher (exclusively hierarchical domain) we have performed tests on examples that have up to 1 000 000 000 (one billion) (flat) elements. These example netlists had also a very high gain factor, eg. the one billion element example netlist consisted of just 5000 atomic elements. This is due to the already mentioned fact that the available big examples include a non-specific highly redundant *DRAM array* subcircuit. This part of the design topology has also non standard interconnections. Hence, a single transistor, the member of the memory cell is connected with millions of other similar transistors that belong to different memory cells, members of the array. The setup in order to successfully process this part of the memory chip demands further work. We have, with the already available setups, managed to run some of the tools/runsets on this highly redundant, non standard netlist.

All throughout the tests we have confirmed the functional correctness. Further, as the hierarchical approach was used, we witness the enormous enhancement in the relation between the hierarchical and flat report counts. This relation is depicted in Figure 6.6-4. The graph gives the distribution of the ration between the number of flat matching reports and the number of hierarchical (condensed) matching reports. The ration is distributed over the flat element count in both netlist families. Graphs overlap several measurements (for nand, inverter, flip/latch and loadcheck rules), as shown in the graph legend. We can see that the ratio between the flat and hierarchical match count constantly increases and reaches, typically, the value of two orders of magnitude, for bigger, realistic in size, example designs.

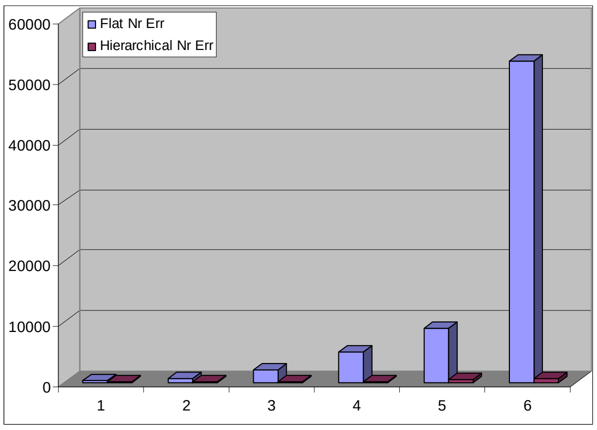


Figure 6.6-3 – Nr. Hier. and flat matches for different hierarchical netlists. The blue bars depict the number of redundant, flat match reports, while in purple the number of corresponding hierarchical matches is given.

In order to illustrate the gain, the linear bar chart gives the growth of the number of flat matches compared to the number of hierarchical matches (Figure 6.6-3). The pairs of bars represent the numbers of flat and hierarchical number of matches, respectively, for 6 different (ordered according their size) hierarchical netlists. This graph clearly illustrates the difference in time needed by the user of the tool (the designer) to evaluate the obtained error protocols. The result which is represented here is in the same time the most revolutionary achievement that the conceptually new hierarchical pattern matching approach allows.

The performance tests are nevertheless also very important as the stable runtime and low expected complexity of the progress of the memory requirement and the time requirement of the hierarchical pattern matching algorithm, allow usage of the hierarchical results in all realistic application cases. For this reason we have thoroughly tested the algorithm potentials from this aspect and proven the positive achievement as well as pointed out the issues that the new algorithm in this early development stage has and that should be addressed in the future.

Let's start with the distribution of the required time for matching of the elementary rules, over the complexity (number of flat elements) of the netlist. All measurements for both netlist families are given together in the graphs in Figure 6.6-5. The

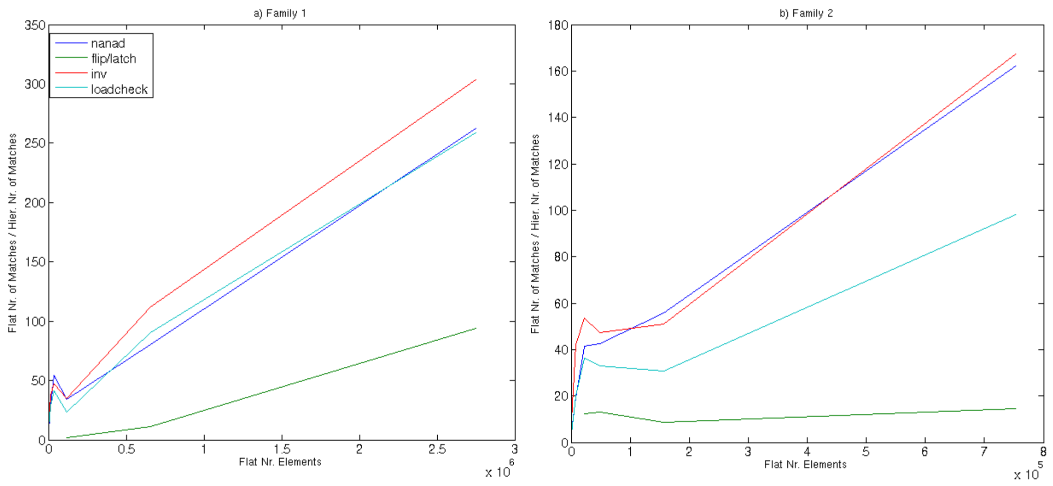


Figure 6.6-4 – Linear distribution of the ratio between the number of flat reports and the hierarchical reports. The graphs give together the obtained results for the nand, flip/latch inv and loadcheck rules.

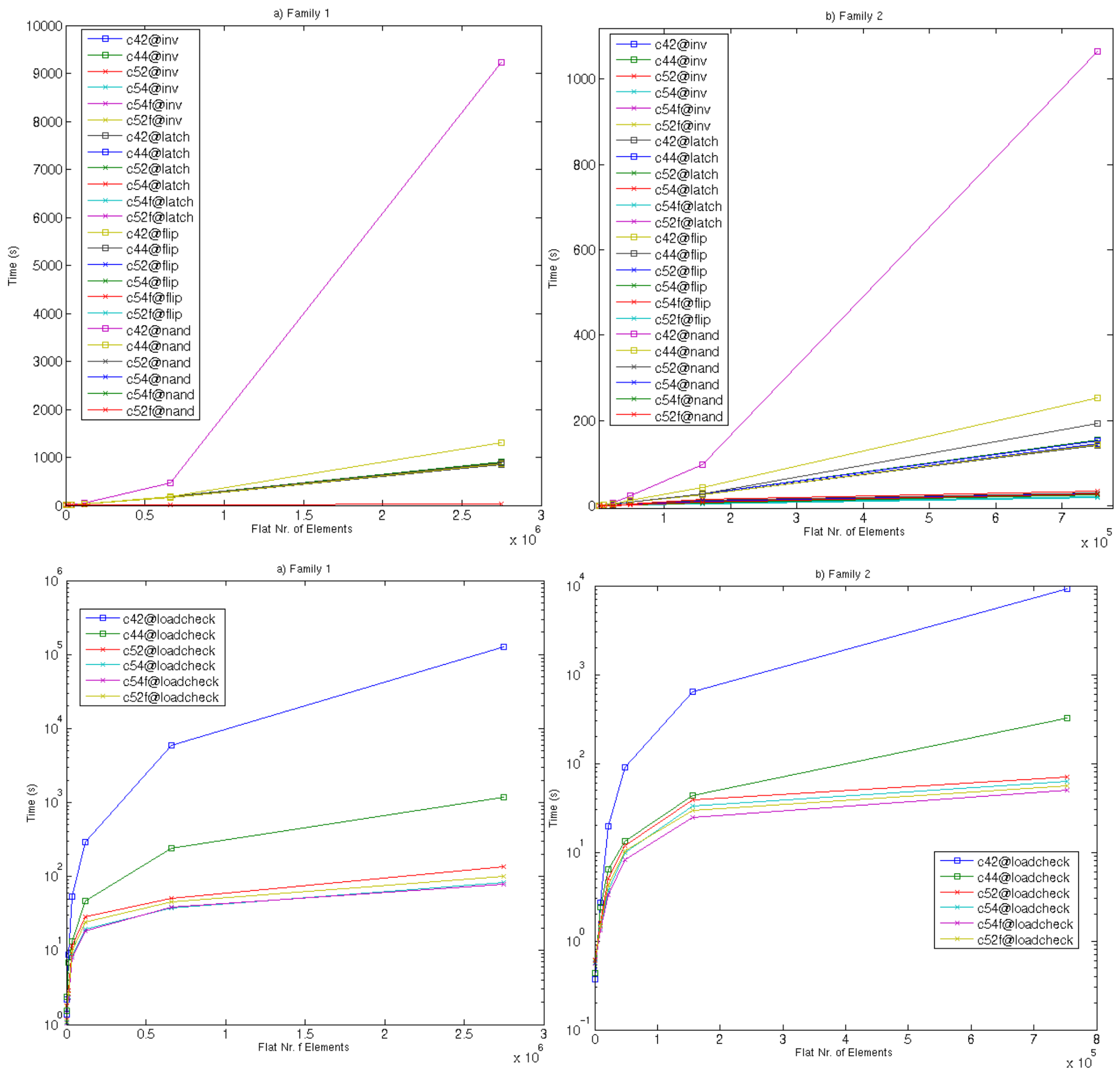


Figure 6.6-5 – time complexity of different algorithm versions. a) performance of elementary rules on Family 1. b) performance of elementary rules on Family 2. c) performance of the loadcheck ruleset on Family 1. d) performance of the loadcheck ruleset on Family 2. Different classify versions are denoted with cxx, where xx is the relevant program version. The elementary rules are: inverter, latch, flip and nand.

tests were performed on 4 different (simple) rules and 6 different program versions, giving 24 curves in the graphs (a) and (b). The curves depicting the measured runtime of the flat algorithm version group in the higher domain of the graph, while the hierarchical runtimes group low, with dramatic differences of the required time to complete the algorithm execution for bigger target netlists.

Apart from these two groups, one can notice that a line which describes the progression of the execution time of the 4.2 algorithm applied on the simple NAND rule shows the super-linear complexity and reaches higher values than typical for other rules of both flat classify versions. This rule is an example of the penalty that the suboptimal 4.2 algorithm version pays for blindly approaching the nets neighbouring the given device which it is analysing, following the nominal order of the device's pins. The time required for the execution of the given rule by algorithm 4.4 is together with all other tested rules, inside the upper cluster. Note further that the differences between flat and hierarchical clusters are more drastic for the Family 1 than for the Family 2. This is in correlation with the difference in height of the hierarchies of the

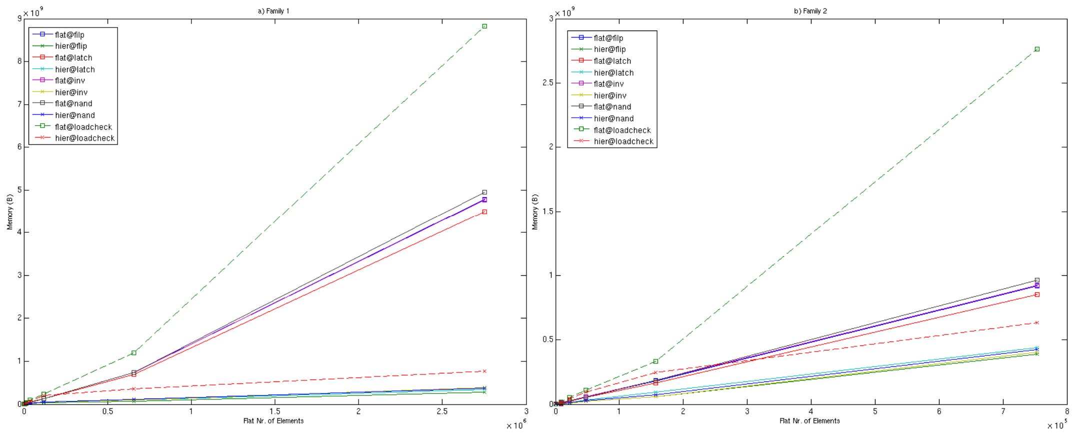


Figure 6.6-6 – Memory Consumption. a) Family 1. b) Family 2. In both families the elementary rule (flip, latch, inv and nand) memory usage measurements are given with solid lines, while the loadcheck memory consumption is depicted by the dotted line. The flat algorithms are marked with a square, while the hierarchical runs are marked with an x.

Family 1 and Family 2. Family 1 has higher hierarchies. This difference in values is noticeable also for the measurements of the memory usage of the algorithm run against two netlist families.

Once many rules were combined in the complex loadcheck program, the relevant numbers got consequently larger, but the differences between the typical algorithm runtimes became even clearer. This is given on two semi-logarithmic graphs that combine the execution times of different classify program versions for both families (Figure 6.6-5.c,d). In this case we have the (trivial) runtimes of the hierarchal algorithm grouping (for the first family) around the value of ~50 seconds for the netlists up to 2.5 millions of elements, the enhanced flat classify algorithm at 10^3 and the measured runtime of the 4.2. classify version up to $1.2*10^5$ (around one and the half days). The corresponding differences are present also in the case of Family 2, just with smaller gaps. To conclude, the measured difference between the execution times is by two orders of magnitudes compared to the enhanced flat algorithm, further, the difference between the enhanced flat algorithm and the initial version is another two orders of magnitude. Note that in this domain the differences between the hierarchical algorithm versions are not drastic. We can explain that by the fact that the hierarchal algorithm has far lower number of attempts to match the given context, it pays also the lower price for each false match.

Having in mind stated above, we have shown that the enhanced flat version allows the stable application of the flat algorithm in the domain where the flattening is possible. The progress of this algorithm version is then linear, the typical complexity claimed by different authors of algorithms in the domain of flat structural pattern matching. The version 4.2 unfortunately shows an indeterministic complexity. The success of the enhancement of the flat algorithm was also proven during the up today more than two years of professional industrial application. The values measured for the hierarchical algorithm versions prove the sub-linear complexity, with respect to the flat size of the given hierarchical netlist.

The measurements of the memory consumption were done for the identical tests that were used to measure the gain in the relation between the hierarchical and flat number of reports and the time complexity of the algorithm. The results obtained are similar to the results of the time measurements, with the addition of identical memory consumption behaviour of the two flat algorithm versions. This was expected as the solution for BPF algorithm does not include the significant memory consumption overhead. On the other hand similar statement is valid for the hierarchical algorithm if one assumes that the size of the materialised flat data portion is neglectable. This is true as the patterns which are being searched for have just couple of elements each.

For given reasons, we have measured the memory consumption on the representative algorithms of both groups, on versions c44 to get the typical memory consumption of the flat pattern matching and c54 in order to get the results for the hierarchical algorithm (Figure 6.6-6).

The measurements have shown that the typical flat approach memory consumption grows linearly, while the hierarchical memory consumption shows once again the sublinear complexity. In the case of the more complex rules the memory consumption is higher for both flat and hierarchical versions, as it is necessary to represent all relevant contexts that were recognised with the corresponding objects.

Let’s now analyse the performance of the algorithm in the higher domain, where no flattening is possible. In this domain we have compared the runtimes of the four different hierarchical versions, to investigate the benefits which the different hierarchical approach enhancements bring. We have tested these algorithm versions against two example netlists which include the DRAM array. These netlists were the full-chip netlists of two example families containing roughly one half of the billion and one billion flat elements for the first and the second family, respectively.

The results we have obtained for measuring the time requirements for this algorithm are given in Figure 6.6-7.

The bars show the runtimes for the latch, flip and the inverter check, respectively. These are the rulesets for which we have managed to obtain the results against the hierarchical netlist that includes the non-standard array. From this graph we can conclude that the runtimes are the most stable and optimal for the algorithm version c54f.

The overall tests have shown that the concept where all the problems that the hierarchical data brings are solved inside the database (the way the data is presented to the algorithm) is feasible. The adaptation to the needs pattern matching algorithm was further easy due to the flexibility of the view architecture. The feasibility and thus functional correctness of the results gained by the application of the VFV to the flat pattern matching algorithm has been strengthened with the in average two orders of

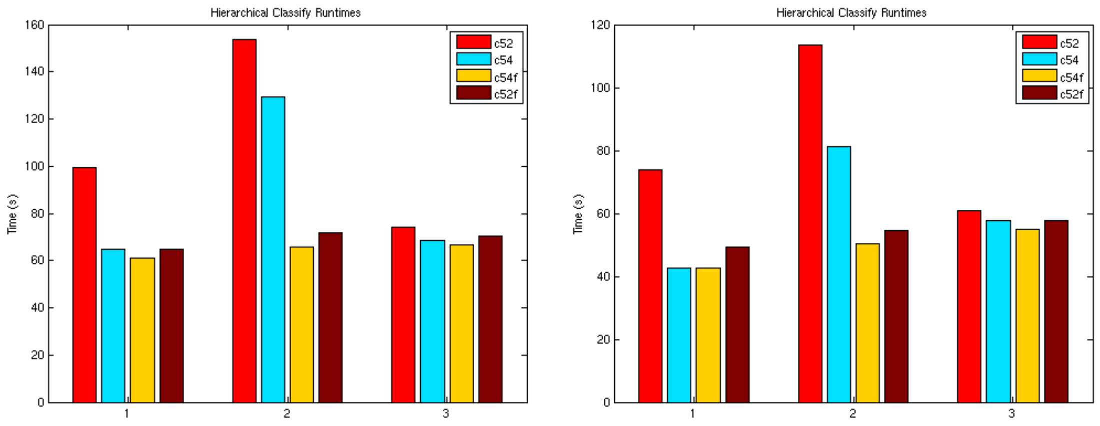


Figure 6.6-7 – Runtime requirements for the examples in the domain where no flattening is possible. 1 – latch rule, 2 – flip rule, 3 – inv rule.

magnitude compression factor for the error reports. Further, the tests have shown that the potentials of the hierarchical algorithm bring the sublinear complexity in memory requirements and required time (sublinear to the number of elements in the equivalent flat neltist). Nevertheless the adaptations of the VFV are required in order to be able to run it in any application case and to get more robust algorithm for the domains of big hierarchical (unflattenable) netlists. In this domain we have, in some sense, the repeating of the history. The c42 flat algorithm had a non-optimal order of progressing into recursion. This was fixed by the version c44 of the flat algorithm. This version is still heuristic and there is no theoretical proof that it will work in any case. The long-term, stable industrial application has solicited the good quality of this approach. We have, in further work, to enhance the optimality of the ordering the pins of elements for the hierarchical algorithm and in despite of the additional complexity in acquiring the quantities weight the different paths properly. However the clear idea and the strategy to fight this problem is ready and should be implemented through the short term research to allow the smooth application of the powerful algorithm on application cases that include atypical highly redundant DRAM array subcircuit.

To conclude, the performed experiments show high potentials of the chosen solution recommending it as a common solution for hierarchical structural pattern matching. Its flexibility and easy adaptability (with possible minimal changes of the view strategy) allows that other existing or future flat algorithms that analyse schematic designs can take benefits of the hierarchical data representation.

7 Conclusion

We have completed the investigation of the problem of structural pattern matching in the area of static verification of VLSI hierarchical designs. A solution has been found for structural pattern matching in hierarchical designs. It can be also applied to other various similar algorithms that need transfer from the flat domain to the hierarchical.

Our contribution includes establishing the methodology of so-called layered views on the hierarchical schematic data. We have identified the standard view architecture that enables polymorphic views on the hierarchical organisation of the given data model. The standard architecture is specified using advanced object-oriented concepts. We have further defined the novel Virtually Flattened View (VFV) using the proposed standard architecture. VFV presents the data of the hierarchical design locally flat. The highlighted, flat data portion can be formed orthogonal to the design hierarchy. Additionally, in order to make the view application easier and more powerful we have developed a technique that enables embossing of the flat data portion that has been created into the primary design hierarchy. This operation affects the design hierarchy and commits the given data portion as a separate subcircuit. The committing technique was designed to enable very quick changes. The complexity of the committing technique is thus tied to the size of the flat data portion and not the size of the cell that is being altered. VFV development included isolation of some specific data structures that enable the proper functioning of the view. It also included the creation of a set of very complex interrelated algorithms on those data structures. VFV architecture is generic and allows flexible upgrades of the view entities to meet specific user algorithm requirements. Hence, entities of the view that model given database elements can include application-specific augmentations of the interface.

In order to provide the evidence of the feasibility of this concept and to achieve the needed hierarchical structural pattern matching method, we have applied the developed VFV on an existing project that implements the incremental pattern matching principle on flat netlists. With the aim of accomplishing this we have used the flexible view design, adapting its entities to the application domain. The flat algorithm could be used with just minor changes. Changes included very local adaptations of the flat algorithm to allow it to handle the new principles, which a fact that the matching is performed on the cell definitions of the hierarchical database brings. The enhancement that is introduced is specified as an upgrade of the matching results report protocol. We have introduced a hierarchical report protocol, where each match is tied to a specific cell of the hierarchical design. This new concept allows non-redundant match reports.

The realistic tests which have been executed on industrial examples have confirmed the functional correctness of the method. Tests have allowed us to properly quantify the hierarchical pattern matching report protocol, which is conceptually new. We have concluded that this new report type allows the improvement of the effectiveness of the algorithm by an average of two orders of magnitude. This means that, by taking use of the hierarchy one can now extract precisely the wanted topologies that are instead of being related to their instantiation contexts, now related to their non-redundant definition contexts, dramatically suppressing needed effort (man power) to analyse the reports.

Performance tests of the algorithm have shown that, as designs increase in complexity, the growth of time and memory required is sub-linear. The new algorithm is several orders of magnitude superior to the flat algorithm already for sample designs with > 80 000 flat transistor count.. We have further shown that the algorithm is now capable of processing target designs that cannot be flattened (using current typical computer resources), those having more than a billion flat elements.

In this domain we have organized the roadmap of enhancement of the set of sophisticated concepts in order to achieve the optimal runtimes. Future work thus is related to exploring the identified possible improvements that the hierarchical data model can bring, as well as fine-tuning and adapting the specific hierarchical design personalisation concepts (the concept of cell variants) to the VFV. In this way we can exploit the new methodology optimally.

The overall results provide a strong recommendation that the described approach can be used as a standard for addressing the problems that hierarchical organisation brings.

APPENDIX

Apendix A (Personalisation by variants)

Commercial EDA databases are in most of the cases implementing the data model of the standard SPICE format. For this reason they should support the concept of cell parameters, also present in the SPICE format. This concept is one of the crutial reasons for the personalization. The concept of parameters leaves a part of the data of the cell definition templated, to be resolved later, in the context of its instantiation. In this way we can have cells with templated transistor widths, lengths or some other device parameters. This concept can be illustrated by the example in Figure A-1. In the example a hierarchical netlist, which apostrophes different concepts of flavouring the instances of the given cells is shown. Note that for clarity only this aspect was taken into account and that the example has no electrical sense. As it is shown, by different parameter defining techniques, instances of the cell A have different parameter values. For example, the transistor in the instance I1 of cell A has the width of 2 and the length of 2, on the other hand instance I2 of the same cell has the values w = 1 and l = 1 for the transistor. Further, as a part of the specific algorithm, nodes (nets) of the hierarchical netlist can be flavoured by type that describes their semantics. This concept is widely used in different applications through the technique of signal propagation. This is another property that can flavour an instantiation of the given cell.

Additionally, cells are characterised by the topology of its pins. Hence, a cell can have its pins shorted somewhere up in the hierarchy. The consequence of this is that two nets, cell pins that are connecting the cell with it's environment, have to be merged (seen as a single net). This produces the topology which is slightly different than one which is given by the cell, depending only on the instantiation context.

In order to suppress redundancy, and still provide user application with resolved templated data of the cell, in place, we introduce the concept of variants to group all instances with identical templated data that is interesting for the specified application.

We create variants by grouping instances of the same definition by mentioned criteria. Therefore, according to the parameter value we can subdivide the given six instances of our example, which share the definition, in two groups. First group of instances has the definition which has parameters resolved with values w = 2 and l = 2 (for instances I1, I3 and I5), and second group which has parameter values w=1 and l =1 (for instances I2, I4 and I6).

Without propagating node types over devices, we can again group different instantiations of the cell A according to the node type. Therefore the grouping according to this criteria connects instances I1 and I3, as they have nets net1 and net2 without the type and grounded net3, further, instances I2 I4 and I6, which have net1 on vint - supply net type, net2 without type and grounded net3 and in the end, instance I5 is in a different variant, as net 3 is at vnwl here. net1 and net2 again have no node type.

A third classification criterion is grouping according to cell pin topology. In the given example we have a group with I1 and I3: net1 and net2 are connected and another group of instances I2, I4, I5, and I6 where no pins are connected directly. In total two variants according to this criteria alone.

In addition, it has shown up useful to create variants of the given cell by grouping instances according to their instantiation position. More precisely, according to the parent cell in which they are directly or indirectly instantiated. This can be useful in order to enable the user application to "concentrate" on the given block of the design

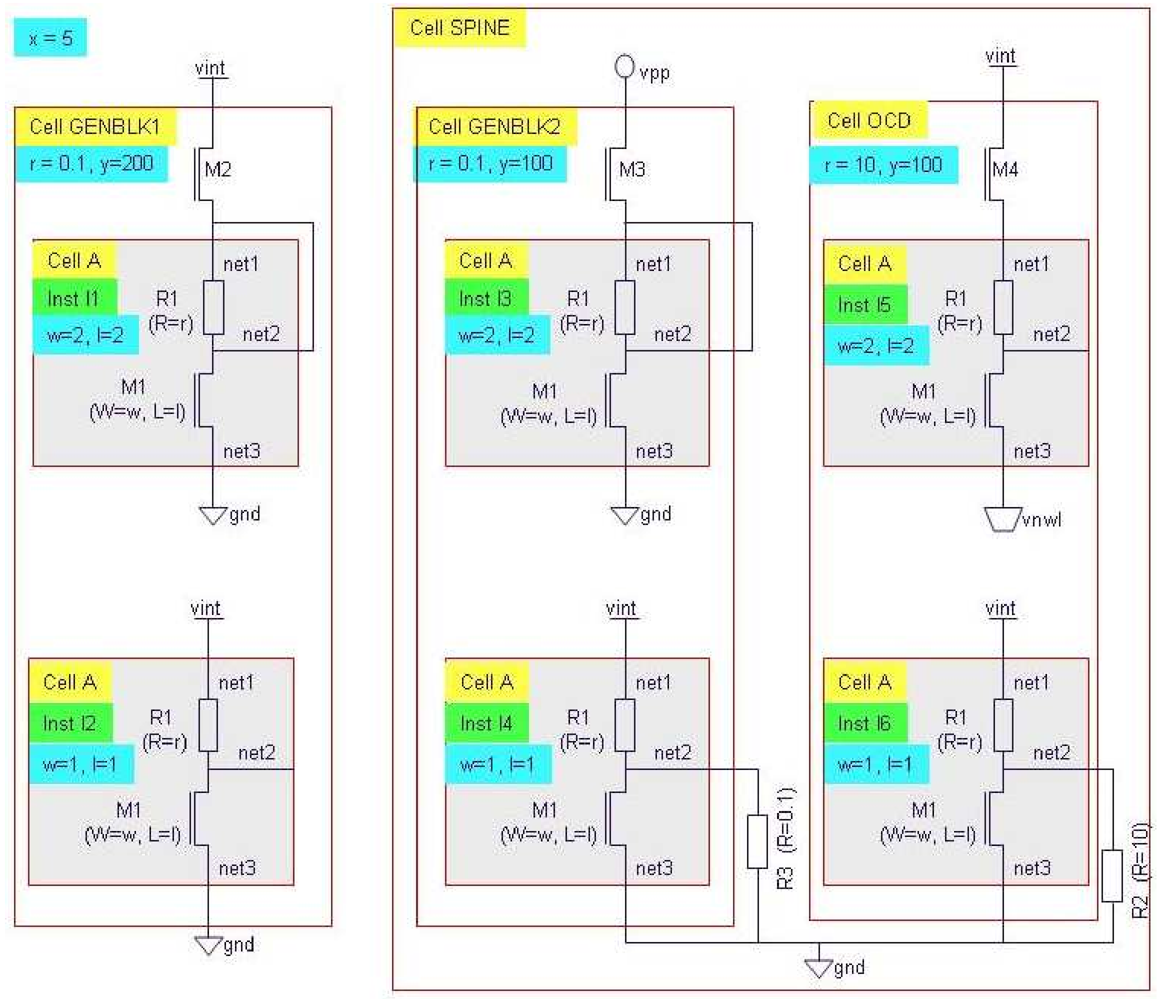


Figure A-1 – Variant criteria

and perform specific algorithm only inside (or outside) it. Note that if we put each instance which has different parent cells to a separate variant, we might end up with a huge number of variants. Therefore, we build the variants based on "parent cell conditions" to be defined and carefully used by the application. In the example, we can, for instance, distinguish between devices inside or outside of the cell GENBLK* and additionally inside or outside SPINE cell. This rule would gives three additional variants:

I1 and I2: In GENBLK* but not in SPINE, I3 and I4: In GENBLK* and in SPINE and in the end I5 and I6: Not in GENBLK* but in SPINE.

Note that we can now group instances which share all of defined criteria in the same time. We achieve this by mutual intersections of all sets of instances, which represent variants according to the single grouping criterion.

To illustrate this we will refer to the example in Figure A-1 once again. If we need parameters, node types, and pin topology but might omit parent cell conditions, we have the variants:

- Variant V1: I1 and I3
  - r=0.1, w=2, l=2
  - net1 and net2 without nodetype, net3 with nodetype gnd
  - pin topology net1 - net2=net1 - net3
- Variant V2: I2 and I4
  - r=0.1, w=1, l=1

  - net1 with nodetype vint, net2 without nodetype, net3 with nodetype gnd
  - pin topology net1 - net2 - net3
- Variant V3: I5
  - r=10, w=2, l=2
  - net1 and net2 without nodetype, net3 with nodetype vnwl
  - pin topology net1 - net2 - net3
- Variant V4: I6
  - r=10, w=1, l=1
  - net1 with pintype vint, net2 without nodetype, net3 with nodetype gnd
  - pin topology net1 - net2 - net3

In the end, it is possible to define any additional, algorithm specific criteria for creating groups of instances of the given cell.

Variants are created in several global, self-altering, hierarchical walks over the TopDownVariants structure to build the **variant graph**. The walk over TopDownVaraints vector is at the beginning identical to the walk over TopDownCells vector, further, in the process of creating variants, duplicates of each cell according to defined criteria are inserted, altering the starting initial structure.

Storing of Variants, interface to acquire templated cell data

Variants are stored in the database that was developed for the industrial application of Qimonda AG with the explicit interface. They are “visible”. Therefore each application that employs the benefits of the variant concept has to explicitly control and achieve templated values using specific interface. This was not a must. Some other alternative implementation, whose vision we have shortly pointed out in Chapter 4 can hide them and perform the regrouping of the instances into several subgroups for certain primary definition. The substantial difference between the variant of the cell and the cell appears once the variant adds some functionality. For instance our database does not, by default allow node types, they are completely introduced, together with the appropriate interface in the variant classes. Nevertheless, as the question of the standard interface is relative, we assume in this thesis that the variant and the cell are actually equivalent terms. More precisely, the term variant just explains the way a given group of instances is obtained. Thus, it is more relevant to the way a given cell is implemented.

Coming back to the way variants are implemented in our industrial database: each Base_Cell has a list of variants associated with it, Figure A-2. Initially this list is void, while it gets populated during the variant creation process.

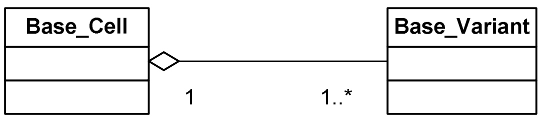


Figure A-2 - Relation between Base_Cell and Base_Variant

In the example of the previous section, we have shown that variants are created according to all combinations of different building criteria. Therefore, in order to represent the data in the most optimal way, each variant object is linked to appropriate set of values, for each criteria type. On the other side, list of criteria value sets is maintained irredundant. This implies the fact that when a new variant is to be entered, the insertion algorithm has to perform the linear search over the list of criteria value sets to determine if the identical set already exists. This linear search hasn't dominated in the runtime of the applications using variants in the industry realm, so far. The described concept is illustrated in Figure A-3. Each variant references a set of criteria value sets. In the illustration, Variant 1 shares parameter Set 1 with the variant 2. If, for instance, variant 1, during the application execution changes some value of the its parameter set, the relevant reference will be relinked to another set, that is either already in the list, or is newly created after the search, while the link between the object of variant 2 and the parameter set one remains.

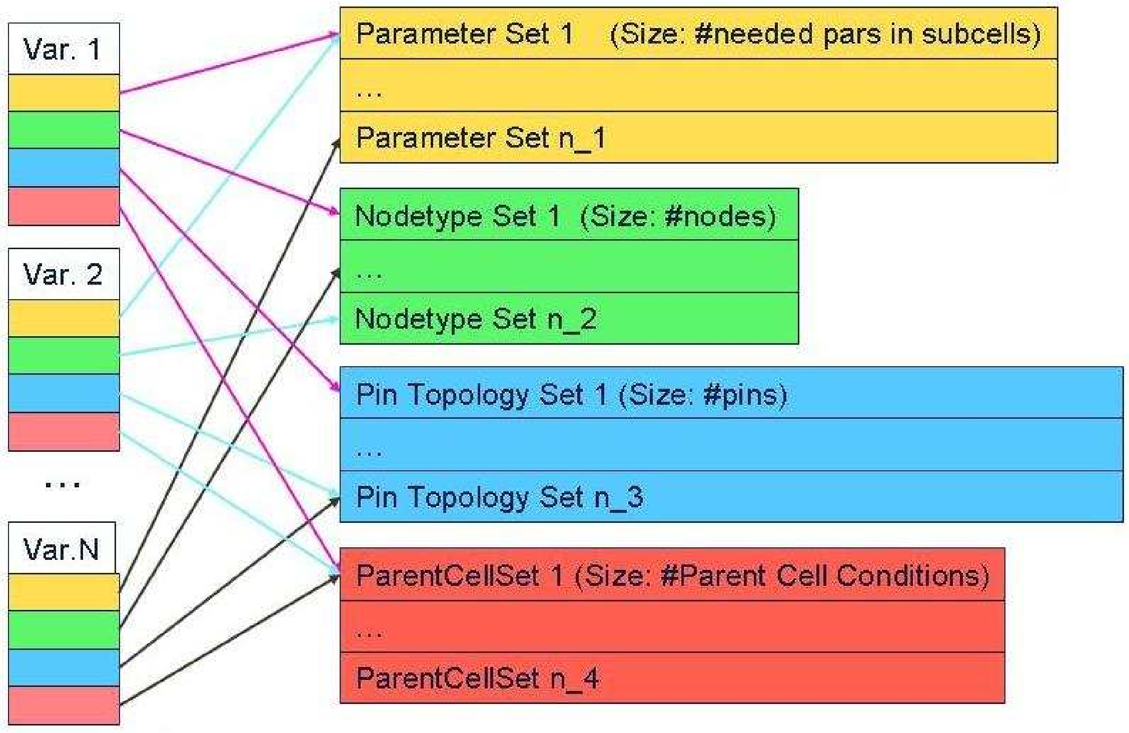

Var. 1
Var. 2
...
Var.N
Parameter Set 1 (Size: #needed pars in subcells)
...
Parameter Set n_1
Nodetype Set 1 (Size: #nodes)
...
Nodetype Set n_2
Pin Topology Set 1 (Size: #pins)
...
Pin Topology Set n_3
ParentCellSet 1 (Size: #Parent Cell Conditions)
...
ParentCellSet n_4

Figure A-3– Variant Criteria storing data structure

Thus, a template class `Base_VariantData` stores different sets of variant specific data in vectors (`template <class T> class Base_VariantDataList : public vector<T *>`).

This kind of architecture allows flexible adding of potential new variant criteria.

Standard NLDB data sets include vectors for:

- parameters as the actual instance parameter values for the parameters which are needed in some arithmetic expression in a cell or its subcells.
- nodeTypes are the types for each single node and also the collection of node types of nodes for each equivalence class (in different vectors).
- pinTopology is the data structure to store the connectivity of pins up the hierarchy. For each group of connected pins, the smallest index of connected pins is stored for all these pins. For example an instance with 5 pins, where pin_0 and pin_4 are connected and pin_1 and pin_2 are connected, would have the pin topology vector 0,1,1,3,0.

- parentCells stores a '1' for a related CellPatternMatch containing a set of patterns for parent cells, if one of the parent cell names matches one of these patterns and a '0' otherwise.

Class `Base_Variant` provides interface methods to access all relevant data stored in `Base_VariantData` object.

Apart from this interface, `Base_Variant` class provides interface and implementation for relations with other variants (father variants and children variants), to form the variant graph.

TopDownVariants

In different application algorithms, together with the variant creation algorithm itself, it is important to access all variants, globally, in a specific order (bottom-up, or top-down), similar to the `TopDownCell` vector. For this reason we create a new container, abstracted in the class `TopDownVariants`, which with it's iterator traverses over all variants of all cells of the design, in a way that, for bottom - up walk, all variants that are instantiated in a variant in focus have already been accessed, during the bottom up walk, and vice versa, for the top-down walk, no variant that is in the current focus has an no so far unvisited parent.
Due to the similarity with `TopDOwnCells` vector, the implementation architecture of the `TopDownVariants` vector inherits classes `TopDownCells` and the appropriate iterator from the class `TopDownCells` and it's iterator, simply adding the additional iteration, over all variants for a given object of the `Base_Cell` class. This design solution is given in the class diagram in Figure A-3.

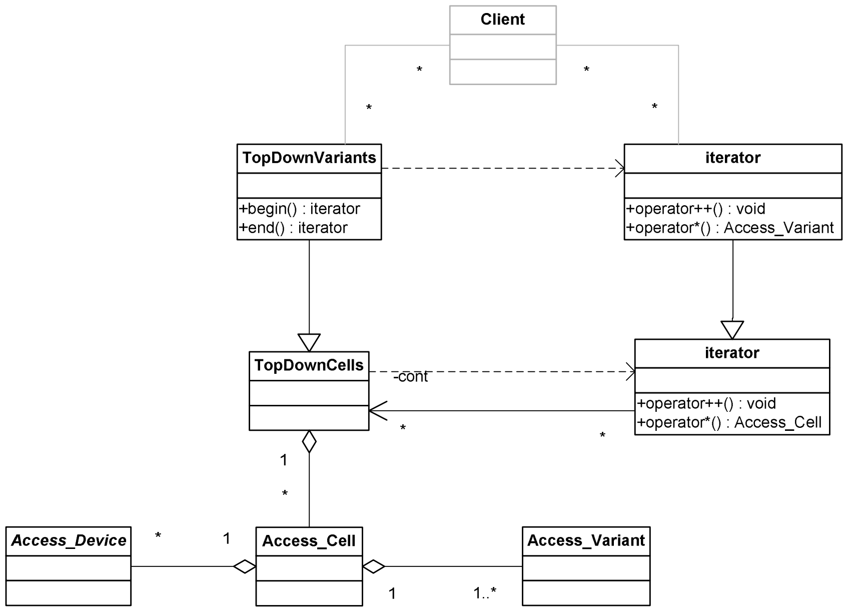


Figure A-3 – TopDownVariants container

Variant Graph

Variant graph is analogue structure to the cell graph. Variant graph offers a structure which is, by the number of elements, somewhere between a definition tree and a instance tree. It could be, more precisely, obtained by collapsing all instance nodes in the instance tree to one node, if these instances belong to the same variant. Therefore we get the ordered graph structure where each element has, in general, multiple number of parent nodes and multiple number of children nodes. Element with no parents in this data structure corresponds to the **top variant** and can be distinguished as a **head** of the variant graph. The elements without children nodes correspond to the bottom variants, with no referenced cells and can be distinguished as **the leafs** of the variant graph. Communication in both directions, from the child node to its parent and vice versa is possible. Therefore for a given variant the user algorithm can access all it's father variants and all it's children variants, directly.

Variant graph allows the application to generate results that are valid for all instances of the given variant simultaneously and still be able to, if necessary, communicate with current variant's immediate parents or children. In order to realize this, a set of methods is added to `Base_Variant` class, together with needed supporting data structures. Therefore, Base_Variant class defines a list of:

- **parents**: the parent variants of the actual variant stored as a list of Base_VariantInstantiationLeader objects. Each Base_VariantInstantiationLeader is a parent variant and a list of instances of the current variant in this parent variant. A recursion over the parents gives all instantiations of a variant.
- **subVariants**: maps each sub instance of the current variant to the associated variants. These subvariants pointers are again stored in a vector. The relation to the instances themselves is again done via the `HasIndex` class which is also a base class for the `Base_Instance` class. The subVariants might be used for top-down walks.

Applications of the Variant Concept

The variant concept is applied in cases where the algorithm needs to personalize given instantiations of the definitions resolving their parameters. The common application is in ERC for highvoltage checks. Once we apply all relevant parameters and pass datatypes over the hierarchy we can create the optimal set of variants and get irredundantly the results valid for all instantiation places of each variant.

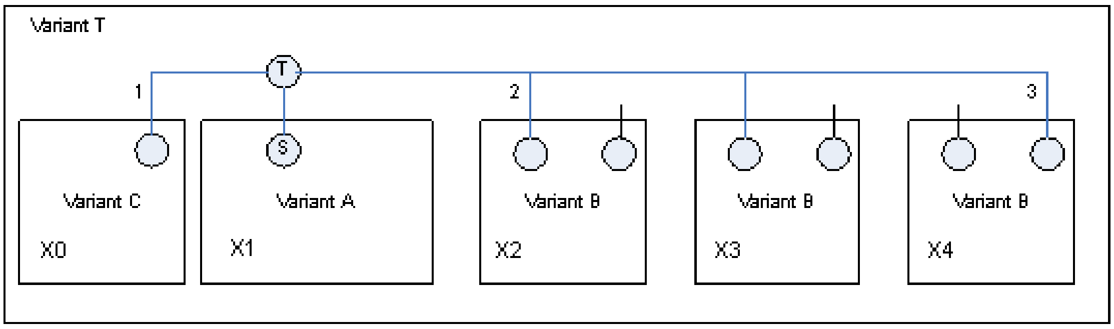
Variants are very important for our solution of the problem of hierarchical structural pattern matching. Pattern matching language allows the usage of node typses ,as well as device parameters. For this reason we will build our algorithms on top of variants that upgrade Base_Cell definition with the important interface that handles the additional properties (the interface to access the information about the node type of the given cell).

For simplicity and not affecting the generality of the explanations of the concepts that were used in this thesis, we will avoid the complication the current implementation of the variant concept introduces and use the terms cell and variant equally, as we have already stressed.

Appendix B (Fingerprint verification principle)

In order to explore the possible enhancements of the execution time that the employment of the hierarchical data model can offer, we have defined the fingerprint verification principle. This principle is adding additional functionality to the virtually flattened view. It is related to the way each net's pins are iterated over. With fingerprint verification we tend to optimize the iteration process and skip all similar iterations. This principle is an update of the multiple context hierarchical node iteration process. +

The principle can be well described and understood using the following example:



In the figure above we see one hierarchical node. The iteration over its pins starts from the subnode in the variant A that is denoted with S. The default iteration would, after going up the hierarchy visit all the neighbors of the subnode T, than descends to all of the variants that are instantiated in the context of the variant T. This is sometimes not optimal. The neighboring devices of the node 1 of the variant B would repeat twice in the iteration. If no positive conclusion (a successful matc-h) was done and the iteration was uninterrupted, we can skip all the redundant subnodes.

This is exactly how the fingerprint verification is defined. At the context of the subnode T, we can maintain information about the instantiations of the variants to which the algorithm has descended to. The information that is maintained is simply a pair formed by the pointer to the given variant and the pointer to the relevant variant pin. If we employ such strategy, at the level of the subnode T the algorithm would first verify if the relevant fingerprint for the Variant C exists, after the determining that the variant is new to the iteration the relevant fingerprint is stored in a specific container (of the given iteration context of the subnode T). The algorithm, further, descends to the Variant B, using the first pin, and skips the instance X3 of the Variant B, after finding that the relevant fingerprint already exists. The next instance where the optimized iteration would be continued is the instance X4, of the Variant B. This time the entry point to the variant is the pin 2, thus a different fingerprint to the one that was already left at the X2.

In order to assure the functional correctness of such an optimized iteration some issues have to be taken into account. For instance, the algorithm might enter the given variant at one pin and leave it at another concluding only outside of the given variant that it can't find the proper match. This might not be the case for some other instantiation. Therefore, in the case where the algorithm leaves the context of the given variant (runs through it) no fingerprint should be left. The issues like this should be addressed in order to have the proper functionality of this optimization.

Appendix C (Hierarchical matching example)

In the following example we are going to match two patterns with the new hierarchical pattern matching algorithm. The algorithm works incrementally, i.e. a pattern can be based on the output of a previous pattern match. Therefore, at first we will isolate all inverters in the hierarchical design using the pattern in Figure C-1 (a). The second matching process is analogue to the first one, still in it we are searching for a specific interconnection of two inverters, Figure C-1 (b).

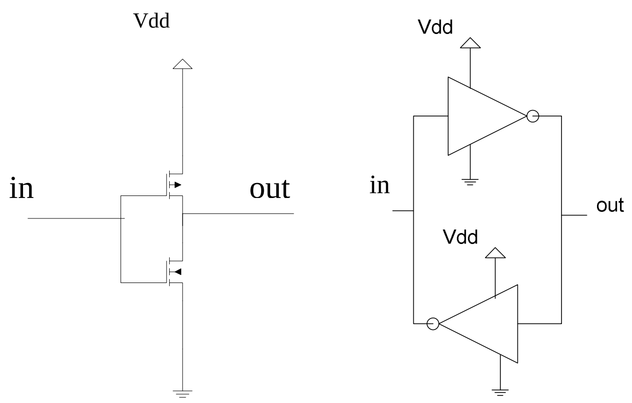


Figure C-1 – Matching Patterns. a) a pattern to match an inverter out of two relevant transistors. b) a pattern to match a latch out of two inverters.

The pattern matching algorithm, in search for these two patterns will be applied on the example hierarchical design shown in Figure C-2. The given hierarchical design semantically describes a latch circuit. Transistors are abstracted in separate subcircuits, MP, which contains an mp transistor alone, and MN, which holds an instance of the mn transistor device. These two cells are further, instantiated in the cell A, as instances X1 of MP and X2 of MN. On the top level, the cell A is, again, instantiated twice: instance X1 of A and X2 of A. The instances are interconnected in a way

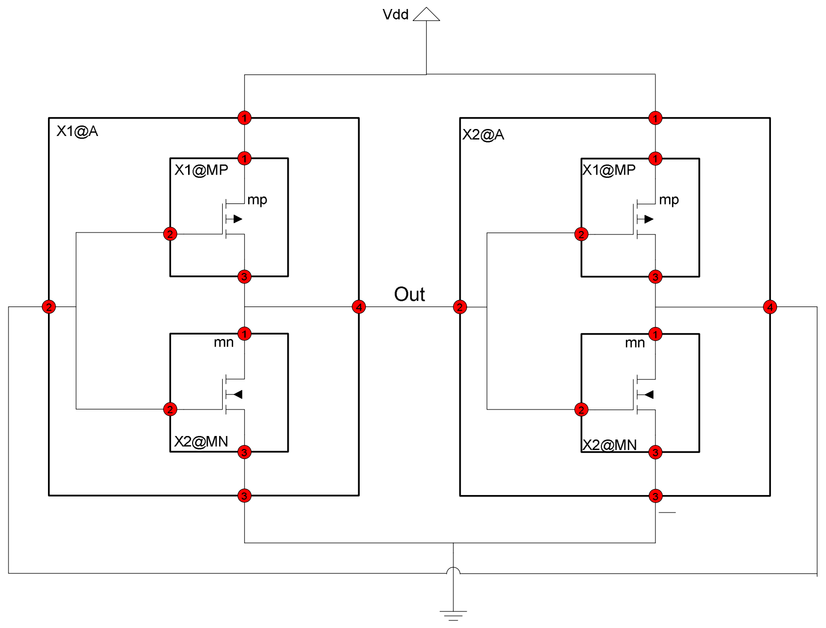
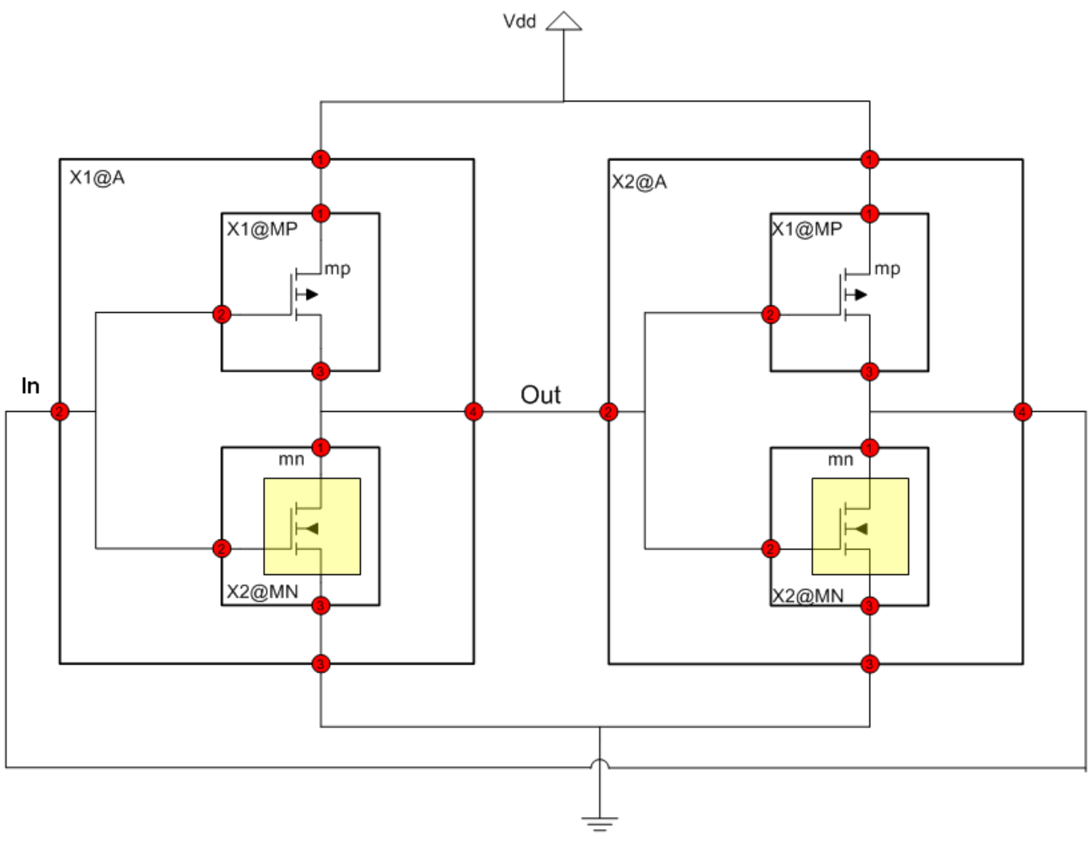
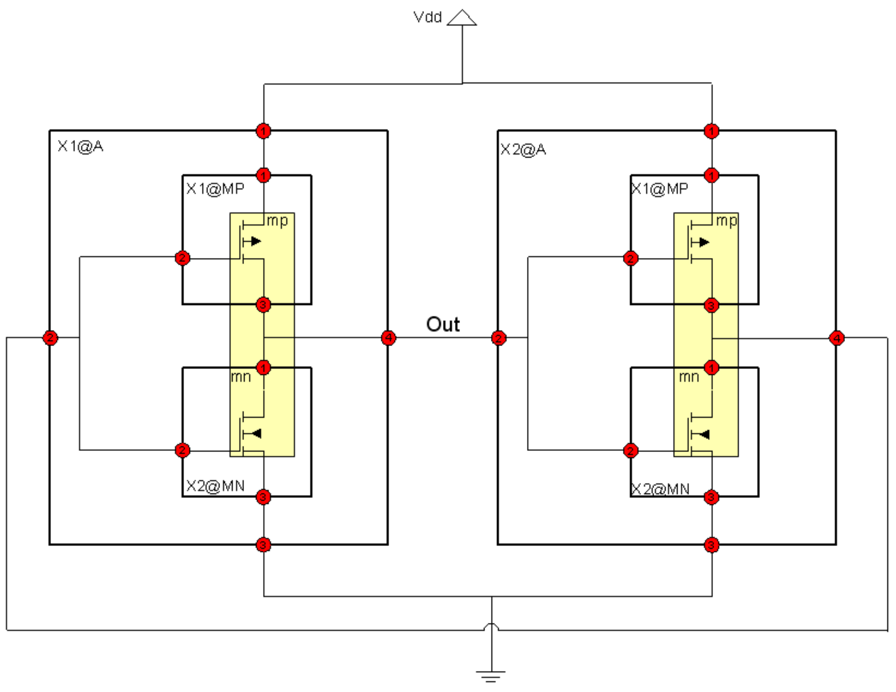


Figure C-0-1 – Hierarchical representation of a latch circuit.

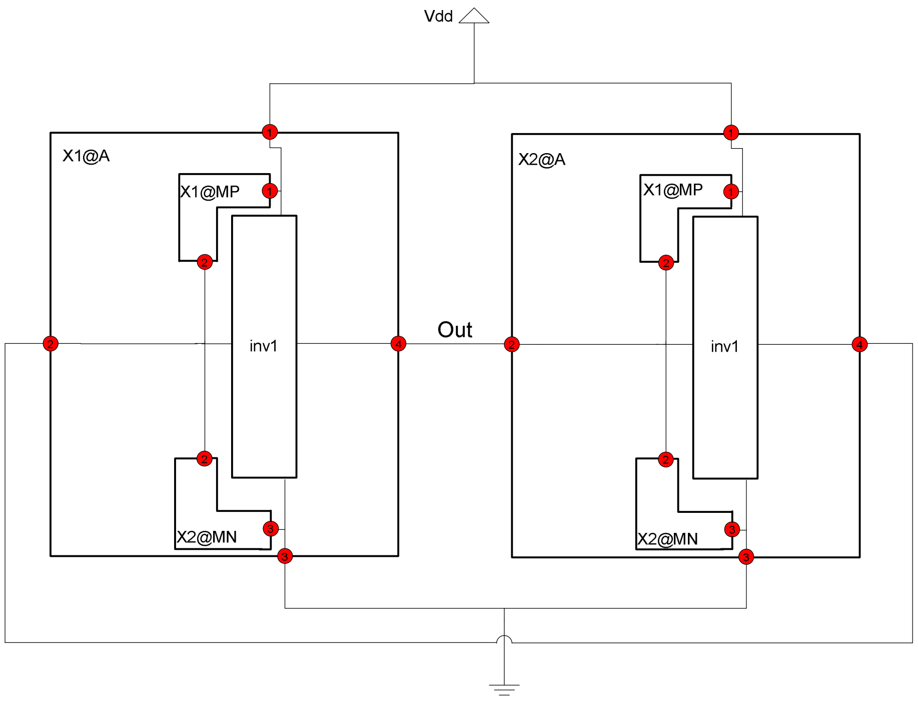
that the overall structure forms a topology of a latch circuit. Therefore, appropriate pins of instances X1 and X2 are shorted, or connected to defined fixed voltages Vdd and gnd. In this level, two semantically important nets are named in and out, for better understanding.



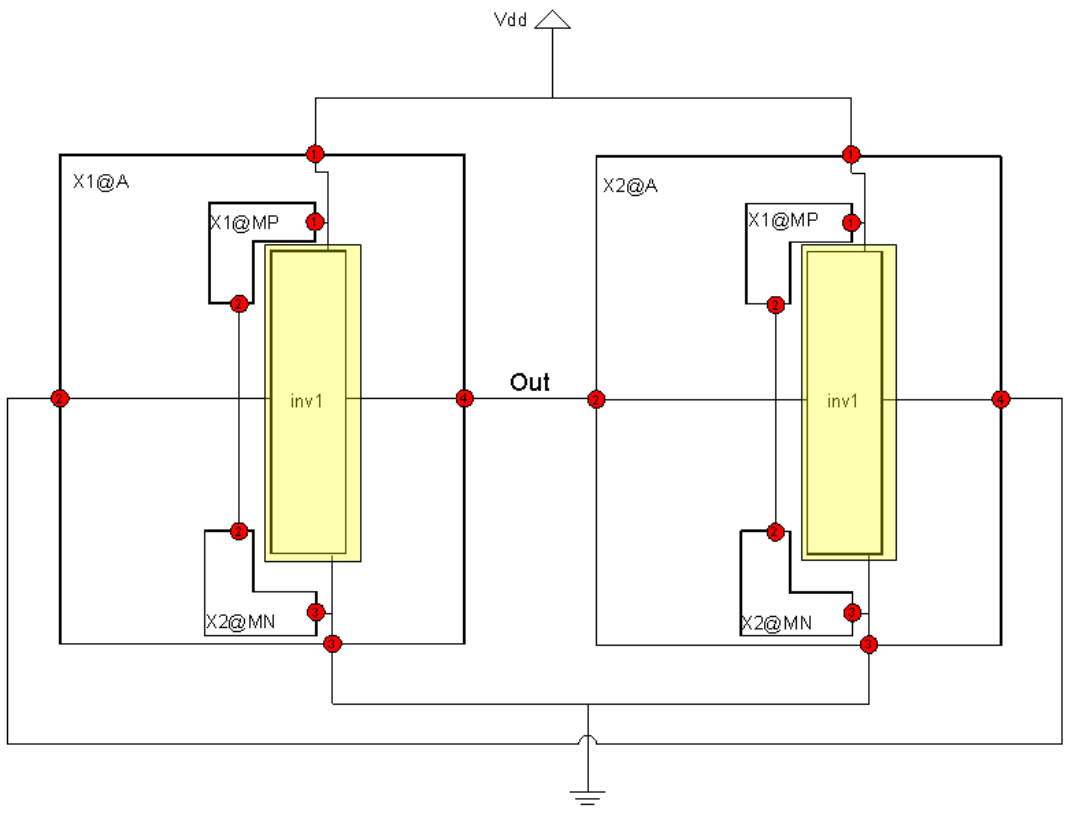
STEP 1: The algorithm starts with the device mp in cell MN. The context level is the MN cell. Following the hierarchical node through port 1, we have to change to the next hierarchy level. The algorithm thus sets the relative top level of the virtual copy of the transistor mn to cell A. This cell A exists in two places in the hierarchical circuit. Therefore also this virtual copy of the transistor exists twice when looked upon the circuit flat. These copies are marked with the yellow, semitransparent field.



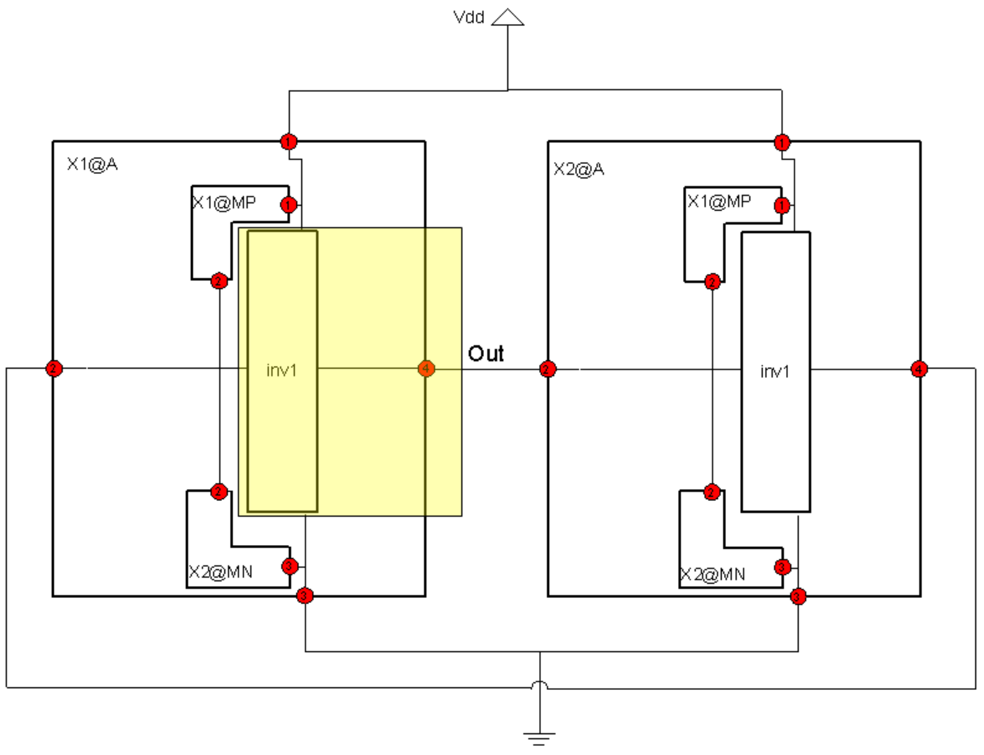
STEP 2: As the matching of the first pattern continues, the VFV dynamically switches the active context from the cell A to the instance X1 of the cell MP, creating the virtual copy of the transistor mp. In the figure, the algorithm has created a consistent flat view of the correct arrangement of two transistors (mp and mn). Still, thanks to the hierarchical layout of the example this virtual view occurs twice which is apostrophed with the yellow patch.



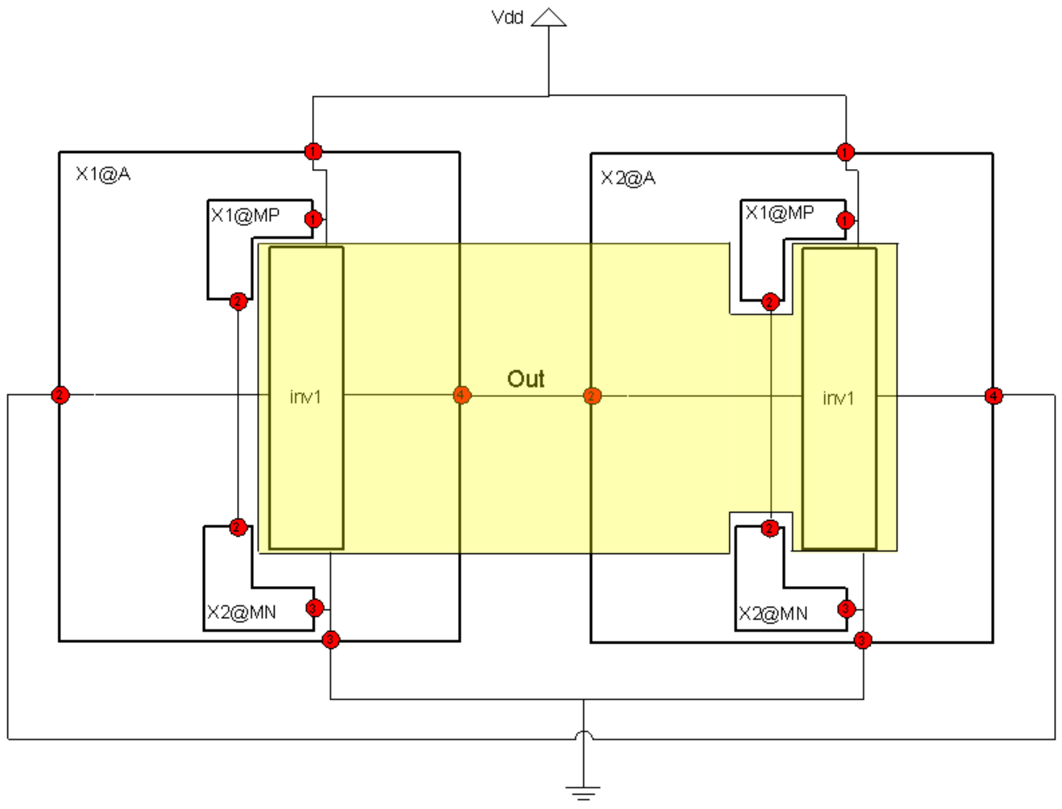
STEP 3: Since the current context view matches the topology of the pattern which was being searched the commitment step is performed. This means that a new instance inv1 of the new subcircuit, whose topology is identical to the pattern, is added to the hierarchical schematics. This modifies the topology of the cells MP and MN inside of A. The devices mp and mn are removed from MP and MN and are moved to the newly inserted subcircuit inv1 . Note that this would produce a variant of the cell MP or MN if e.g. we had another instance of MP placed somewhere in the design without an adjacent MN. In such a case this instance of cell MP would keep its old topology. Proper connectivity is still maintained. Note that the position of the pins of cells MP and MN is changed to make the figure more elegant. That has, however, no electrical or semantic importance. Additionally, to depict the change of the hierarchical topology the shapes of the cells MP and MN intuitively show that some devices are now removed.



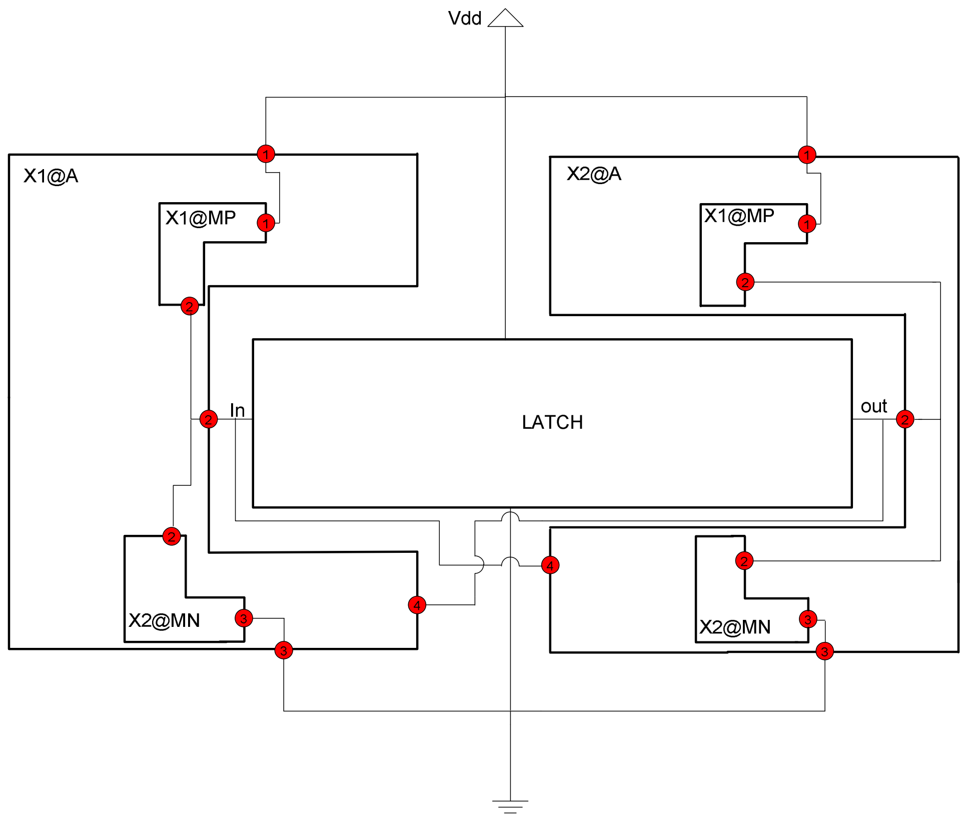
STEP 4: After the commitment the second pattern (interconnected inverters) is being searched for. Now the building blocks of this pattern are the inverters which have been recognized and committed to the hierarchy during the match process of the previous rule. This step is analogue to the matching of the previous pattern at STEP 1.



STEP 5: The context of the flat view dynamically gets changed to the parent cell of the cell A. In our case it is the top cell, but in general it can be any regular cell. Again a virtual copy of the inv1 within the top cell is generated. The multiplicity of the locally flat view is now equal to the number of instantiations of this “relative top” cell. Note that the pattern is not anymore valid for the instance inv1 inside instance X2 of A!



STEP 6: This is analogue to the STEP 2 of the first matching process. The flat view properly represents the arrangement of two inverters and their interconnection. This leads to another match as the topology of the current flat view is identical to the topology of the latch pattern.



STEP 7: This step is analogue to STEP 3 of the first matching process. An instance of a cell containing the latch pattern is committed to the relative top level. This influences the topology of two instances of the cell A. As the two instances are connected differently when looking from top level, two variants of the cell A are generated. The new hierarchical topology is consistent and prepared for any other algorithm.

To summarize, in this example we have successfully demonstrated one possible scenario where we have used the functionality of the VFV to be able to see specific parts of the hierarchical netlist as if they were flat. Therefore the utility algorithm, could navigate through the neighbourhood of each starting device that was offered by the specific device iterator of the VFV. For each matching place a materialized flat data portion was built and kept in consistency with the hierarchical netlist. For each successful match, the current state of the materialized flat data portion was committed to the hierarchical netlist, affecting the neighbouring hierarchy, by the sophisticated algorithm.

Bibliography

1. Moore, G.E., *Cramming more components onto integrated circuits*, in *Readings in computer architecture*. 2000, Morgan Kaufmann Publishers Inc. p. 56-59.
2. Frerichs, M., Attributed to Steinkopf, U., *EDA tool Effort Waves*. 2009, personal communication.
3. Diestel, R., *Graph Theory*. 2005: Springer-Verlag.
4. Ullmann, J.R., *An Algorithm for Subgraph Isomorphism.* J. ACM, 1976. **23**(1): p. 31-42.
5. Corneil, D.G. and C.C. Gotlieb, *An Efficient Algorithm for Graph Isomorphism.* J. ACM, 1970. **17**(1): p. 51-64.
6. Cortadella, J.a.V., G. *A relational view of subgraph isomorphism*. in *Proc. 5th Int. Seminar on Relational Methods in Computer Science*. 2000. Québec, Canada
7. Gold, S. and A. Rangarajan, *A Graduated Assignment Algorithm for Graph Matching.* IEEE Trans. Pattern Anal. Mach. Intell., 1996. **18**(4): p. 377-388.
8. Gold, S. and A. Rangarajan, *Graph matching by graduated assignment*, in *Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition (CVPR '96)*. 1996, IEEE Computer Society.
9. Papadimitriou, C.H. and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. 1998: Prentice-Hall, Inc. 496.
10. Kosowsky, J.J. and A.L. Yuille, *The invisible hand algorithm: solving the assignment problem with statistical physics.* Neural Netw., 1994. **7**(3): p. 477-490.
11. van Genderen, A.J. and N.P. van der Meijs. *Reduced RC models for IC interconnections with coupling capacitances*. in *Design Automation, 1992. Proceedings., [3rd] European Conference on*. 1992.
12. Vanoostende, P., P. Six, and H.J. De Man, *DARSI: <e1>RC</e1> data reduction [VLSI simulation]*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 1991. **10**(4): p. 493-500.
13. Luellau, F., T. Hoepken, and E. Barke, *A technology independent block extraction algorithm*, in *Proceedings of the 21st conference on Design automation*. 1984, IEEE Press: Albuquerque, New Mexico, United States.
14. Ohlrich, M., et al., *<italic>SubGemini</italic>: identifying subcircuits using a fast subgraph isomorphism algorithm*, in *Proceedings of the 30th international conference on Design automation*. 1993, ACM: Dallas, Texas, United States.
15. Ebeling, C., *GeminiII: A Second Generation Layout Validation Tool*, in *Conference on Computer Aided Design (ICCAD)*. 1988. p. 322-325.
16. Ling, Z., *An algorithm for subgraph isomorphism based on resource management with applications*. 1998, University of Hawai'i. p. 190.
17. Ling, Z., *Subcircuit Extraction with Subgraph Isomorphism*, IBM Almaden Research Center - EDA Shape Processing.

18. Chanak, T.S., *Netlist Processing for Custom VLSI via Pattern Matching*. 1995, Stanford University.
19. Olbrich, M., A. Rein, and E. Barke, *An improved hierarchical classification algorithm for structural analysis of integrated circuits*, in *Proceedings of the conference on Design, automation and test in Europe*. 2001, IEEE Press: Munich, Germany.
20. Pelz, G. and U. Roettcher, *Pattern matching and refinement hybrid approach to circuit comparison.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 1994. **13**(2): p. 264-276.
21. Pelz, G. and U. Roettcher. *Circuit comparison by hierarchical pattern matching*. in *Proceedings of the Conference on Computer Aided Design (ICCAD).* 1991.
22. Rubanov, N., *SubIslands: the probabilistic match assignment algorithm for subcircuit recognition.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2003. **22**(1): p. 26-38.
23. Rubanov, N., *A High-Performance Subcircuit Recognition Method Based on the Nonlinear Graph Optimization.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2006. **25**(11): p. 2353-2363.
24. Rubanov, N., *An efficient subcircuit recognition using the nonlinear graph matching*, in *Proceedings of the 18th annual symposium on Integrated circuits and system design*. 2005, ACM: Florianolpolis, Brazil.
25. Rubanov, N. *Bipartite graph labeling for the subcircuit recognition problem*. in *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on*. 2001.
26. Zhang, N. and D.C. Wunsch, II. *A fuzzy attributed graph approach to subcircuit extraction problem*. in *Fuzzy Systems, 2003. FUZZ '03. The 12th IEEE International Conference on*. 2003.
27. Vijaykrishnan, N. and N. Ranganathan. *SUBGEN: a genetic approach for subcircuit extraction*. in *VLSI Design, 1996. Proceedings., Ninth International Conference on*. 1996.
28. Nian, Z., D.C. Wunsch, II, and F. Harary, *The subcircuit extraction problem.* Potentials, IEEE, 2003. **22**(3): p. 22-25.
29. Chang, W., *A Novel Extraction Algorithm by Recursive Identification Scheme*, in *IEEE International Symposium on Circuits and Systems*, 2001, Editor. 2001: Australia.
30. Zhang, N. and D. Wunsch, *A Novel Subcircuit Extraction Algorithm using Heuristic Dynamic Programming (HDP)*, in *International Conference on VLSI*. 2002: Las Vegas, Nevada, USA.
31. Batra, P. and D. Cooke, *Hcompare: a hierarchical netlist comparison program*, in *Proceedings of the 29th ACM/IEEE conference on Design automation*. 1992, IEEE Computer Society Press: Anaheim, California, United States.
32. Terem, Z.K., G.; Vardi, M.Y.; Irron, A., *Pattern search in hierarchical high-level designs*, in *Electronics, Circuits and Systems, ICECS 2004, IEEE International Conference on*. 2004. p. 519-522.
33. Pattee, H., *Hierarchy theory: The challenge of complex systems*. 1973: George Braziller New York.
34. Ahl, V. and T. Allen, *Hierarchy theory: a vision, vocabulary, and epistemology*. 1996: Columbia University Press.

35. Engels, G. and A. Schürr, *Encapsulated hierarchical graphs, graph types, and meta types*. Electronic Notes in Theoretical Computer Science, 1995. **2**: p. 101-109.

36. Jones, M.C. and E.A. Rundensteiner, *View materialization techniques for complex hierarchical objects*, in *Proceedings of the sixth international conference on Information and knowledge management*. 1997, ACM: Las Vegas, Nevada, United States.

37. Lavagno, L., G. Martin, and L. Scheffer, *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. 2006: CRC Press, Inc.

38. Mallis, D., *Si2 OpenAccess API Tutorial*. 2008: Silicon Integration Initiative, Inc.

39. *Object-oriented databases with applications to CASE, networks, and VLSI CAD*, ed. G. Rajiv and H. Ellis. 1991: Prentice-Hall, Inc. 447.

40. Bales, M., *Facilitating EDA Flow Interoperability with the OpenAcess Design Database*, in *Electronic Design Processes Conference*. 2003.

41. Brevard, L., *Introduction to Milkyway*, in *Electronic Design Processes Conference*. 2003.

42. Guiney, M. and E. Leavitt, *An introduction to OpenAccess: an open source data model and API for IC design*, in *Proceedings of the 2006 conference on Asia South Pacific design automation*. 2006, IEEE Press: Yokohama, Japan.

43. Blanchard, T., *Assessment of the OpenAccess Standard: Insights on the new EDA Industry Standard from Hewlett-Packard, a Beta Partner and Contributing Developer*, in *Proceedings of the 4th International Symposium on Quality Electronic Design*. 2003, IEEE Computer Society.

44. Fowler, M., *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 2003: Addison-Wesley Longman Publishing Co., Inc. 256.

45. Rumbaugh, J., I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. 2004: Pearson Higher Education.

46. Rundensteiner, M.J.a.E.A., *An Object Model and Algebra for the Implicit Unfolding of Hierarchical Structures*. 1997, Electrical Engineering and Computer Science Dept., University of Michigan.

47. Truyen, E., et al. *A generalization and solution to the common ancestor dilemma problem in delegation-based object systems*. in *Proceedings of the 2004 Dynamic Aspects Workshop* 2004.

48. Gamma, E., et al., *Design patterns: Elements of reusable object-oriented design*. 1995, Addison-Wesley Reading, MA.

49. Freeman, E., B. Bates, and K. Sierra, *Head first design patterns*. 2004: O'Reilly & Associates, Inc.

50. Kappel, G., W. Retschitzegger, and W. Schwinger. *A Comparison of Role Mechanisms in Object-Oriented Modeling*
in *Proceedings of the conference: Modellierung '98, GI-Workshops*. 1998. Münster, Germany.

51. Fowler, M., *Dealing with roles*, in *Conference on Pattern Languages of Programs*. 1997: Monticello, Illinois, USA. p. 97-34.

52. Bäumer, D., et al., *The role object pattern*, in *Conference on Pattern Languages of Programs*. 1997: Monticello, Illinois, USA.

53. Josuttis, N., *The C++ Standard Library: A Tutorial and Reference*. 1999, Addison Wesley Professional.

54. Cormen, T.H., et al., *Introduction to Algorithms*. 2001: McGraw-Hill Higher Education.

55. Duncan, C.A., S.G. Kobourov, and V.S.A. Kumar, *Optimal constrained graph exploration.* ACM Trans. Algorithms, 2006. **2**(3): p. 380-402.
56. Frerichs, M., *Netstats, Qimonda inhouse tool*. 2005, personal communication.
57. *Valgrind Home*. [cited 2008. 12.12.]; Available from: http://www.valgrind.org.