Verification of electronic designs by reconstruction of the hierarchy

Verification of electronic designs by reconstruction of the hierarchy

A.P. Kostelijk

Verification of electronic designs by reconstruction of the hierarchy

A.P. Kostelijk

The work described in this thesis has been carried out
at the PHILIPS RESEARCH LABORATORIES at Eindhoven,
the Netherlands, as part of the Philips Research programme.

CIP-gegevens Koninklijke Bibliotheek, Den Haag
Kostelijk, A.P.
Verification of electronic designs by
reconstruction of the hierarchy
Proefschrift Technische Universiteit Eindhoven,-Met lit. opg.,
-Met samenvatting in het Nederlands.
ISBN 90-74445-14-4
Trefw.: IC-ontwerp, verificatie, patroonherkenning, algorithme

Verification of electronic designs by reconstruction of the hierarchy

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen, in het openbaar te verdedigen op woensdag 28 september 1994 te 16.00 uur

door

Anton Pieter Kostelijk

Geboren te Grootschermer

Dit proefschrift is goedgekeurd door de promotoren

prof.dr.Ing. J.A.G. Jess en
prof.dr.ir. W.M.G. van Bokhoven.

Acknowledgements

This work was performed at the IC-design Centre, Philips Research Laboratories Eindhoven, since 1985. Many people have contributed to this work. In particular I owe a lot to the following persons.

The management of the IC-design Centre, especially groupleader Gerard Beenker, his predecessor Leo Nederlof, and former director Theo Claasen, for giving me the opportunity and support to perform this work.

Guido Schrooten for being the co-initiator of Vera, and for major contributions to the initial Vera setup.

Brian Lynch and Paul Kuppen († 1992) for sharing their talents with me for many years, and creating major parts, manuals and applications.

Bart De Loore, our "super user", for his questions, discussion, enthousiasm, innovative Vera usage, and joint publications.

Johan Jonkheid for his initiative and endurance power to set up the Vera support team.

Andries van der Veen, René Segers and Marc Verra for their contributions.

Furthermore, I thank all users for their interest in my work and their criticism.

I thank all current and former members of the IC-design Centre and the excellent system support team for a pleasant and cooperative atmosphere.

The following people have given substantial feedback on drafts of this thesis: Prof. J.A.G. Jess, Prof. E.H.L. Aarts, Prof. W.M.G. van Bokhoven, Prof. P.M. Dewilde, Gerard Beenker, André Slenter, Simon Thorn.

Finally, I am grateful to my wife, Elly Vogelzang, for her support and encouragement.

Contents

Chapter 1

Introduction

This chapter introduces and provides a motivation for the subject of this thesis. The subject is the verification of electronic designs by hierarchy reconstruction, and in particular the structure verification of the layout design of an IC (Integrated Circuit). The first section introduces briefly the notions of IC-design, synthesis and verification. The second section describes the main trends in IC-design. In the third section, the IC-design process is subdivided into two steps, high level design and layout design, to indicate the position of the thesis' subject in this process. Both steps are described in some detail. The last section describes the aim, subject and structure of the thesis.

1.1 IC-design, synthesis and verification

This section introduces the notions of IC-design, synthesis and verification. IC-design is the implementation of an initial IC-specification into a layout, which after processing results in an IC that meets the initial IC-specification. An initial IC-specification describes the required behavior of the IC, and constraints of the IC, such as timing, layout size, package demands, power consumption, etc. A layout describes the geometrical positions of different materials in an IC, thus forming the basic components, such as transistors, resistors and capacitances, and the wiring network connecting the basic components that determine the IC's behavior and properties. An example of a layout is shown in Figure E.2 of Appendix E. The implementation is performed in a number of design steps as indicated in Figure 1.1 to keep overview of the large number of details. A design step

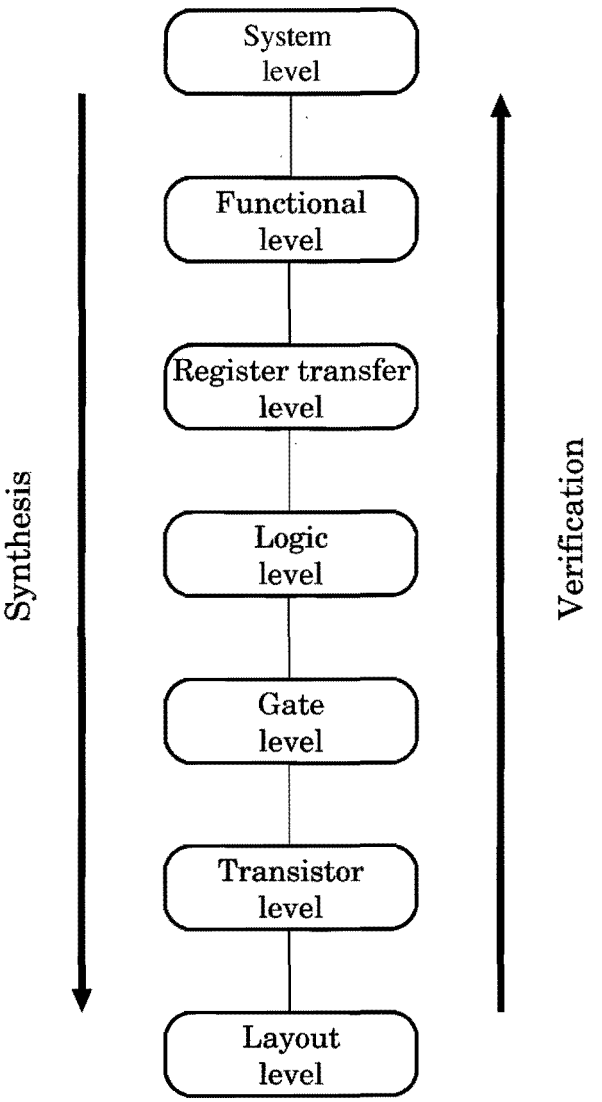


Figure 1.1: *Example of a design trajectory.*

consists of a synthesis phase and a verification phase. In the synthesis phase of a design step, a description which is the result of the previous steps, called the specification of the current step, is transformed into a new description, that brings the aim, a layout, closer by. The new description is called the implementation of the current step. Since synthesis is a complicated activity, the implementation may be incorrect with respect to the specification. In the verification phase of a design step, it is established that the implementation of the current step is consistent with the specification of the current step. When the verification is successful, the implementation of the current step is accepted as the specification of the next step. When the verification is not successful, the errors must be corrected.

The description of the IC-design process so far is an example of a divide-and-conquer top-down method. This method is applicable only when the consequences at the lower levels of high-level choices can be estimated accurately. However, when details of an IC-design are not yet filled in, for example the size of the design remains hard to predict from the initial IC-specification. In addition, details at a lower level may dramatically influence the higher levels. Therefore, a strict divide-and-conquer design method is impracticable. In current design practice, several complete design iterations from initial specification to layout are necessary, to map out the consequences of the choices made during the synthesis phases. Figure 1.1 shows the hierarchy levels and steps for one example of a design style [Veend92.1]. Each level is indicated by a rounded box. A complete iteration consists of the traversal of a top-down synthesis trajectory and a bottom-up verification trajectory. On a smaller scale, e.g., between two levels, many top-down bottom-up iterations are usually made.

1.2 Trends in IC-design

This section briefly analyzes the trends in IC-design. Even after four decades of IC-design, the main trend remains unaltered: the complexity of an IC, i.e., the number of details involved for designing the IC, is larger than the complexity of the previous IC. As a consequence, the amount of specialization keeps on growing, and the CAD-design environments become larger and larger. The relative design effort spent on verification, compared to the effort spent on synthesis, grows due to the growing number of details. In addition, the permanent flow of newly created synthesis tools shifts the design bottleneck even further towards verification.

Silicon technology nowadays allows the integration of many millions of transistors. However, the number of transistors of an IC is only an indirect indicator for the complexity of a design. For instance, a memory IC may contain many millions of transistors, but most of them are designed by repeating the same pattern millions of times. Because memory ICs require little design effort per transistor, the ICs having the largest transistor count are usually memories. The main design challenge is not to increase the transistor count that can be handled, but to design complex systems with as little effort as possible. Complex systems consist of many different functions and parts, resulting in a design containing millions of transistors without global repetition.

An additional phenomenon that comes with the complexity growth is that todays ICs are no longer designed with a single design method or a single design style only. A design is the result of a mixture of design methods and styles, each mastered by a team of specialists. An IC may contain parts consisting of standard cells, ROM modules, PLA modules, embedded memories, analog parts, macro-cells, and library blocks. Different parts may be designed by logic synthesis, manually, by application specific synthesis tools, or be copied from another design etc. All parts are integrated into one layout. This involves the placement and connection of many thousands of terminals. Whether the right parts are present, whether they are implemented correctly, and whether the parts are connected correctly to each other, has become a major verification question and is therefore the main subject of this thesis.

1.3 High level design and layout design

Two major steps can be identified in a design process, i.e.,

- high level design and
- layout design.

In the following, both steps are briefly described, with emphasis on the verification phases.

High level design
The aim of high level design (see also Figure 1.2) is to take the initial IC specification that describes input output behavior and a set of constraints, to find a high level structure that implements the specification

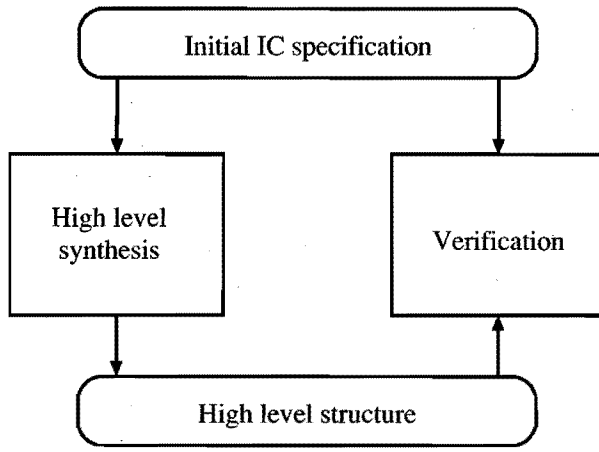


Figure 1.2: *High level design.*

correctly. A structure refers to a set of interconnected components that make up the design. In general, a structure or network can be given at various levels of abstraction, relating to various levels of detail. Since high level design deals with the properties as present at the levels close to the initial IC-specification, the outcome is a high level structure that contains no lower level details. The high level structure description may consist of macro-cells, standard-cells, analog cells, etc. Different parts of the initial IC specification may be synthesized by different design methods and styles. Like all design steps, high level design consists of a synthesis phase and a verification phase. In the high level synthesis phase, the top-down step is performed. In the high level verification phase one establishes the consistency between the initial IC specification and the high level network. This is mainly done by simulation, although for specific steps, better alternatives exist [Malik88], [Koste93], [Genoe92]. Much of research is done in this area [AFMC89], [TPCD92], [CHAR93], but many methods have not been as successful in the design practice as their creators hoped.

Layout design

Figure 1.3 shows the layout design phase. In this part of the design trajectory, the high level structure resulting from high level synthesis is transformed into a layout. A layout describes the geographical position of dif-

ferent materials in an IC. The different materials represented in the layout are usually referred to as the "layers" of a design. By using floor planning, placement and routing tools, various libraries, macro-cell generators and usually some manual editing, etc., a complete layout is generated. During this process, not only many different libraries and complicated CAD tools are used, but also the different design parts are brought together and usually some manual modifications are made. Verification is therefore an essential part of this design step, taking a major part of the layout design effort. These facts have motivated the subject described in this thesis, i.e., the improvement of verification in the layout design. The aim of the layout

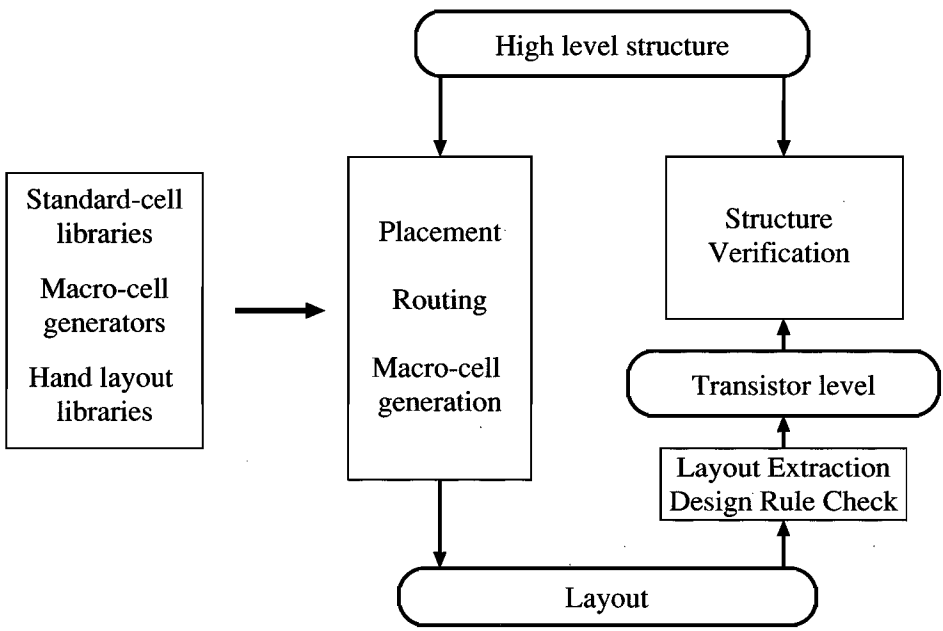


Figure 1.3: *Layout design.*

verification phase is to check whether the high level structure description has been implemented appropriately in the layout. In the following section, this process is described bottom-up, starting with the layout. The verification is subdivided into three steps.

In order to produce operational basic components such as transistors, resistors and capacitors, the layout must obey technology dependent de-

sign rules [Veend92.2] that prescribe geometrical constraints on the layers. Therefore, the first verification step is to verify the layout design rules. Commercial layout extraction tools such as DRACULA, or Philips' LOCAL45, are well capable of verifying the absence of layout design rule violations. Since this verification step is common and widely used, it will not be described in further detail here.

A second step consists of the extraction of basic components (transistors, capacitors, resistors) from layout, as shown in Figure 1.3. The tools for design rule checking are able to perform this task as well and it is done simultaneously with the design rule checking. Most of the components are transistors at this level, only for some analog parts may capacitors and resistors be extracted as well. The effect of *parasitic* capacitors and resistors, e.g., resulting from long wires in the layout, are usually checked by timing verifiers or by circuit simulation. Neither timing verification nor parasitics are subjects of this thesis.

The third verification step, called structure verification, consists of checking whether a high level structure description has been implemented correctly at the transistor level. The focus of this thesis is on an effective method of performing structure verification. This method, called "hierarchy reconstruction", is introduced in the next section.

1.4 Hierarchy reconstruction

This section describes the subject, motivation, relevance and structure of this thesis. The subject is an effective method of performing structure verification, called "hierarchy reconstruction". The first paragraph describes the need for structure verification. The second paragraph derives the properties that a structure verification method should have. The third paragraph explains briefly the verification method including the relevance of the method compared with other approaches. The final paragraph describes the structure of the thesis.

The need for structure verification

As indicated in Section 1.2 and elaborated in Section 1.3, a modern IC design consists of many parts coming from many different sources, put together during the layout design phase. Not only is the design of each part a complicated task that needs verification, but also putting the parts together appropriately involves many CAD tools and libraries, and often error-prone

manual work. Furthermore, every non-trivial computer program contains bugs, and since many computer programs are involved during layout design, one cannot rely on the result without checking afterwards whether the right components have been connected correctly.

Structure verification requirements

Any structure verification method must check whether the transistor level structure is connected correctly according to the high level structure. To get a reliable verification method, it should not rely on information added during the synthesis phase, but start from the final result, i.e., the layout. The aim of IC-design is a correct layout where a processed IC will meet the initial IC-specifications. This implies that both synthesis and verification by itself are not aims of IC-design, but inevitable steps, performed to get a reliable layout. A structure verification tool should therefore require little designer effort and few computer resources. The tool should also be able to handle current industrial designs, i.e., a design composed of a million transistors. A structure verification tool should not only signal the presence of an error, but it should indicate the cause of errors, to diminish the extra design effort in correcting the errors. With the growing number of components and connections in IC-designs, the importance of error indication grows as well. The common practice of combining different (perhaps slightly modified) parts coming from different sources in one layout, indicates that connectivity errors at the high structure level are likely to occur, so especially errors at different structure levels should be indicated appropriately.

Hierarchy reconstruction

Existing structure verification methods are simulation, functional abstraction, netlist comparison at the transistor level and rule-based recognition. As will be shown in Chapter 2, these methods fail to combine complete verification, reasonable run times and appropriate error indication. The hierarchy reconstruction method as described in this thesis however succeeds in combining complete verification, reasonable run times and appropriate error indication.

Hierarchy reconstruction is a method that starts with a transistor level netlist that has been extracted from a layout. By identifying clusters of basic components forming a higher level unit, one can reduce the size of the netlist, and obtain a netlist consisting of higher level components. By per-

forming this recognition process repeatedly, and on subsequent levels, one can obtain a netlist at the top level, allowing a high level netlist comparison to verify the correctness of the high level structure with the actual layout. When successful, the original high level structure has been reconstructed by the method. An advantage of the method is that any hierarchy that leads to the same top-level structure can be used, i.e., the verification hierarchy can be chosen independently to the one used in the layout synthesis phase.

The aim of this thesis is to describe the hierarchy reconstruction method, and to show that the method works in practice. Basically, the hierarchy reconstruction method is implemented as a sequence of different sub-circuit recognition operations. The core of the thesis describes the recognition algorithm in detail, to explain why the method works. The algorithm will be shown to combine high run time efficiency, flexibility and effective error indication.

Based on this work, an environment called Vera [Koste89], [Koste88], [Deloor90], [Koste91], [Koste92.2], [Koste92.3] has been developed that supports the hierarchy reconstruction method. Vera is an acronym for VERification Assistant.

Overview

The thesis is subdivided as follows. Chapter 2 describes existing structure verification methods found in literature. Chapter 3 describes the hierarchy reconstruction method, and the tools needed to make the method operational are inventorized. The main part of thesis is found in Chapter 4, where the recognition algorithm is described. Chapter 5 describes other tools, in addition to the sub-circuit recognition, needed to make the hierarchy reconstruction method work. In this chapter especially the verification of parameterized macro-cells such as a Random Access Memory layout part will be described. Chapter 6 shows some results of the method. The conclusions and suggestions for future work are given in Chapter 7.

Chapter 2

Literature on structure verification

The literature on structure verification is partitioned into a section describing different methods on structure verification, and a section on structure recognition algorithms. Structure recognition is the core of the hierarchy reconstruction method. The strong and weak points of the methods and algorithms are summarized and compared with the structure verification requirements of Section 1.4. A conclusion finishes this chapter.

2.1 Structure verification methods

Existing methods to verify a high level structure description versus the transistor level structure are simulation, functional abstraction and netlist comparison.

2.1.1 Simulation

The classic verification method, simulation, aims at predicting the behavior of a circuit for a given set of input patterns. After simulating the structure at high level and at the transistor level, the resulting behaviors should be the same. When every input pattern leads to similar behavior for both levels, the structures are proven correct. The prediction of behavior is based on models for component interconnection, and an algorithm that combines the models and input patterns. Therefore many distinct simulators exist, supporting various kind of models [Graaf89] and using

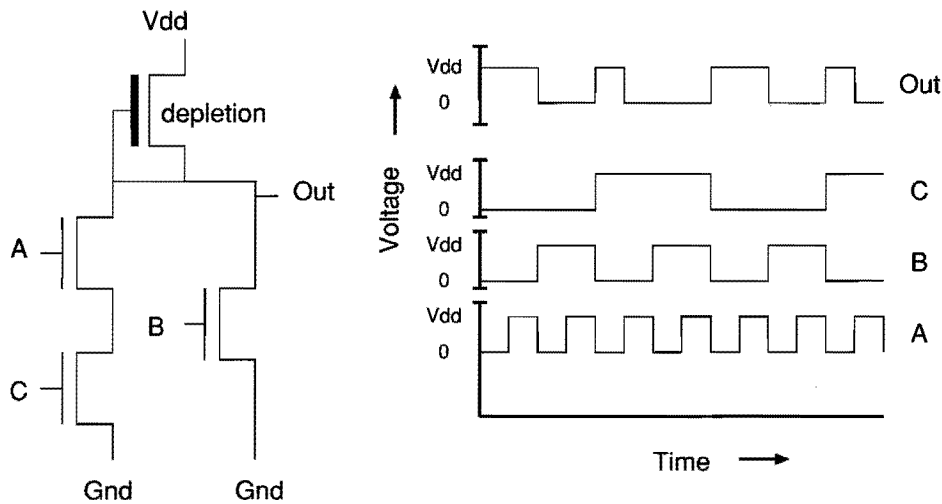


Figure 2.1: *Simulation.*

different algorithms [Jones94], [VHDL93], [Chua75], [Nagel75], [Feldm92]. At the transistor level, a switch-level simulator is used for digital designs, and a circuit simulator is used for analog designs. At high level, a VHDL simulator [VHDL93] is often used. Figure 2.1 shows a simulation example in which the Out signal is computed for given input signals A, B and C, for the transistor structure as drawn on the left hand side. A simplified switch-level model is used for this example. The depletion transistor is modeled as a finite resistance, and the NMOS transistors are modeled as ideal switches. A modern switch-level level simulator is described in [Jones94]. The predictive value of the computation depends on the models used and the numerical simulation algorithm. Except for several analog circuits, existing models and simulation algorithms in general lead to reliable predictions of circuit behavior. However, for verifying all possible input patterns the run time grows exponentially with the number of inputs and memory-cells, so even for small designs, simulation leads to excessive run times when used for structure verification. Also, the interpretation of simulation results, tracing back the origin of faulty behavior, is usually hard and time-consuming. Furthermore, a side effect of the specialization in IC-design mentioned in Section 1.2 is that the designer who combines all

parts into a complete IC has little knowledge of the details that are needed when searching for errors.

We conclude that simulation is an expensive and ineffective method for structure verification.

2.1.2 Functional abstraction

The functional abstraction method as described in [Apte82], [Boehn88], [Bryant87] computes the behavior of a transistor network by transforming the network into a set of Boolean operations. The advantage of this method with respect to simulation is that the complete behavior of a transistor level structure is derived in one step. Input patterns are not needed. The functional abstraction method is based on path-tracing. For each net, the paths leading to the ground or supply nets are analyzed. The rules to derive the Boolean function differ per technology. For instance (Figure

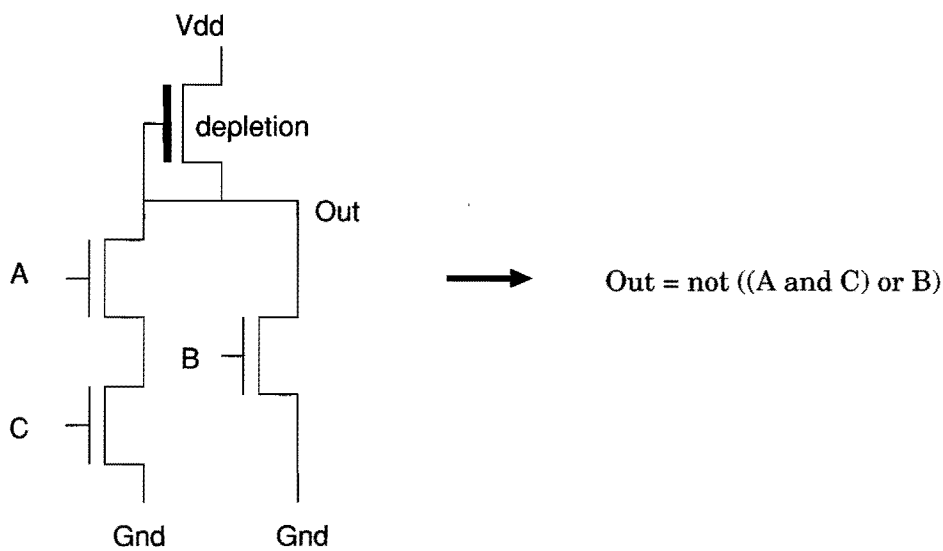


Figure 2.2: *Functional abstraction.*

2.2), in NMOS circuits [Apte82], a depletion transistor connects the supply net called Vdd with the intended Boolean output net called Out. Assume that the Gnd net has a constant potential of 0 Volt, and is also associated

with the Boolean False value, and the Vdd net has a constant potential of 5 Volt, associated with the Boolean True value. The pull-down function of the output net defines when the output has zero potential, i.e, when the Boolean output value is False. The pull-down function is found by interpreting parallel branches to Gnd as an OR function, and serial paths to Gnd as an AND function. The actual Boolean function of the Out net is now given by the Boolean negation of the pull-down function. For full CMOS, both a pull-down function and pull-up function are identified, which must be the Boolean negation of each other. As described in [Ramme92], [Bolse89], the functional extraction method has recently been elaborated for CMOS, in which special attention was paid to timing and clocking-strategies.

The functional abstraction method has been popular for some time now. The premise of the method is that the mapping of electronic functions into layout can be formalized by a simple set of mathematical rules. Although for a limited set of functions this is indeed the case, such as for some combinatorial gates in pure CMOS, this is certainly not the case for all implementations. For digital design parts, one needs additional manual hints for memory-cells. Also, pass gate logic, such as wired-ORs, are a problem [Veend92.3]. The modelling of sized transistors as Boolean networks remains an issue [Verli92]. In [Dever92], a mixed approach of functional abstraction and structure recognition is presented. Functional abstraction so far is limited to Boolean gate level. For instance, no general functional abstraction method exists that abstracts any set of Boolean gates forming an n-bit adder. Also, no automatic functional abstraction method is known for analog designs, at present.

Summarizing we conclude that the maturity of known functional abstraction methods is insufficient for structure verification, especially when different design styles are combined in one layout.

2.1.3 Netlist comparison

The netlist comparison method such as in [Ebeli83], [Waten83], [Spick83], [Ebeli88], compares the transistor level netlist extracted from layout with a reference netlist, by proving or disproving graph isomorphism between the netlists. Figure 2.3 shows how netlist comparison is used. The result of netlist comparison is either a cross-reference list or a discrepancy list. For this particular example, the result is a cross reference list. The cross-reference list indicates which elements are isomorphic to one another, as

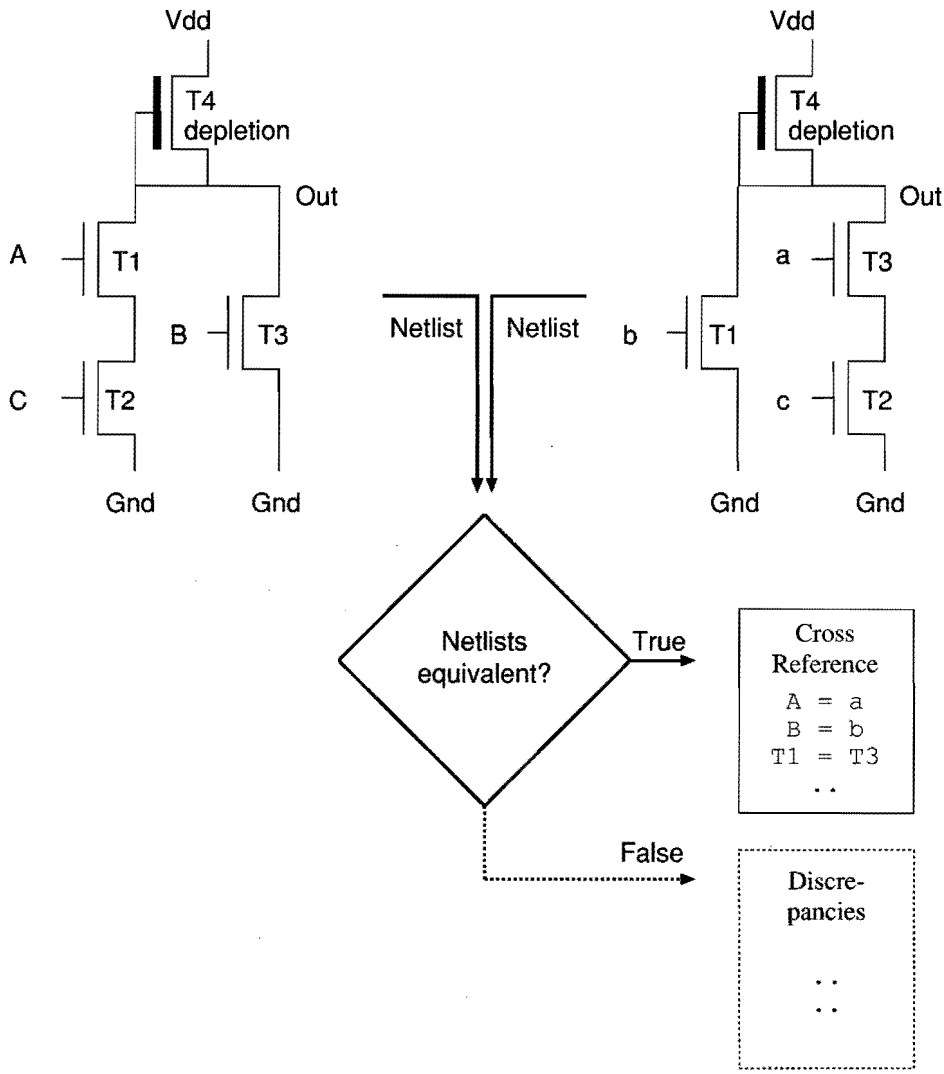


Figure 2.3: *Netlist comparison.*

indicated by Figure 2.3. A discrepancy list indicates the elements that contradict isomorphism between the netlists.

Existing algorithms for proving graph isomorphism are based either on depth-first search [Spick83], or on refinement [Ebeli83]. The aim of both approaches is to derive an isomorphism function ϕ, i.e., a bi-jective mapping from the elements (nets and components) of the reference netlist, to the transistor level netlist, which preserves adjacency and other properties of every element.

Depth-first search

The depth-first search algorithm for sub-graph isomorphisms will be explained extensively in Chapter 4. This paragraph describes briefly the depth-first search version of a graph isomorphism algorithm. The algorithm first defines a search tree that represents the set of all mappings between the netlists. Each path from the root to a leaf of the search tree represents one particular mapping. Next, the depth-first search algorithm constructs an isomorphism function ϕ, by traversing the search tree starting from the root, to determine a path that corresponds to an isomorphism function. The traversal downwards continues until the partial function associated with the current path is inconsistent with preservation of adjacency or other properties of every netlist element. In that case, backtracking occurs to find alternative paths.

Proving graph isomorphism by depth-first search, works appropriately for small netlists. For medium and large netlists, this approach leads to unacceptable run times. Therefore, all modern algorithms for proving graph isomorphism are based on refinement.

Refinement

The principle of refinement (see also [Read77]) is informally explained as follows. First it is established that the number of components and number of elements are equal in both netlists. Next, the elements, i.e., components and nets, in the two netlists are iteratively partitioned into sets of elements having equal properties. The initial partitioning is based on initial properties. The initial properties are defined by local features of the elements, such as the type of a component and the number of connections of a net. In every iteration that follows the initialization, the property of every element is reassigned to a value computed by combining (see [Ebeli83]) the current property and the current properties of the neighbors. In this way charac-

teristics of the neighborhood around every element, at a distance equal to the iteration step number, are combined. The partition can now be refined based on the updated element properties. When both netlists are partitioned into sets of one element (singleton sets), the pair of elements having equal properties in both netlists are assumed to be isomorphic. Unfortunately, it is not always possible to reach a partition consisting of singletons. When a netlist is symmetrical (or to state it exactly: when the number of automorphisms [Harar72.1] is larger than one), refinement into a partition of singletons is not possible. For the example in Figure 2.4, refinement can-

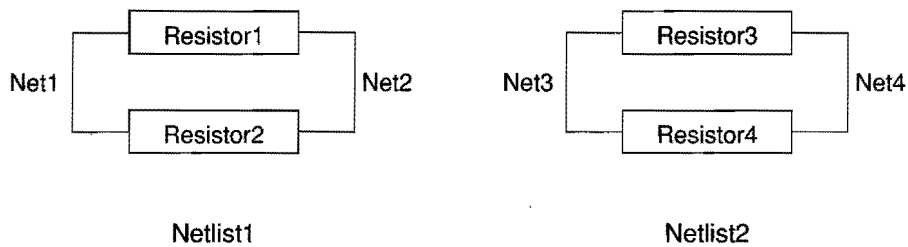


Figure 2.4: *Example for which netlist comparison by refinement fails.*

not be used to determine isomorphism. After initialization, the nets and resistor components of the netlists have exactly the same role, also when the neighborhood is taken into account, so refinement until a partition of singletons is reached is not possible. The refinement algorithm will conclude that netlist1 is not isomorphic to netlist2. This means that when errors are given by the method, based on not reaching two partitions of singletons, the netlist may still be isomorphic. In other words, this method may result in false negatives.

Despite this disadvantage, netlist comparison based on a refinement algorithm is widely used because it is available, and the run times are acceptable. Netlist comparison is often referred to as LVS (Layout Versus Schematics), since the method is often embedded in a graphical CAD-environment. Obviously, next to the extracted transistor netlist, one requires a reference transistor netlist for comparison. This leads us to another weak point of the method, the need for a reference netlist. Often, a complete netlist is not available, or it is copied from the corresponding transistor level in the synthesis trajectory. In the latter case only errors occurring between transistor level and layout are noticed, the synthesis

steps between transistor level and high level netlist remain unverified. As a method to check the transistor level versus the layout however, netlist comparison is very efficient when no errors are found. When a discrepancy is signalled by netlist comparison, the cause of discrepancy may be indicated poorly, as was shown in a recent study [Rovers93]. For example, in a small design of 780 components (see color figures E.1 and E.1a in Appendix E) two wires were accidently interchanged. Instead of reporting that two wires were interchanged, a massive and unstructured error report of 43 pages resulted. Figure E.1 (Appendix E) shows the nets and components mentioned in the report. The figure shows that for a single exchange of wires, in a small layout, errors are indicated at many places in the design. For VLSI designs, this is even worse, due to the size of the netlists at the transistor level.

In [Batra92], a hierarchical netlist comparison program is described. The program uses extra hierarchy information that is manually added in virtual layout layers, to indicate the intended hierarchy. In this way expansion of all structure levels down to the transistor level is partly omitted. The disadvantage of this method, as mentioned by the authors as well, is that the addition of hierarchy information is cumbersome. Additionally, it adds a new source of errors to the design trajectory.

We conclude that the netlist comparison method for structure verification is too restricted, because it needs a reliable reference netlist. Furthermore, the error indication is ineffective, and false negatives are inherent to the main algorithm.

2.2 Literature on structure recognition

This section describes the state of art with respect to sub-circuit recognition. In the 1960s and early 1970s, structure recognition was studied by discrete mathematicians focussing on graph theory. They referred to it as the problem of identifying sub-graph isomorphisms. They showed that the problem of identifying sub-graph isomorphisms is NP-hard [Read77], which lead to very pessimistic views on the possibility of applying sub-graph recognition algorithms [Berzt73]. The exponential growth of computer power in the last decades made worst case computation feasible for small problems. Furthermore, the usefulness of sub-circuit recognition was recognized in various other sciences, including electronics. Especially for rule-based systems, several recognition programs have been developed. However, the effort in

this area was mainly spent on the possible applications. Developing an effective sub-circuit recognition algorithm was usually not the main focus. On the other hand, other work has been published, whose intention is similar to our work, but whose efficiency is limited. The rest of this section describes first some rule-based systems, followed by several other systems.

2.2.1 Rule-based systems

Rule-based systems as described in [Dever92], [Ramme92], [Bolse89], [Rubic84], [Spick88], use structure recognition in addition to functional abstraction (see Section 2.1.2), mainly to check electrical design rules. Although only the structure recognition aspect of these systems is considered here, the intended functionality is more general. The employed recognition methods are all rule-based, consisting of a user-defined set of clauses (goals), and a depth-first search algorithm that tries to find solutions that satisfy these goals. The set of clauses specifies a pattern that represents a sub-circuit. For these methods, sub-circuit recognition is not considered as a single problem, but recognition is directly subdivided into a set of sub-problems, the clauses. This immediate subdivision of the recognition problem fixes the search order of depth-first search algorithms, and the efficiency of the search process strongly depends on the incidental ordering of the clauses. Since the problem is not analyzed before applying depth-first search, the order usually results in a bad performance. Other speed improving techniques as will be described in Chapter 4 are absent as well. Therefore, these methods are usually inefficient, and the execution time is very sensitive to the actual definition of a rule, leaving a large responsibility to the user. The results with respect to efficiency are poor, and they are only given for small designs. Regarding another important issue, error indication, little is known, as it is not mentioned in the papers.

We conclude that the efficiency of these systems with respect to structure recognition is insufficient. The merits of these systems are the exploration of applicability of rule-based techniques. We have described structure verification by hierarchy reconstruction as a possible application for the Vera environment in [Koste88], [Koste89]. The method includes both standard-cell structure recognition and macro-cell structure recognition. We have published more results and details in [Deloor90], [Koste91], and showed a complicated design containing 140 000 transistors that was verified in reasonable time on a common workstation. However, we did not explain any details of the structure recognition algorithm, because a patent

was pending at that time [Koste92.4]. Part of this thesis is therefore devoted to the information missing in these articles.

2.2.2 Other systems

The method for verifying a layout versus a top level structure by means of layout extraction followed by hierarchy reconstruction is nicely described in [Nebel86]. However, the structure recognition algorithm is not very clear. According to the conclusions, the speed of the structure recognition algorithm needs to be improved, and the error indication needs refinement. A corresponding paper [Nebel87] indeed shows that the performance of the structure recognition algorithm is poor. The computational efficiency behaves experimentally as $O(n^2)$, where n denotes the number of transistors in a design.

A standard-cell structure recognition algorithm, to be used for hierarchy reconstruction, is briefly described in [Pelz91]. It is based on a depth-first search algorithm, in which the search order is determined from the signal flow through a MOS-circuit [Jouppi87], and some limited heuristics. In addition, the problem of ambiguity when matching a library (see Section 5.4) is briefly explained. The problem of partly overlapping matches, that may lead to ambiguity, is not mentioned (see also Section 4.51). The work does not include macro-cell recognition and the size of the structures that are recognized seems rather small. Unfortunately, run times are only given for 7 small designs. Five designs contained less than 2 500 transistors, and the largest two contained about 33 000 and 61 000 transistors. Explicit error indication was not mentioned.

Article [Pelz94] is an elaborated version of the previous paper, [Pelz91]. After explaining the hierarchy construction method, the article claims to have proven the following theorem:

> The expected run time complexity of the hierarchy construction method is $O(n.p.j)$, where n is the number of components and nets of the transistor level netlist, p is the average number of components and nets of the sub-structures used for recognition, and j is the number of sub-structures that is being recognized.

The proof given for this claim first reduces the efficiency computation to a formula depending on several characteristics of the sub-structures. Next, 12 designs of the ISCAS '89 benchmark [ISCAS89] are selected and the characteristics are evaluated. Based on the trend of these 12 designs it

is concluded that the claim holds. In my opinion, giving 12 examples for which the theorem holds is not a general proof. Further, the article advocates a hybrid structure verification approach, by combining hierarchy construction, limited high level cell expansion, and netlist comparison, as indicated schematically in Figure 2.5. The expansion is limited down to

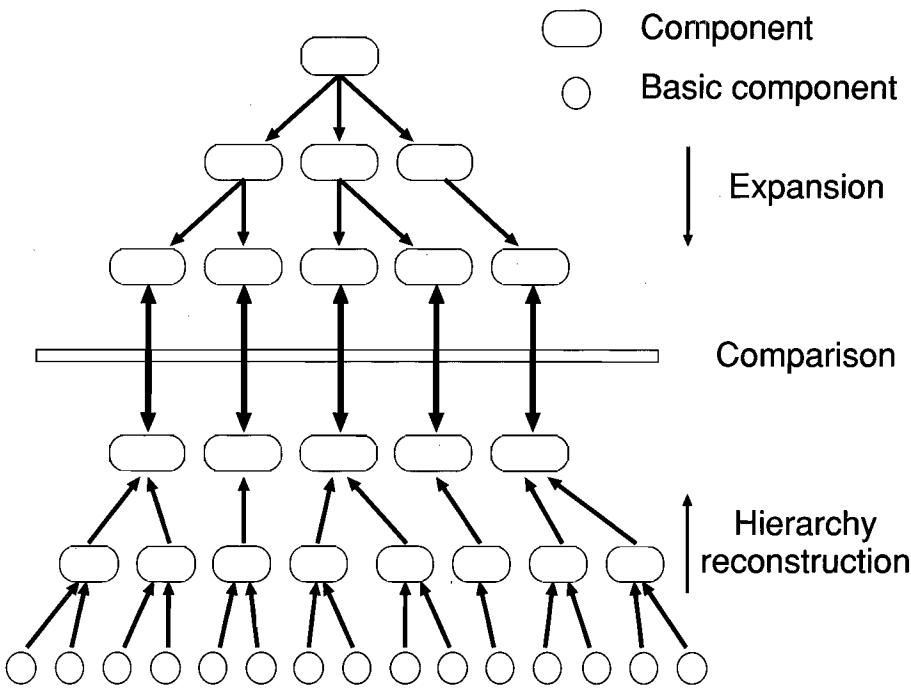


Figure 2.5: *Hybrid approach for structure verification.*

some intermediate level, the hierarchy construction is limited up to the same intermediate level, followed by a comparison of the top-down and bottom-up structures. The author claims to combine the benefits of both methods, i.e., hierarchical error location, the use of different hierarchies for synthesis and verification, and limited run times. The brief results of [Pelz91] are repeated. In addition, the results when applying the hybrid approach for the same designs is given. The hybrid approach is up to 33 % faster. The motivation for a hybrid approach indicates that the author is not satisfied with the performance of the structure recognizer. In my view, a gain of 33 % in run time efficiency is too little to justify the use of such a complicated

method. Perhaps an unmentioned argument for introducing the method is the inability to handle macro-cell recognition. It also remains unclear what level should be selected as the intermediate level at which expansion and hierarchy construction should meet. The error indication becomes very complicated, because the error indication of netlist comparison is weak by itself (Section 2.1.3), and has to be translated back to the original top level structure for interpretation as well. Another disadvantage of the hybrid approach is that one relies on a part of the synthesis phase, the expander, to be correct without checking.

2.3 Conclusion

Several attempts in the past have aimed at tackling the structure verification problem. So far none of the methods in Section 2.1 meet the requirements as stated in Section 1.4. Not only do run time performance and error indication still present problems, but also several methodological questions remain unsolved.

With respect to the structure recognition algorithms for applying a structure reconstruction method (Section 2.2), the performance of the systems is either insufficient, or unknown. Explicit error indication has not been described.

For the structure recognition based systems, the best results have been reported by the author in [Deloor90], [Koste91], but the recognition algorithm was not described. Of the other papers, [Pelz91] and [Pelz94] are the most interesting, although only results for small designs are given, macro-cell hierarchy reconstruction was not included, and no explicit error indication was mentioned.

Chapter 3

The hierarchy reconstruction method

This chapter describes the hierarchy reconstruction method. After giving a global introduction, the required information and the tools needed to make the method operational are derived from existing hierarchy constructs. The requirements are summarized and ordered into an operational model. The model shows that in addition to a sub-circuit recognizer, other tools are also needed. The remaining chapters of this thesis focus on the implementation of the operational model and on the results obtained with the implementation.

3.1 Introduction

Hierarchy reconstruction aims at verifying consistency between the transistor level structure and a high level structure. The transistor level structure has been extracted from a layout, as described in Section 1.3. The high level structure consists of standard cells and macro-cells. In order to simplify terminology, we also call a fixed analog block a standard cell. Macro-cells are instances of parameterized modules. For example, an n-bits adder is a module with parameter n, and a 7-bits adder is a macro-cell, generated by the adder's module generator instantiated with n=7. Standard cells are not parameterized with respect to their structure. The hierarchy reconstruction method is based on stepwise bottom-up abstraction. By using a sub-circuit recognizer, the simplest sub-structures in the transistor netlist, such as inverters, nands, etc., are identified first. Next, the higher level

sub-structures, such as memory-cells, etc., are found, as indicated in Figure 3.1. Recognition and abstraction of increasingly complex structures takes place, until no further abstraction is possible, and the highest level structure has been reached. By using netlist comparison at the top level, the constructed high level structure description can now be compared (see Figure 1.3) with the initial top level structure, to establish consistency.

The following information is needed for the construction process:

- a non-parameterized component library describing higher level components as a network of connected lower level components, and
- a parameterized module library, describing how a module instance (a macro-cell) is composed of interconnected lower level components, for given parameter instances.

In these libraries, layout related information is not included. The following tools are needed as well to be able to perform the construction process:

- a fast sub-circuit recognizer for performing structure recognition,
- a netlist comparison tool, and
- a tool, called the controller, that supervises the reconstruction process.

The latter should interpret the library information, and order the sequence of structures to be recognized.

Since there is no layout information required for hierarchy reconstruction, the effort to set up the library is limited. Compared to the effort spent on making (writing) a module generator and a standard cell library, our library effort is negligible. Since the information is set up differently and independently from the top-down library and synthesis tools, the probability of errors slipping through unnoticed is very small. In the remaining sections of this chapter, the method is elaborated into an operational model, that shows the prerequisites and their relations that must be implemented.

3.2 Hierarchy and structure parameters

This section describes the hierarchy reconstruction method in more detail by considering the role of structure parameters in a design hierarchy. Depending on the role of structure parameters, hierarchy is partitioned into

four categories. The requirements for constructing hierarchy are derived for each category. As indicated in the introduction of this chapter, the main distinction is between macro-cells that are instances of parameterized modules, and standard cells that are not parameterized. In theory, little can be said about the semantics of the structure parameters of a module. However, for a module to be usable, the semantics of a structure parameter should not be complicated. Therefore, our partitioning is based on actual semantics of parameters that are used in IC-design [Wouds90]. In all cases, parameters indicate either repetition of some structure, or a function, represented by a table.

Category 0: Non-parameterized modules
A non-parameterized module is a module that has a fixed structure. The reconstruction of non-parameterized modules such as standard cells is the first, and most important, step in reducing the complexity of a large circuit. For example replacing all transistor pairs forming an inverter circuit, by an inverter component can be performed as follows:

1. retrieve the inverter structure, to act as the template,

2. identify all transistor pairs matching the template,

3. replace the identified transistor pairs by inverter components.

After the matched inverter transistors have been abstracted, other or higher level components can be constructed in the same way. Figure 3.1 illustrates the hierarchy construction steps of an inverter structure, followed by the construction of a memory cell.
In summary, hierarchy construction for Category 0 modules requires

(a) a library, containing the structures of all non-parameterized (sub-)modules,

(b) a pattern-matcher, capable of finding matches of a given library template in a large network, and

(c) an "abstractor", to replace found matches by the corresponding higher level component.

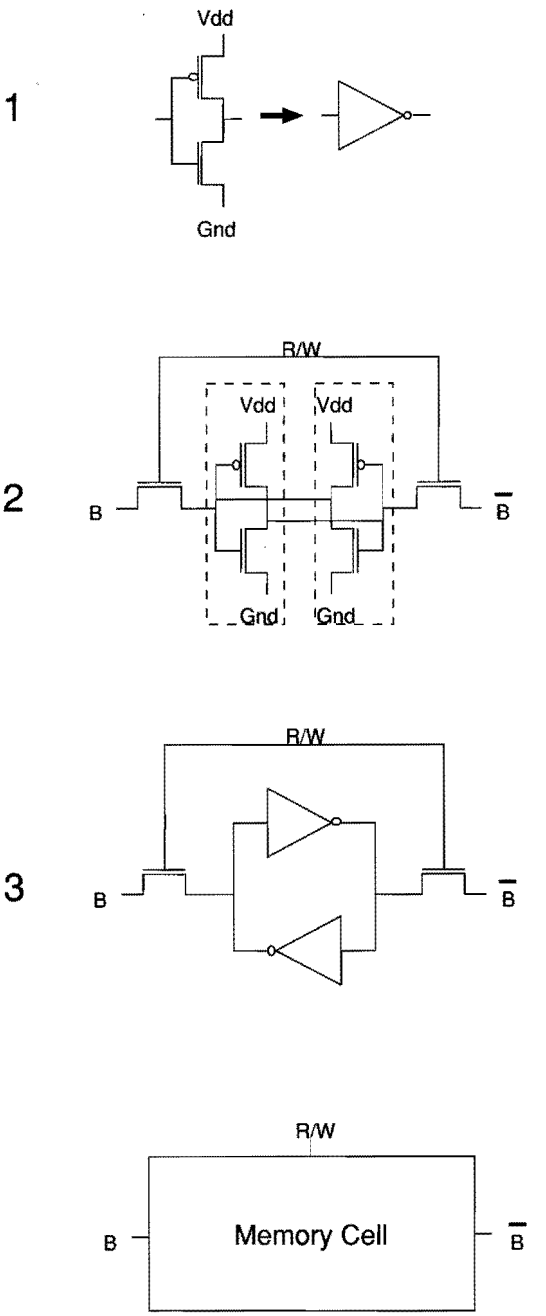


Figure 3.1: *Non-parameterized hierarchy reconstruction*

Category 1: Singly parameterized modules
A singly parameterized module is a module of which the structure of a corresponding macro-cell is determined by the value of one structure parameter, k, that indicates structure repetition. The parameter value is a number of a limited integer domain. When two values are allowed only, the parameter indicates the absence or presence of some part of the structure. When the parameter has more than two acceptable values, it indicates serial or parallel repetition of some part of the structure. Obviously, when the parameter has one possible value, the parameter has no meaning for the structure of the module. The parameter may be associated with the number of bits in a data or control word. The hierarchy construction process of this category is explained for the abstraction of all memory words in a memory core (first step in Figure 3.2). After the reconstruction of the memory cells (see Category 0), the following is performed:

1. the value k, i.e., the number of memory cells connected in parallel to the same r/w select line, is derived from the network,

2. a parameterized module generator produces a structure template for the k-bit memory word instance,

3. the circuit components, matching this template, are replaced by a k-bit memory word.

Compared to Category 0 modules, Category 1 modules require:

(d) a structure-repetition detector, capable of recovering parameter values from the repetition in a network,

(e) a library of parameterized structure template generators.

Category 2: Multiple parameterized modules
A multiple parameterized module is a module of which the structure of a macro-cell is determined by multiple parameter values. A RAM module is an example having multiple parameters. The structure of a RAM macro-cell may be a function of four parameters: x-decoder depth, y-decoder depth, z-decoder depth and word-length. Hierarchy reconstruction of these modules is performed by repeating singly parameterized hierarchy reconstruction. Figure 3.2 illustrates the reconstruction process of the core of a RAM. The number of bits in a word is determined first from the number of memory cells connected to the r/w select lines. All memory words can now be

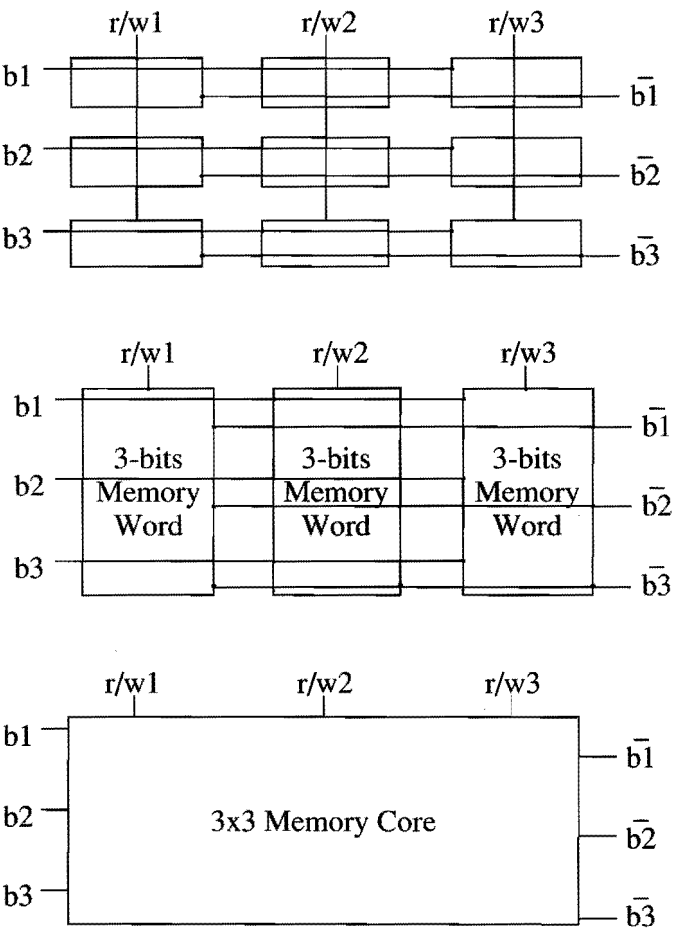


Figure 3.2: *Parameterized hierarchy reconstruction*

abstracted, as described for Category 1 modules. Secondly, the memory core depth, i.e., the number of memory words, is determined by the number of parallel connections to both bit line {b1, b2, ...} and inverse bit line $\{\overline{b1}, \overline{b2}, \ldots\}$ signals. The complete memory core can now be abstracted. Compared to Category 1 modules, Category 2 modules have no additional requirements.

Category 3: Modules having a table parameter
This category is defined by modules that have a table parameter. Typical examples of this category are ROM and PLA modules. Their functionality is determined by a (truth) table. As an example, the reconstruction of ROM modules is briefly described. The ROM module is implemented as a row decoder, a column decoder and a core. The data contents in the core of the ROM are represented by the presence or absence of a transistor at the crosspoint of a word and a bit line. The row decoders and column decoders can be reconstructed in the same way as Category 2 modules. In addition to Category 2 modules, Category 3 modules require:

(f) a table-extractor, to retrieve the function of the macro-cell.

3.3 The operational model

In Figure 3.3 the requirements derived in the previous section are summarized in the operational model of Figure 3.3. The operational model indicates the tools, libraries and relations between them to transform the transistor level structure, or circuit, into a top level structure. The cell library, at the left-hand side of Figure 3.3, stores component definitions as a non-parameterized or a parameterized structure of connected lower level components. The tools that operate on the circuit, shown at the right-hand side of Figure 3.3, are a sub-circuit recognizer, an abstractor, a structure-repetition detector and a table extractor. The sub-circuit recognizer identifies occurrences of a circuit pattern, called a template, the abstracter replaces a match of a template by the corresponding higher level template component, the structure repetition detector identifies iterative structure parameter values, and finally the table extractor derives the function related to table parameter of a module.

The controller organizes the interaction between the tools, the library and the circuit. For hierarchy reconstruction of non-parameterized modules, only the objects connected by bold lines are needed. The order of

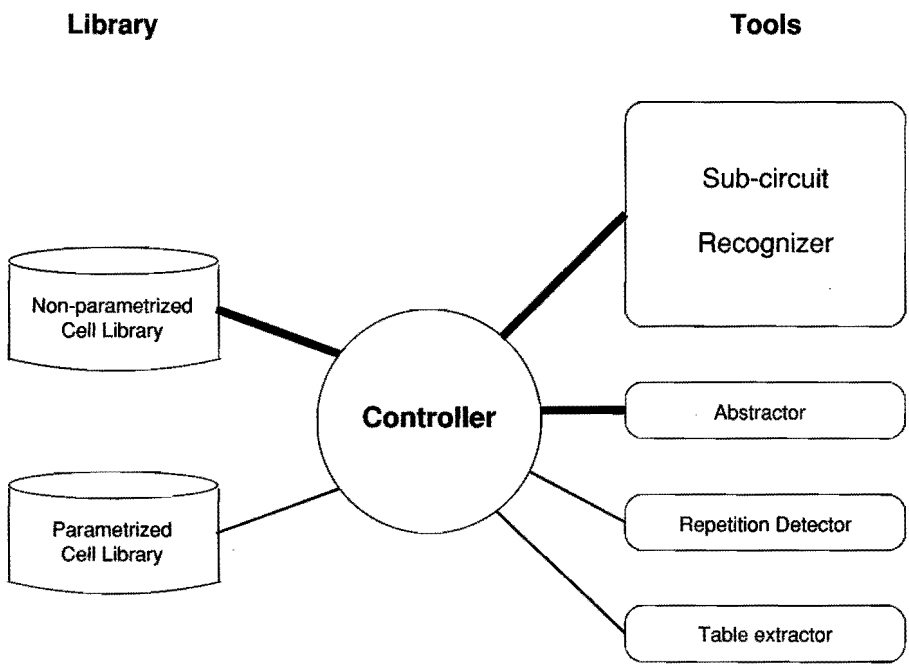


Figure 3.3: *Operational model for hierarchy reconstruction*

abstraction is determined by the controller. For parameterized modules, the other objects are needed as well. For every module generator, a verification controller is created, which describes how a macro-cell can be reconstructed. By activating the associated controller, the user starts the reconstruction process of all instances of that specific module generator in the design. The controller determines the order in which the tools operate on the network, and gathers and provides the necessary information.

Control and libraries are set up separately and independently from the module generator, since they require a different view, namely bottom-up instead of top-down. By separating the information used in the top-down and bottom-up path, the probability of the same error occurring in both descriptions, thus escaping detection, is very small.

In the following chapters, the model of Figure 3.3 is elaborated. Some interesting tools are elaborated in more detail than other, less interesting tools. The sub-circuit recognizer is the most important tool to make hierarchy reconstruction operational. As described in Chapter 2, a suitable sub-circuit recognition tool has not yet been presented. The next chapter describes our sub-circuit recognition tool. Since the subject is subtle but crucial for the applicability of hierarchy reconstruction, it is described thoroughly, starting from formal definitions, and explaining the algorithm in detail, including the crucial efficiency enhancements. Chapter 5 describes the implementation of the remainder of the operational model, followed by Chapter 6, showing the results of the method for a typical design.

Chapter 4

The sub-circuit recognizer

4.1 Introduction

In the previous chapter, it has been shown that a sub-circuit recognizer is the main tool needed for making the hierarchy reconstruction method operational. This chapter discusses the sub-circuit recognizer in detail. When the casual reader is interested in the hierarchy reconstruction method, but not that much in the algorithms supporting it, this chapter can be omitted.

The sub-circuit recognition problem is informally described as the problem of finding all occurrences of a *template circuit* in a usually larger *main circuit*. Sub-circuit recognition is identical to the problem of finding all isomorphic sub-graphs in a graph [Read77]. In the mathematical literature it is an example of an NP-hard problem, implying that no method exists that solves each instance of the problem in polynomial time [Read77]. Every algorithm shows exponentially growing run times for some set of problem instances, unless the very unlikely condition known as "P = NP" holds. Therefore, the hope of finding a useful algorithm had vanished for some mathematicians [Berzt73]. From a practical point of view however, it still makes sense to search for algorithms that efficiently solve many often encountered sub-graph isomorphism problem instances, although no guarantee of run times can be given. In fact, without a powerful sub-circuit recognizer, the hierarchy reconstruction method cannot be employed.

The remainder of this chapter is organized as follows. Section 4.2 defines the sub-circuit recognition problem in a formal way. Section 4.3 describes the primary algorithm. Section 4.4 describes the post processing of the results of the primary algorithm. Section 4.5 shows some extensions that

enhance the usability and flexibility of the primary algorithm. Section 4.6 describes diagnosis feedback when the recognition finds fewer matches than expected. Section 4.7 shows experimental results and an analysis of the run times. Section 4.8 finishes the chapter with conclusions.

4.2 Definitions

Section 4.2.1 enumerates notational conventions and several general notions. Section 4.2.2 defines formally a circuit. Based on the definition of a circuit, Section 4.2.3 defines the template circuit and the main circuit, followed by the sub-circuit recognition problem. To allow accurate efficiency argumentation for the algorithms in the next chapters, Section 4.2.4 briefly describes the data representation of a circuit in a computer.

4.2.1 General notions and notation

A *set* is a collection of elements, in which each member occurs once.

A *multi-set* is a collection in which elements may occur multiple times.

The *multiplicity* of an element a of a multi-set B, i.e., the number of occurrences of a in B, is denoted by $\mu_a(B)$.

For a (multi-)set B, $|B|$ denotes the number of elements in B.

For a set B, 2^B denotes the set of all sets over B, i.e., the *power set* of B.

For a set B, $\mathbb{N}^B$ denotes the set of all multi-sets over B.

An *ordered (multi-)set* is denoted by $(x_1, x_2, \ldots)$. A shorthand notation for the ordered (multi-)set is $\mathbf{x}$, i.e., by using boldface fonts. $\mathbf{x}_j$ denotes the *prefix* $(x_1, \ldots, x_j)$. An ordered multi-set is also called a *sequence*.

For a set B, B^+ denotes the set of all non-empty sequences over B.

An *unordered (multi-)set* is denoted by $\{x_1, x_2, \ldots\}$.

The *set operators* for union, intersection and set-minus are denoted by the symbols $\bigcup$, $\bigcap$ and $\setminus$. The result of a set operator is an unordered set. An operand is either an unordered set, or it is interpreted as an unordered set, when the operand is an (un)ordered multi-set or an ordered set.

For sets A, B and a function F, $F : A \rightarrow B$, A is called the *domain* and B is called the *co-domain* of function F. Furthermore, $F(A)$ denotes a subset of B called the *image*.

For sets A, B, B^A denotes the set of all functions having domain A and co-domain B.

In definitions, the first character of the notion being defined are written uppercase.

Definition 4.1 *Restriction of a Function*
For a function F, $F : A \rightarrow B$, and a subset C of A, the Restriction of Function F to C is the function denoted by $F|_C$, defined as $F|_C : C \rightarrow B$, for all $a \in C : F|_C(a) = F(a)$. □

Definition 4.2 *Equivalence Set*
For $a, b \in A$, a is called equivalent to b when $F(a) = F(b)$. The Equivalence Set of $a \in A$ with respect to the function $F : A \rightarrow B$, denoted by $[a]_F$, is defined by $[a]_F = \{x \in A \mid F(x) = F(a)\}$. □

Definition 4.3 *Quotient Set*
For a function $F : A \rightarrow B$, the Quotient Set A/F is defined by $A/F = \{[a]_F \mid a \in A\}$. A/F is a partition of set A. □

Definition 4.4 *Canonical Map*
For a function $F : A \rightarrow B$, the Canonical Map $g : A \rightarrow A/F$ is defined by $g(a) = [a]_F$. Hence, the canonical map maps an element onto the equivalence set of which it is a member. □

Definition 4.5 *Characteristic Function*
For a function $F : A \rightarrow \{True, False\}$, F subdivides set A into the equivalence sets $[True]_F$ and $[False]_F$, called the true-set and false-set. Therefore, $A/F = \{[True]_F, [False]_F\}$. Since F can be used to define a set and its complement, F is called the Characteristic Function of set $[True]_F$. □

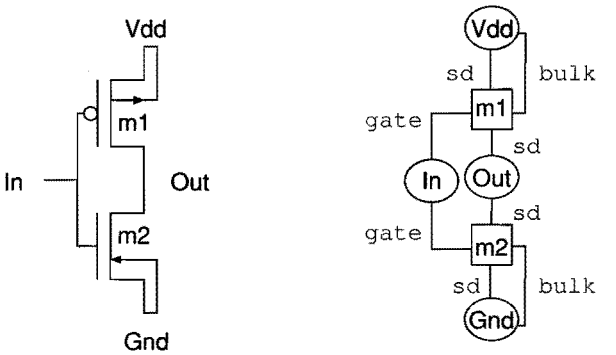
The true-set of a characteristic function $D : A \rightarrow \{True, False\}$, i.e., $[True]_D$, is denoted by $\mathcal{D}$. The true-set of a characteristic function $d : A \rightarrow \{True, False\}$, i.e., $[True]_d$, is denoted by δ. Hence calligraphic fonts are used for the true-set of a function denoted in uppercase, and Greek fonts are used for functions denoted in lowercase.

Definition 4.6 *Pair Function*
For $k \in \mathbb{N}$, ordered sets $A = (a_1, \ldots, a_k)$, $B = (b_1, \ldots, b_k)$, the Pair Function $A \bullet B : A \rightarrow B$ is defined by $A \bullet B(a_i) = b_i$ for $i = 1, \ldots, k$. □

The (multi-)sets and functions of the *main circuit* are denoted in boldface, for example $\mathbf{G} = (\mathbf{V}, \mathbf{T}, \mathbf{A}, \mathbf{E}, \mathbf{TC})$. The (multi-)sets and functions of a *template circuit* are denoted emphasized, for example $G = (V, T, A, E, TC)$. The (multi-)sets and functions of a *sub-circuit* of the main circuit are denoted calligraphically, for example $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{A}, \mathcal{E}, \mathcal{TC})$.

For an undirected edge e, $vert(e)$ denotes the unordered pair of vertices $\{u, v\}$ that are connected by e.

4.2.2 The circuit definition




```
(
   (MOS m1 vdd in out vdd 2e-6 2e-6 pmos)
   (MOS m2 out in gnd gnd 1e-6 2e-6 nmos)
)
```

Figure 4.1: *Schematics, graph picture and netlist representation of a circuit. The attribute names and attribute values of m1 and m2 are not included in the graph.*

An example

Figure 4.1 shows an example of a circuit. The figure displays the schematics, a picture of the circuit as a bipartite graph, and a netlist representation of a circuit. The schematic representation is normally used by electronic engineers. Since the circuit will formally be defined as a labeled bipartite multi-graph, the graph picture represents best the formal circuit definition. A netlist representation is a textual representation that can be used to store

a circuit in a computer file. Before formally defining a circuit, the relation between schematics, the picture of the graph and the netlist is explained. The schematics of Figure 4.1 consist of transistor components m1 and m2, and nets vdd, in, out and gnd. In the picture of the circuit graph, the components are depicted by squares, the nets are depicted by ovals. Together, the squares and ovals are the vertices of the graph. The connections of the schematics are depicted by lines, determining the undirected edges of the graph. Note that the graph is bipartite since ovals are only connected to squares and vice versa. Note also that the graph contains multi-edges, such as the connections between component m1 and net vdd. In order to distinguish between different kinds of connection, each edge of the graph is labeled with a terminal class. Both m1 and m2 are instances of the component type MOS. A component type defines the properties (property name and property value pairs) of a component, such as the number and kind of connections, the possible attribute names, etc. Figure 4.2 shows the MOS component type definition. The list of terminal classes defines the connections of MOS, i.e., one terminal of class gate, two terminals of class sd - short-hand for "source or drain", which are considered equivalent terminals - and one terminal of class bulk. The three attributes of MOS are named width, length and model. The attributes are not included in the graph of Figure 4.1. Figure 4.3, shows another component type definition, having other properties in addition to those of Figure 4.1.

The netlist representation is component oriented. It enumerates the component type, the component name, the connections to nets and attribute values corresponding to the attribute names. The connections and attribute values are ordered according to the terminal classes and attribute names found in the description of a component type; see Figure 4.2. For example, the last attribute, named model, is used to indicate a pmos attribute value for m1, and nmos for m2.

The formal definition of a circuit

The definitions are based on the following basic notions:

- τ denotes a non-empty set of types,
- Γ denotes a non-empty set of terminal classes,
- α denotes a set of attribute names,
- β denotes a set of attribute values.

```
(MOS
      (  Terminal-classes   (sd gate sd bulk)
         Attribute-names    (width length model)
      )
)
```

Figure 4.2: *Example of a component type, defining TTC(MOS) = (sd, gate, sd, bulk) with property name* `Terminal-classes` *and TA(MOS) = (width, length, model) with property name* `Attribute-names`

The empty set, denoted by $\emptyset$, is not a member of τ, Γ, α or β. A circuit will be defined as an interconnected set of components and nets. Every component will have a type label, which will be defined by the type function. The "type terminal classes" function and the "type attributes" function will be defined on types, to prescribe the labels of connections to a component and the attribute names of a component. A component has a second label, the attribute, that will be defined by the attribute function to assign attribute values for corresponding attribute names. A net will not have an associated type label. The distinction between components and nets will therefore be defined based on the type function. The edges between components and nets will be labeled by a terminal class. The multi-set of terminal classes that are labels of the edges between two vertices will be defined as the terminal classes function which will used extensively in Section 4.2.3.

Definition 4.7 *Type Terminal Classes*
For each type $t \in \tau$, the Type Terminal Classes function $TTC : \tau \rightarrow \Gamma^+$ assigns to each type a sequence of terminal classes. □

For example, in Figure 4.2, TTC(MOS) = (sd, gate, sd, bulk). Note that a sequence is an ordered enumeration of items in which repetition may occur.

Definition 4.8 *Type Attributes Function*
For each type $t \in \tau$, the Type Attributes Function $TA : \tau \rightarrow 2^{\alpha}$ assigns a set of attribute names to a type. □

For example, in Figure 4.2, TA(MOS) = (width, length, model).

Definition 4.9 *Type Function*
For a set of vertices V, the Type Function $T : V \rightarrow \tau \cup \{\emptyset\}$ assigns either a type or $\emptyset$ to a vertex. □

Definition 4.10 *Set of Components*
For a set of vertices V and a type function T, the Set of Components C is defined by $\{ v \in V \mid T(v) \in \tau \}$. □

Definition 4.11 *The Set of Nets*
For a set of vertices V and a type function T, the Set of Nets N is defined by $\{ v \in V \mid T(v) = \emptyset \}$. □

Obviously, $N = V \setminus C$, since $\emptyset \notin \tau$.

Definition 4.12 *Attribute Function*
For a set of vertices V and a type function T, the Attribute Function $A : V \rightarrow 2^{\alpha \times \beta}$ assigns to each component a set of ordered attribute name, attribute value pairs, and $\emptyset$ to a net. For a component $v \in C \subset V$, every attribute name of $A(v)$ must be a member of $TA(T(v))$, i.e., a member of the attribute names of the corresponding type of v, and only one attribute value is associated. □

For example, $A(m1) = \{(width, 1e-6), (length, 1e-6), (model, pmos)\}$ in Figure 4.1.

Definition 4.13 *Set of Multi-edges*
For a vertex set V and a type function T, an edge connects a component $u \in C \subset V$ and a net $v \in N \subset V$. The connected vertices $\{u, v\}$ of an edge e are denoted by $vert(e)$. A Set of Multi-edges is a set of edges for which several edges may connect the same vertices. □

Definition 4.14 *Terminal Class Function*
For a set of multi-edges E, the Terminal Class Function $TC : E \rightarrow \Gamma$ assigns a label to each edge, called the terminal class. It indicates what kind of connection is meant. □

Definition 4.15 *Terminal Classes Function*
For a set of vertices V, a set of multi-edges E and a terminal class function TC, the Terminal Classes Function $TCS : V \times V \rightarrow \mathbb{N}^{\Gamma}$ is defined by $TCS(u, v) = \{ TC(e) \mid e \in E : vert(e) = \{u, v\} \}$. It assigns the multi-set of all terminal class labels to a vertex pair $\{u, v\}$. Hence, it represents the edges connecting u and v, including their multiplicity. When $TCS(u, v) = \emptyset$, no edges connect u and v. Since an edge is undirected, TCS is a symmetric function. □

In Figure 4.1 for example, we see two edges connecting vdd and m1, having labels *sd* and *bulk*, so $TCS(vdd, m1) = \{sd, bulk\}$.

Definition 4.16 *Degree Function*
For a set of vertices V, a set of multi-edges E and a terminal class function TC the Degree Function $DEGREE : V \times \Gamma \rightarrow I\!N$ assigns to a vertex v and a terminal class c the number of edges incident with v having a terminal class label c. □

Definition 4.17 *Circuit*
A Circuit G is an undirected labeled bipartite multi-graph, defined by a 5-tuple $G = (V, T, A, E, TC)$, for which

- V is a set of vertices,
- T is a Type function as defined in Definition 4.9,
- A is an Attribute function as defined in Definition 4.12,
- E is a set of multi-edges as defined in Definition 4.13,
- TC is a Terminal Class function as defined in Definition 4.14.

The set of components C is defined in Definition 4.10. The set of nets N is defined in Definition 4.11. The degree function $DEGREE$ is defined in Definition 4.16. The terminal classes function of G is defined by Definition 4.15. Furthermore, for any component $v \in C \subset V$, any terminal class $c \in \Gamma$ the following must hold:

$$DEGREE(v, c) = \mu_c(\ TTC(\ T(v)\ )\ ). \tag{4.1}$$

In other words, the number of edges per terminal class of a component v is determined by the type of the component v. The *size of the circuit*, denoted by $|G|$, is defined by the number of edges plus the number of vertices, i.e., $|E| + |V|$. □

4.2.3 The sub-circuit recognition problem

The sub-circuit recognition problem aims at finding all occurrences of a template circuit, called the matches, in a usually larger main circuit. Every match is a sub-circuit of the main circuit. The sub-circuit recognition problem will be defined slightly more specifically than the abstract sub-graph

isomorphism problem. In the first place, the general sub-graph isomorphism problem is usually described for unlabeled graphs. In our case, the edges of a graph are labeled by the terminal class function (see Definition 4.14), and the vertices have two labels, defined by the type function (see Definition 4.9) and the attribute-function (see Definition 4.12). The labels of matching sub-circuits must correspond to the labels of the template circuit. Secondly, for a template circuit we want to distinguish between *external nets* whose match may have more connections than specified, and *internal nets* whose match must have the same connection pattern.

The primary sub-circuit recognition problem will be defined by successively defining the main circuit, the template circuit, isomorphism functions, and a sub-circuit. Next, the relation between the isomorphism functions and the solution set of problem, called the matches, is described. An example of a template finishes the problem definition.

Definition 4.18 *Main Circuit*
The Main Circuit is defined as a circuit $\mathbf{G} = (\mathbf{V}, \mathbf{T}, \mathbf{A}, \mathbf{E}, \mathbf{TC})$, as defined in Definition 4.17 of Section 4.2.2. The component set and net set of $\mathbf{G}$ are denoted by $\mathbf{C}$ and $\mathbf{N}$, respectively. The terminal classes function of $\mathbf{G}$ is denoted by $\mathbf{TCS}$. The degree function of $\mathbf{G}$ is denoted by $\mathbf{DEGREE}$. □

Note the use of a boldface font for the notions relating to the main circuit.

Definition 4.19 *Template Circuit*
The Template Circuit is defined by

- a connected non-empty circuit $G = (V, T, A, E, TC)$ as defined in Definition 4.17, and
- a subset of the net set of G, NE, called the external net set.

The component set and net set of G are denoted by C and N, respectively. The set $V \setminus NE$, denoted by NI, is called the internal net set. The terminal classes function of G is denoted by TCS. The degree function of G is denoted by $DEGREE$. □

Note the use of an emphasized font for the notions relating to the template circuit.

Definition 4.20 *Isomorphism Predicate*
For a main circuit $\mathbf{G} = (\mathbf{V}, \mathbf{T}, \mathbf{A}, \mathbf{E}, \mathbf{TC})$, a template circuit $G = (V, T, A, E, TC)$ with external net set NE, the Isomorphism Predicate

$$\mathbf{S} : \mathbf{V}^V \rightarrow \{True, False\}$$

is defined by $\mathbf{S}(\phi) = True$ for any function ϕ, $\phi : V \rightarrow \mathbf{V}$ if and only if the conditions

$$\phi \text{ is one-to-one,} \tag{4.2}$$

$$\forall c \in V : \ T(c) = \mathbf{T}(\phi(c)), \tag{4.3}$$

$$\forall c \in V : \ A(c) = \mathbf{A}(\phi(c)), \tag{4.4}$$

$$\forall e \in E, \ vert(e) = \{u, v\} \ : \ TCS(u, v) = \mathbf{TCS}(\phi(u), \phi(v)), \tag{4.5}$$

$$\forall n \in NI, \forall c \in \Gamma : \ DEGREE(n, c) = \mathbf{DEGREE}(\phi(n), c), \tag{4.6}$$

$$\forall n \in NE, \forall c \in \Gamma : \ DEGREE(n, c) \leq \mathbf{DEGREE}(\phi(n), c) \tag{4.7}$$

hold. When $\mathbf{S}(\phi) = True$, ϕ is called an *isomorphism*[1]. $\mathbf{S}$ is a characteristic function whose true-set is the set of all isomorphisms, denoted by $\mathcal{S}$. □

Conditions 4.3 and 4.4 require that both the type and the attribute name, attribute value pairs of the matched components and corresponding template components are equal. The type and the attribute name, attribute value pairs of a net are equal by definition (see Definition 4.11, Definition 4.12). Condition 4.5 requires the existence of a separate equally labeled main circuit edge for each labeled template edge. Condition 4.6 requires that mappings of internal nets in $\mathbf{G}$ are exclusively connected to components as specified by the template, whereas Condition 4.7 requires that mappings of external nets in $\mathbf{G}$ have at least connections as specified by the template G.

Definition 4.21 *Sub-circuit*
For a main circuit $\mathbf{G} = (\mathbf{V}, \mathbf{T}, \mathbf{A}, \mathbf{E}, \mathbf{TC})$, a subset $\mathcal{V}$ of $\mathbf{V}$, the Sub-circuit $\mathbf{G}|_{\mathcal{V}}=(\mathcal{V}, \mathcal{T}, \mathcal{A}, \mathcal{E}, \mathcal{TC})$ is defined by

$$\mathcal{T} = \mathbf{T}|_{\mathcal{V}}, \tag{4.8}$$

$$\mathcal{A} = \mathbf{A}|_{\mathcal{V}}, \tag{4.9}$$

[1]Strictly speaking, ϕ is called an isomorphism only when $|V| = |\mathbf{V}|$. When $|V| < |\mathbf{V}|$, ϕ is called a *monomorphism*. Monomorphism, isomorphism and other notions are described as a special case of the homomorphism in [Stanat77]. According to Definition 4.22, for an isomorphic sub-circuit $\mathbf{G}|_{\phi(V)}=(\mathcal{V}, \mathcal{T}, \mathcal{A}, \mathcal{E}, \mathcal{TC})$, the function $\phi' : V \rightarrow \mathcal{V}$, defined by $\phi'(v) = \phi(v)$ for each $v \in V$, is an isomorphism for which $|V| = |\mathcal{V}|$ with respect to G and $\mathbf{G}|_{\phi(V)}$. So when ϕ is a monomorphism, an isomorphism can always be defined by changing the co-domain $\mathbf{V}$ into $\phi(V)$. Therefore, we ignore the difference between monomorphism and isomorphism.

$$\mathcal{E} = \{e \in \mathbf{E} \mid vert(e) = \{u, v\} \wedge u, v \in \mathcal{V}\}, \tag{4.10}$$

$$\mathcal{TC} = \mathbf{TC}|_{\mathcal{E}}. \tag{4.11}$$

Obviously, $\mathbf{G}|_{\mathcal{V}}$ is a circuit. The component set and net set of $\mathbf{G}|_{\mathcal{V}}$ are denoted by $\mathcal{C}$ and $\mathcal{N}$, respectively. The terminal classes function of $\mathbf{G}|_{\mathcal{V}}$ is denoted by $\mathcal{TCS}$. The degree function of $\mathbf{G}|_{\mathcal{V}}$ is denoted by $\mathcal{DEGREE}$. □

Note the use of a calligraphic font for the notions relating to a sub-circuit. According to Equations 4.8, 4.9, 4.11, the functions in sub-circuit $\mathbf{G}|_{\mathcal{V}}$ are equal to the equivalent functions in $\mathbf{G}$, when restricted to $\mathcal{V}$. The set of multi-edges $\mathcal{E}$ (see Equation 4.10) is defined by the subset of multi-edges $\mathbf{E}$ that connect any two vertices of $\mathcal{V}$.

Definition 4.22 *Isomorphic Sub-circuit*
For a main circuit $\mathbf{G} = (\mathbf{V}, \mathbf{T}, \mathbf{A}, \mathbf{E}, \mathbf{TC})$, a template circuit $G = (V, T, A, E, TC)$ with external net set NE, their isomorphism predicate $\mathbf{S}$, isomorphism function $\phi : V \rightarrow \mathbf{V}$, the Isomorphic Sub-circuit is defined by $\mathbf{G}|_{\phi(V)}$. □

Since ϕ is one-to-one (Equation 4.2), $|\mathcal{V}| = |V|$. Since the number of edges connected to a component is fixed (see Equation 4.1), $|\mathcal{E}| = |E|$. The sub-circuit recognition is defined next.

Definition 4.23 *Sub-circuit Recognition Problem*
For a main circuit $\mathbf{G} = (\mathbf{V}, \mathbf{T}, \mathbf{A}, \mathbf{E}, \mathbf{TC})$, a template circuit $G = (V, T, A, E, TC)$ with external net set NE, the isomorphism set $\mathcal{S}$ as defined in Definition 4.20, the Sub-circuit Recognition Problem is to find the set of matches $\mathcal{M}$ defined by

$$\mathcal{M} = \{\mathbf{G}|_{\phi(V)} \mid \phi \in \mathcal{S}\}.$$

□

In other words, the solution to the sub-circuit recognition problem is given by the set of all sub-circuits of the main circuit that are isomorphic to the template circuit. The isomorphism predicate $\mathbf{S}$ implicitly defines the true-set $\mathcal{S}$, and thus the matches $\mathcal{M}$. For a given isomorphism function ϕ, an isomorphic sub-graph $\mathcal{G}$ can easily be constructed according to Definition 4.21.

In summary, the problem of finding all matches of template circuit in a main circuit is equivalent to finding the set of isomorphism functions $\mathcal{S}$ based on the isomorphism predicate. The set of matches can be constructed easily from $\mathcal{S}$.

```
( NOR
    (  Terminal-names     (in1 in2 out)
       Terminal-classes   (in in out)
       Network (   (MOS t1 vdd in1 out vdd 3e-6 ?  ptype)
                   (MOS t2 vdd in2 out vdd 3e-6 ?  ptype)
                   (MOS t3 gnd in1 h gnd 1e-6 ?  ntype)
                   (MOS t4 h in2 out gnd 1e-6 ?  ntype)
               )
       Restrictions (      ()  ; fixed nets
                           ((in1 in2) (out) (vdd) (gnd))
                               ; external nets
                           ()  ; common components
                           T   ; reduce symmetry
                    )
       Global-nets         (vdd gnd)
    )
)
```

Figure 4.3: *The NOR type, describing both the external component view (represented by the property named* `Terminal-classes`*) and internal template view (represented by the properties named* `Network`, `Restrictions`, `Global-nets`*).*

An example of a template circuit

In Figure 4.3, the type NOR is described. One can distinguish an external and an internal view in the NOR type description. The external view is given by the terminal classes property, describing the NOR as one component having two equivalent terminals (the inputs) and one other terminal (the output). The absence of an attribute names property indicates that the NOR component has no attributes. The internal view describes the constituent parts of a NOR, as given by the network, restrictions and global nets properties. When used in a template circuit for recognition, the NOR network property and the NOR restrictions are used. The restrictions of a type should not be confused with the restriction of a function, as defined in Definition 4.1. The template NOR network property has the same format as the main circuit, and describes most of the template. To show some of the flexibility of this entry, the width of every MOS transistor is prescribed in the template network, whereas the length is left unspecified. The restrictions property specifies details of the template circuit not present in the network entry. Most of the restrictions will be explained in Section 4.5.2, but the external nets {in1, in2, out, vdd, gnd} are indicated by the second item in the restrictions. The reason why the external net set is partitioned will be explained in Section 4.5.2. The internal nets of the template are the nets that are not external, such as {h} for a NOR. The terminal names property relates the template level (the internal view) to the component level (the external view), via the consecutive terminal classes. The global nets entry enumerates external nets that are not in the terminal names list.

4.2.4 The internal data representation of a circuit

To represent a circuit in a computer, different data structures may be used (e.g., see [Reingol77]). Depending on the ease of implementation, the memory usage and the corresponding efficiency of the applied graph algorithms, a choice has to be made. For our problem, the *adjacency list representation* fits closely to the recognition algorithm, because it supports fast access to edges related to a vertex. Within this representation, the set of vertices V of a circuit $G = (V, T, A, E, TC)$ is explicitly represented as a set. The set of multi-edges E, however, is represented by using the present classes function and the adjacency function defined as follows.

Definition 4.24 *Present Classes Function*
For a circuit $G = (V, T, A, E, TC)$, the Present Classes Function $\gamma : V \rightarrow 2^{\Gamma}$ is defined as

$$\gamma(v) = \{\ TC(e) \mid e \in E : v \in vert(e)\ \}.$$

The function γ associates a set of terminal classes with each vertex. □

Definition 4.25 *Adjacency Function*
For a circuit $G = (V, T, A, E, TC)$, the Adjacency Function $Adj : V \times \Gamma \rightarrow \mathbb{N}^V$ is defined as

$$Adj(v, c) = \{\ u \mid e \in E : vert(e) = \{u, v\} \wedge TC(e) = c\ \}.$$

The function Adj associates a multi-set of adjacent vertices with each vertex and terminal class. □

So the connectivity and label information that has so far been accumulated into the function TCS (Definition 4.15) is actually stored in the function Adj. From the definitions of Adj, TCS and $DEGREE$ the following can be derived for a circuit $G = (V, T, A, E, TC)$. For every $u, v \in V, c \in \Gamma$:

$$\mu_u(Adj(v, c)) = \mu_c(TCS(u, v)) \tag{4.12}$$

$$DEGREE(v, c) = \mid Adj(v, c) \mid \tag{4.13}$$

$$\mu_u(Adj(v, c)) = \mu_v(Adj(u, c)) \tag{4.14}$$

Equation 4.14 shows the symmetry of the Adj function. The function Adj is stored as a set of argument, value pairs. This implies that each edge is stored twice, firstly in the adjacency list of the connected component, and secondly in the adjacency list of the connected net. For every component vertex v, no storage is required to implement the present classes function γ, since this information can be derived directly from the associated component type's terminal classes, by mapping the sequence $TTC(T(v))$ to a set. For a net vertex, the present classes set is stored explicitly as a list. We can conclude that the memory usage is $O(|V| + 3|E|)$. The advantage of this representation is that at the cost of extra memory usage $(2|E|)$, the related classes and edges of every vertex can be accessed directly with the functions γ and Adj.

4.3 The primary algorithm: backtracking

Based on the definitions of the previous sections, this section describes a backtracking algorithm to find the matches of the sub-circuit recognition problem. There are many methods to find all matches of a given template circuit in a main circuit. We will first discuss a simple brute-force method, that links directly to the definition of the primary sub-circuit recognition problem. After this introduction, a more efficient method will be described that is based on backtracking. After a short description of this well known problem solving method, the sub-circuit recognition problem will be transformed to fit a backtracking approach. Since efficiency is crucial for the recognition process, the main part that follows highlights efficiency critical elements, finally leading to the primary recognition algorithm. In this algorithm, the search order of backtracking plays a major role.

4.3.1 The brute-force approach

The set of isomorphism functions $\mathcal{S}$ (see Definition 4.20) directly leads to all matches of the sub-circuit recognition problem, according to Definition 4.23. Therefore, the aim is to find $\mathcal{S}$, i.e., the true-set of the isomorphism predicate $\mathbf{S}$. The brute-force approach consists of the following two steps to find $\mathcal{S}$.

1. In the first step, the set $\mathcal{Y}$ of all one-to-one functions in $\mathbf{V}^V$, called the candidate set, is generated. Obviously, $\mathcal{S} \subset \mathcal{Y} \subset \mathbf{V}^V$.

2. In the second step, $\mathcal{S}$ is derived from $\mathcal{Y}$ by removing the elements $\phi \in \mathcal{Y}$ that do not obey the isomorphism predicate $\mathbf{S}$.

After Step 1 and Step 2, the matches are now given by the isomorphic sub-circuit (see Definition 4.22) of each isomorphism ϕ in $\mathcal{S}$. Since Step 2 only entails the evaluation of $\mathbf{S}$ (see Definition 4.20), only Step 1 will be described briefly.

Let $\mathbf{o}$ be an ordered set, enumerating all elements in V. Every ordered set $\mathbf{w}$ of $|V|$ elements of $\mathbf{V}$ corresponds to a mapping $\phi \in \mathcal{Y}$ of s and vice versa. Step 1, the generation of $\mathcal{Y}$, can therefore be implemented by enumerating every ordered set $\mathbf{w}$. The number of elements in $\mathcal{Y}$ is equal to the number of ordered sets of $|V|$ elements of $\mathbf{V}$, i.e.,

$$|\mathcal{Y}| = \frac{|\mathbf{V}|!}{(|\mathbf{V}| - |V|)!}. \tag{4.15}$$

This implies that Step 2 has to be applied $|\mathcal{Y}|$ times.

The advantage of this approach is that every isomorphism is identified. The disadvantage is that the order of the algorithm, dominated by the order of the first step, both for typical case and worst case, is equal to $|\mathcal{Y}|$, multiplied by the ordered set size. Hence,

$$O(\text{brute force algorithm}) = O(\frac{|V| \times |\mathbf{V}|!}{(|\mathbf{V}| - |V|)!}. \tag{4.16}$$

Obviously, this algorithm has little practical relevance because of its inefficiency, but a refinement of this method, called *backtracking*, is directly related. For a backtracking algorithm, the test for a successful match (Step 2) is applied during the selection of a candidate set $\mathcal{Y}$ (Step 1). It has the advantage that all isomorphisms are still identified, but unsuccessful sets of candidates in $\mathcal{Y}$ are excluded earlier, which allows far better average run-times than can be expected in the worst case.

4.3.2 Backtracking in general

Backtracking is a well-known method to solve a certain class of search problems. In this section, the backtracking solution method is described for a general problem. It forms the framework, in which the sub-circuit recognition problem will be embedded in subsequent sections.

After stating the definition of a general search problem, the concept of a general search tree will first be described. Next, a description of the backtracking method to find the solution set of the problem is given. Finally, some remarks on the performance of the method are given.

Definition 4.26 *Search Problem*
A Search Problem is defined by

1. finite sets Y_j, $j = 1, \ldots, k$, that define a *search space* $\mathbf{Y} = Y_1 \times \ldots \times Y_k$,
2. a *search predicate* $\mathbf{D} : \mathbf{Y} \rightarrow \{True, False\}$.

The set of solutions of the problem is equal to the true-set $\mathcal{D}$ of $\mathbf{D}$. □

$\mathcal{D}$ can be computed by evaluating $\mathbf{D}$ for every tuple $\mathbf{y} \in \mathbf{Y}$. This method, the general brute-force method, is usually very expensive, since $|\mathbf{Y}|$ is large. The backtracking method may skip large parts of $\mathbf{Y}$ by using a general search tree, based on a permutation $\mathbf{p}$ of $(1, \ldots, k)$. Figure 4.4 shows an

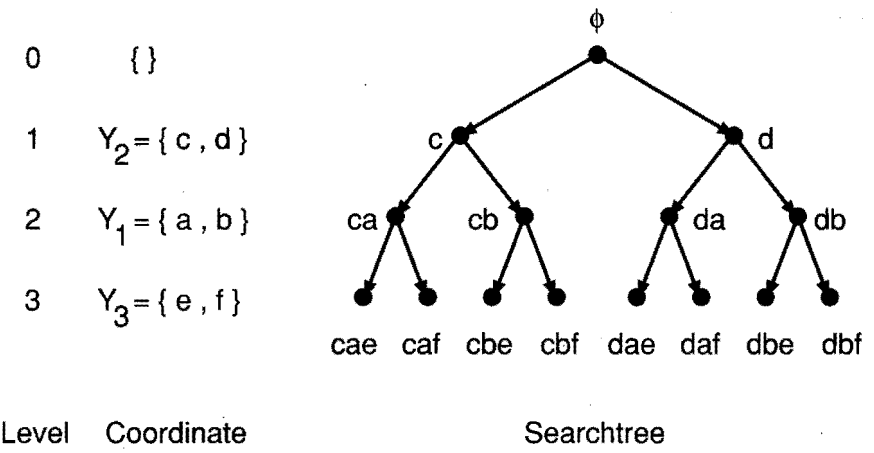


Figure 4.4: *A search tree, for* $\mathbf{Y} = \{a, b\} \times \{c, d\} \times \{e, f\}$, *and* $\mathbf{p} = (2, 1, 3)$. *The levels associated with the vertices are indicated on the left side, including the relevant coordinate of* $\mathbf{Y}$.

example of a search tree for which $\mathbf{Y} = \{a, b\} \times \{c, d\} \times \{e, f\}$, and $\mathbf{p} = (2, 1, 3)$. For j in $\{1, \ldots, k\}$, the number p_j assigns the coordinate Y_{p_j} to level j, e.g., the set $Y_{p_1} = Y_2 = \{c, d\}$ is associated to level 1. We see that starting from the root vertex $\emptyset$ at level 0, each vertex represents a sub-space of $\mathbf{Y}$. The sub-space is recursively partitioned, at every proceeding level. Each leaf represents a permuted tuple of $\mathbf{Y}$, e.g., the vertex *cae* corresponds to tuple *ace*.

Definition 4.27 *General Search Tree*
For a search problem (Definition 4.26), for any permutation $\mathbf{p} = (p_1, \ldots, p_k)$ of $(1, \ldots, k)$ called a *search order*, a General Search Tree is a directed non-cyclic graph $G_{\mathbf{p}} = (V_{\mathbf{p}}, E_{\mathbf{p}})$. The set of vertices $V_{\mathbf{p}}$ is given by

$$V_{\mathbf{p}} = \bigcup_{j=0}^{k} V_j. \tag{4.17}$$

V_j is called the set of vertices at level j. V_0 is defined by $\{\emptyset\}$, and $\emptyset$ is called the *root*. The vertices at level j, $j = 1, \ldots, k$, are given by

$$V_j = \prod_{i=1}^{j} Y_{p_i}. \tag{4.18}$$

At each vertex $v = (y_1, \ldots, y_j)$ at level j, $0 \leq j < k$, a set of edges points at the vertices of the set $\{(y_1, \ldots, y_j)\} \times Y_{p_{j+1}}$. Hence, the edge set is given by

$$E_{\mathbf{p}} = \bigcup_{j=0}^{k} \bigcup_{v \in V_j} \{\ (v, w) \mid w \in \{v\} \times Y_{p_{j+1}}\}. \tag{4.19}$$

For every vertex $v \in V_{\mathbf{p}}$, a unique path exists from root $\emptyset$ to vertex v. □

The backtracking method described next, uses a search tree traversal to examine $\mathbf{Y}$ selectively. To employ a backtracking method for a search problem, the function $\mathbf{D}$ is decomposed according to the following definition.

Definition 4.28 *Demand Function Decomposition*
For a search problem (see Definition 4.26), a search tree (see Definition 4.27) with search order $\mathbf{p}$, a Demand Function Decomposition of the function $\mathbf{D}$ is defined by an ordered set of functions $\mathbf{d} = (d_1, \ldots, d_k)$, $d_i : V_i \rightarrow \{True, False\}$, called *demand functions*, for which for every tuple $\mathbf{y} = (y_1, \ldots, y_k) \in \mathbf{Y}$

$$\mathbf{D}(\mathbf{y}) = d_1(y_{p_1}) \wedge d_2(y_{p_1}, y_{p_2}) \wedge \ldots \wedge d_k(y_{p_1}, \ldots, y_{p_k})$$

holds. □

For example, suppose $k = 3$ and $Y_i = \{0, 1\}$ for $i = 1, 2, 3$. Suppose 0,1 correspond to $False, True$. Suppose for any tuple $(y_1, y_2, y_3) \in \mathbf{Y}$, $\mathbf{D}(y_1, y_2, y_3) = (y_1 \vee y_2) \wedge y_3$. The brute force method would evaluate $\mathbf{D}$ for all tuples in $\mathbf{Y}$. However, from the definition it is clear that when $y_3 = 0$, $\mathbf{D}$ evaluates to $False$. To examine the third coordinate of $\mathbf{Y}$ first, we use the search tree resulting from $\mathbf{p} = (3, 2, 1)$. Figure 4.5 shows the corresponding search tree, including the chosen set of demand functions $\mathbf{d} = (d_1, d_2, d_3)$.

In general, for a vertex $v = (y_1, \ldots, y_j)$ at level j, $1 \leq j \leq k$, $d_j(v)$ indicates whether any of the leaves reachable from v may represent an element of the solution set $\mathcal{D}$. Hence, when $d_j(v) = False$, the sub-set

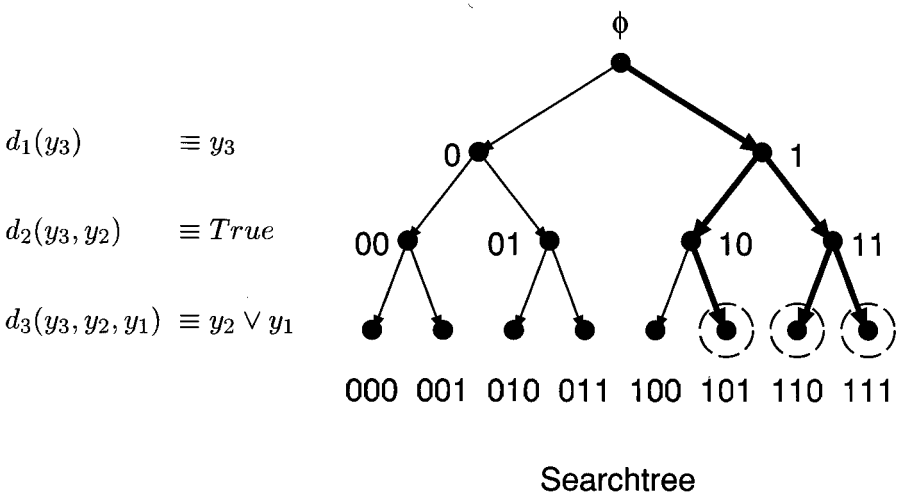


Figure 4.5: *A search tree, for* $\mathbf{Y} = \{0,1\}^3$, $\mathbf{p} = (3,2,1)$, *and* $\mathbf{D}(y_1, y_2, y_3) = (y_1 \vee y_2) \wedge y_3$. *On the left, a decomposition of the search predicate* $\mathbf{D}$ *into* $\mathbf{d} = \{d_1, d_2, d_3\}$ *is given. The bold edges indicate the traversed edges, the thin parts of the tree are skipped. The encircled leaves represent the three solutions of the set* $\mathcal{D}$.

$\{(y_1, \ldots, y_j)\} \times Y_{p_{j+1}} \ldots \times Y_{p_k}$ is not part of the solution set $\mathcal{D}$. This offers the possibility to skip the related parts of $\mathbf{Y}$ when computing $\mathcal{D}$. For Figure 4.5, at level 1, $d_1(0) = False$ implies that the total left part of the search tree can be ignored, since the leaves are not part of $\mathcal{D}$. The backtracking method exploits this property by traversing the search tree as follows. Starting from the root $\emptyset$, each path $\emptyset, \ldots, v$ in $G_{\mathbf{p}}$ is followed, until at some level j, $d_j(v) = False$. In this way, only a sub-tree of $G_{\mathbf{p}}$ is traversed, as indicated with the bold edges in Figure 4.5. The different coordinates $Y_1, \ldots, Y_k$ are entered in the order $\mathbf{p}$, explaining the name search order for $\mathbf{p}$. The leaves that are reached during the traversal constitute $\mathcal{D}$. For Figure 4.5, $\mathcal{D} = \{101, 011, 111\}$.

Definition 4.29 *Partial Search Predicate*
For a partial demand set $\mathbf{d} = (d_1, \ldots, d_j)$ of a search predicate $\mathbf{D}$ and search order $\mathbf{p}$, $1 \leq j \leq k$, the Partial Search Predicate is defined by

$\mathbf{D}_j : Y_{p_1} \times \ldots \times Y_{p_j} \rightarrow \{True, False\}$ for $\mathbf{y}_j \in Y_{p_1} \times \ldots \times Y_{p_j}$, by

$$\mathbf{D}_j(\mathbf{y}_j) = d_1(y_1) \wedge \ldots \wedge d_j(y_1, \ldots, y_j).$$

The true-set of $\mathbf{D}_j$ is denoted by $\mathcal{D}_j$. By using the inverse permutation function $\mathbf{p}^{-1} : Y_{p_1} \times \ldots \times Y_{p_k} \rightarrow Y_1 \times \ldots \times Y_k$ that maps a leaf vertex to its corresponding tuple in $\mathbf{Y}$, the relation between $\mathbf{D}_k$ and $\mathbf{D}$ is given by $\mathbf{D}(\mathbf{y}) = \mathbf{D}_k(\mathbf{p}^{-1}(\mathbf{y}))$ for $\mathbf{y} \in \mathbf{Y}$. This implies that

$$\mathcal{D} = \{\mathbf{p}^{-1}(\mathbf{y}) \mid \mathbf{y} \in \mathcal{D}_k\}, \tag{4.20}$$

see Definition 4.28. □

The set of vertices at level j that is encountered during search tree traversal by backtracking is equal to $\mathcal{D}_j$. Note the difference between $\mathbf{D}_k$ and $\mathbf{D}$.

Definition 4.30 *Traversal Size*
For a search order $\mathbf{p}$, a search tree $G_\mathbf{p}$ of k levels (Definition 4.27), partial search predicates $\mathbf{D}_i$, $i = 1, \ldots, k$ (Definition 4.29), the Traversal Size of $G_\mathbf{p}$, denoted by $|G_\mathbf{p}|$, is defined by

$$|G_\mathbf{p}| = \sum_{i=1,\ldots,k} |\mathcal{D}_i|.$$

A Partial Traversal Size of $G_\mathbf{p}$ up to level j, $j \leq k$, denoted by $|G_\mathbf{p}|_j$, is defined by

$$|G_\mathbf{p}|_j = \sum_{i=1,\ldots,j} |\mathcal{D}_i|.$$

□

The traversal size of a search tree is equal to the total number of search tree vertices that are visited during the application of a backtracking method. The backtracking method is defined as follows.

Definition 4.31 *Backtracking Method*
For a search space $\mathbf{Y}$ and search predicate $\mathbf{D}$ (Definition 4.26), a search order $\mathbf{p}$, the search tree $G_\mathbf{p}$ (Definition 4.27), a demand function decomposition $\mathbf{d} = (d_1, \ldots, d_k)$ and partial search predicates $\mathbf{D}_1, \ldots, \mathbf{D}_k$ (Definition 4.29), the Backtracking Method is defined by the following steps.

1. Find the first candidate set $\mathcal{Y}_1$ of D_1, i.e., the true-set $\mathcal{D}_1 \equiv \delta_1$. Initialize $\mathcal{D}_j = \emptyset$ for $j \in \{2, \ldots k\}$.

2. For every $j \in \{1, \ldots, k-1\}$ find for every $\mathbf{y}_j = \{y_1, \ldots, y_j\} \in \mathcal{D}_j$ the candidate set

$$\mathcal{Y}_{j+1} = \{y \in Y_{p_{j+1}} \mid d_{j+1}(\mathbf{y}_j, y)\}, \tag{4.21}$$

to construct

$$\mathcal{D}_{j+1} = \{\mathbf{y}_j \times \mathcal{Y}_{j+1}\} \cup \mathcal{D}_{j+1}. \tag{4.22}$$

3. Use Equation 4.20 of Definition 4.29 to compute the solution set $\mathcal{D}$ from $\mathcal{D}_k$.

□

During a sequential execution of this method, a candidate set may become empty. This means that the current path of the search tree need not be followed further. So one retreats from this branch, and continues with another path. This pattern explains the name of the method, backtracking. The proof by induction that every solution set $\mathcal{D}_j$, $j = 1, \ldots, k$, is generated, is straightforward, since Step 1 and Step 2 correspond directly to the basis and induction step of the proof. Since one backtracks when a candidate set $\mathcal{Y}_{j+1}$ is empty, the method requires $|G_{\mathbf{p}}|$ steps.

Some remarks on the method

For any search problem defined by a search space, a search predicate $\mathbf{D}$, and a search order $\mathbf{p}$, one can always define a set of demand functions. For example, the trivial decomposition $d_1 \equiv \ldots \equiv d_{k-1} \equiv \mathit{True\_function}$, and $d_k(v) \equiv \mathbf{D}(\mathbf{p}^{-1}(v))$ can always be defined, but obviously leads to a complete traversal of $G_{\mathbf{p}}$, and is not useful. Therefore, the "quality" of the decomposition determines the ability of skipping large parts of $\mathbf{Y}$. Also, the chosen search order usually strongly effects the size of the traversed part. Figure 4.6 shows a variant of Figure 4.5, with $\mathbf{p}' = (1, 2, 3)$. In Figure 4.6, the best decomposition still traverses a larger part than in Figure 4.5. This shows that one order allows better demand functions than others. When the most discriminating coordinates are first in the search order, the most selective demands are near to the top of the search tree, and the number and size of unsuccessful paths is reduced. Note that a permutation of coordinates is not simply equivalent to a permutation of demand functions. Also, the encircled leaves corresponding to the solution set $\mathcal{D}$ are different for Figure 4.5 and Figure 4.6.

We conclude that the efficiency of backtracking depends firstly on the search order, and secondly on a proper decomposition of $\mathbf{D}$ into a demand set $\{d_1, \ldots, d_k\}$.

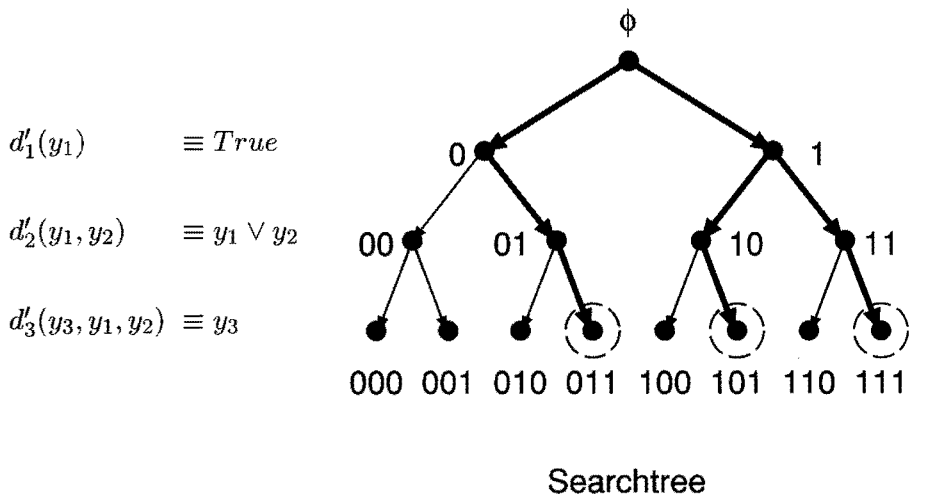


Figure 4.6: *Alternative backtracking search tree and traversed sub-tree for different search order* $\mathbf{p}'$. *The sub-tree is larger than in Figure 4.5, and other leaves (the encircled vertices) represent the same solution set* $\mathcal{D}$.

4.3.3 Backtracking and sub-circuit recognition

After having described the backtracking method in general, this section describes the first steps towards the transformation of the sub-circuit recognition problem as described in Section 4.2.3 into a backtracking problem.

According to Definition 4.31, Definition 4.26 and Definition 4.27 in the previous section, a backtracking process is characterized by

1. sets Y_i, $i = 1, \ldots, k$, that define the search space $\mathbf{Y}$,
2. a search predicate $\mathbf{D} : \mathbf{Y} \rightarrow \{True, False\}$, whose true-set $\mathcal{D}$ is the solution set,
3. a search order $\mathbf{p}$ defining a general search tree,
4. a decomposition of $\mathbf{D}$ into a demand set $\mathbf{d}_k = (d_1, \ldots, d_k)$.

By specifying these characteristics for the case of sub-circuit recognition, the sub-circuit recognition problem can be solved by backtracking.

In Section 4.2.3, the sub-circuit recognition problem definition (Definition 4.23) is based on the isomorphism predicate **S** (see Definition 4.20) defined by the isomorphism conditions (Equations 4.2,...,4.7). The isomorphism predicate **S** implicitly defines the solution set

$$\mathcal{S} = \{\phi \in \mathbf{V}^V \mid \mathbf{S}(\phi)\}.$$

The search space **Y** is defined as follows.

Definition 4.32 *Search space*
V is the template vertex set. **V** is the main circuit vertex set. The Search Space **Y** has $k = |V|$ dimensions. The coordinate sets are defined by $Y_i = \mathbf{V}$, $i = 1, \ldots, k$. Hence $\mathbf{Y} = \mathbf{V}^{|V|}$. □

Since the coordinate sets are equal, the search tree (see Definition 4.27) is equal for any search order **p**. Therefore, we define **p** to be the identity permutation $(1, \ldots, k)$. This implies that $\mathcal{D} = \mathcal{D}_k$, so Step 3 of the backtracking method (Definition 4.31) is trivial. For an ordered set $\mathbf{s} = (v_1, \ldots, v_k)$, called the search list, enumerating the elements of V, every tuple $\mathbf{w} = (w_1, \ldots, w_k) \in \mathbf{Y}$ corresponds to a pair function $\phi : V \to \mathbf{V}$, denoted by $\mathbf{s} \bullet \mathbf{w}$, according to Definition 4.6 defined by $\phi(s_i) = w_i$ for $i = 1, \ldots, k$. When $\mathbf{S}(\phi) = True$, it is part of the solution set $\mathcal{S}$. For different search lists, the search tree remains the same, but the function ϕ that corresponds with a tuple $\mathbf{w} \in \mathbf{Y}$ is different. This means that the selection of a search order **p**, is now transformed into a selection of an ordered set **s**, the search list.

Definition 4.33 *D*
For an isomorphism predicate **S** (see Definition 4.20), the search space **Y** (see Definition 4.32), for an ordered set of template vertices $(v_1, \ldots, v_k)$ called the search list **s**, the search predicate $\mathbf{D} : \mathbf{Y} \to \{true, false\}$ is defined for $\mathbf{w} \in \mathbf{Y}$ by

$$\mathbf{D}(\mathbf{w}) = \mathbf{S}(\mathbf{s} \bullet \mathbf{w}).$$

For $j \leq k$, a prefix $(v_1, \ldots, v_j)$ is called a *partial search list*, denoted by $\mathbf{s}_j$, and $\mathbf{s}_k$=**s**. □

The following questions have remained:

1. how to order the template vertices V into a search list **s**,
2. how to decompose **D** into an ordered set of demands $\mathbf{d} = (d_1, \ldots, d_k)$,

3. how to traverse the search tree efficiently, for a given set of demands $\mathbf{d}$,

4. how to find the candidate sets $\mathcal{Y}_{j+1}$ efficiently during the search tree traversal.

Two aspects are important for efficiency: the cost of the ordering and decomposition method, and the cost of the backtracking process. The cost for ordering and decomposition must be less than the gain in backtracking to be justified.
These four questions are addressed in the next sections. To understand the consequences of the search list order, one must know the decomposition and remainder of the backtracking process. Therefore, the search list ordering will be addressed after the other three, in Section 4.3.7.

4.3.4 The decomposition of the sub-circuit recognition problem

After reviewing briefly the starting point of sub-circuit recognition by backtracking as given in the previous section, this section describes the decomposition of the problem into a set of demands $\mathbf{d} = (d_1, \ldots, d_k)$, for a given search list $\mathbf{s}$. In addition to the decomposition, it will be shown that the demand set $\mathbf{d}$ actually defined is equivalent to the isomorphism predicate $\mathbf{S}$ (Definition 4.20, Section 4.2.3), that defines the search predicate $\mathbf{D}$ (Definition 4.33).

The starting point
A given main circuit $\mathbf{G} = (\mathbf{V}, \mathbf{T}, \mathbf{A}, \mathbf{E}, \mathbf{TC})$ is defined as in Definition 4.18, Section 4.2.3, including components set $\mathbf{C}$, nets set $\mathbf{N}$, terminal classes function $\mathbf{TCS}$ and degree function $\mathbf{DEGREE}$. A given template circuit $G = (V, T, A, E, TC)$ with external nets NE is defined as in Definition 4.19, Section 4.2.3, including components set C, the nets set N, internal nets set NI, terminal classes function TCS and degree function $DEGREE$. Note that the template circuit G is connected and non-trivial. Figure 4.7 shows the schematics and the graph of the template circuit example introduced in Section 4.2.3, Figure 4.3. The number of template vertices is k. Since we want to exploit connectivity to reduce the traversed part of a search tree, we will demand that the search list is prefix connected, defined as follows.

Definition 4.34 *Prefix Connected*
For a circuit $G = (V, T, A, E, TC)$ defined according to Definition 4.17, $k = |V|$, an ordered set of vertices $(v_1, \ldots, v_k)$ is Prefix Connected when for $i = 2, \ldots, k$ v_i is connected to some predecessor v_j, $(j < i)$. □

A given search list $\mathbf{s}$ is an ordered prefix connected set $(v_1, \ldots, v_k)$, enumerating all template vertices. For search list $\mathbf{s}$, the partial search lists $\mathbf{s}_i = (v_1, \ldots, v_i)$ for $i \in \{1, \ldots, k\}$. For the main circuit $\mathbf{G}$, the template circuit G and the search list $\mathbf{s}$, the isomorphism predicate $\mathbf{S}$ is defined by Definition 4.20, Section 4.2.3, based on the equations $4.2, \ldots, 4.7$. The search list $\mathbf{s}$ and isomorphism predicate $\mathbf{S}$ together define the search predicate $\mathbf{D}$ according to Definition 4.33. In the previous section, the role of the search order $\mathbf{p}$ has been taken over by the search list $\mathbf{s}$. Therefore, the general search tree $G_{\mathbf{p}}$ (Definition 4.27) translates into the following search tree definition.

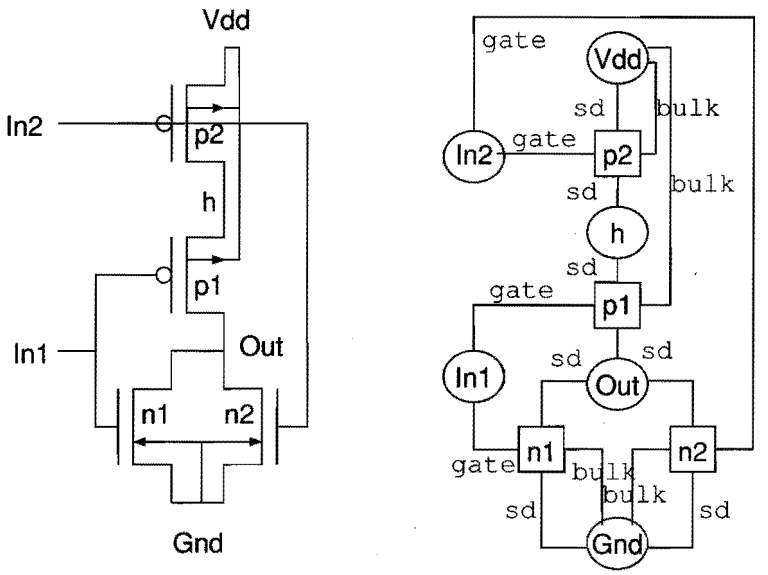


Figure 4.7: *The schematics and circuit of a NOR template, introduced in Figure 4.3 of Section 4.2.3. The internal net set is given by* $NI = \{h\}$, *and the external net set is given by* $NE = \{Vdd, In2, Out, Gnd, In1\}$.

Definition 4.35 *Search Tree*
For the main circuit vertex set $\mathbf{V}$, the template circuit vertex set V and a search list $\mathbf{s}$, the Search Tree is a directed non-cyclic graph $G_{\mathbf{s}} = (V_{\mathbf{s}}, E_{\mathbf{s}})$ according to Definition 4.27. The set of vertices $V_{\mathbf{s}}$ is given by

$$V_{\mathbf{s}} = \bigcup_{j=0}^{k} V_j. \tag{4.23}$$

V_j is called the set of vertices at level j. V_0 is defined by $\{\emptyset\}$, and $\emptyset$ is the root. The vertices at level j, $j = 1, \ldots, k$, are given by $V_j = \mathbf{V}^j$. The edge set $E_{\mathbf{s}}$ is given by

$$E_{\mathbf{s}} = \bigcup_{j=0}^{k} \bigcup_{v \in V_j} \{(v, w) \mid w \in \{v\} \times \mathbf{V}\}. \tag{4.24}$$

Each vertex $\mathbf{w} \in V_{\mathbf{s}}$ at level j is associated with a function $\phi_j = \mathbf{s}_j \bullet \mathbf{w}$. □

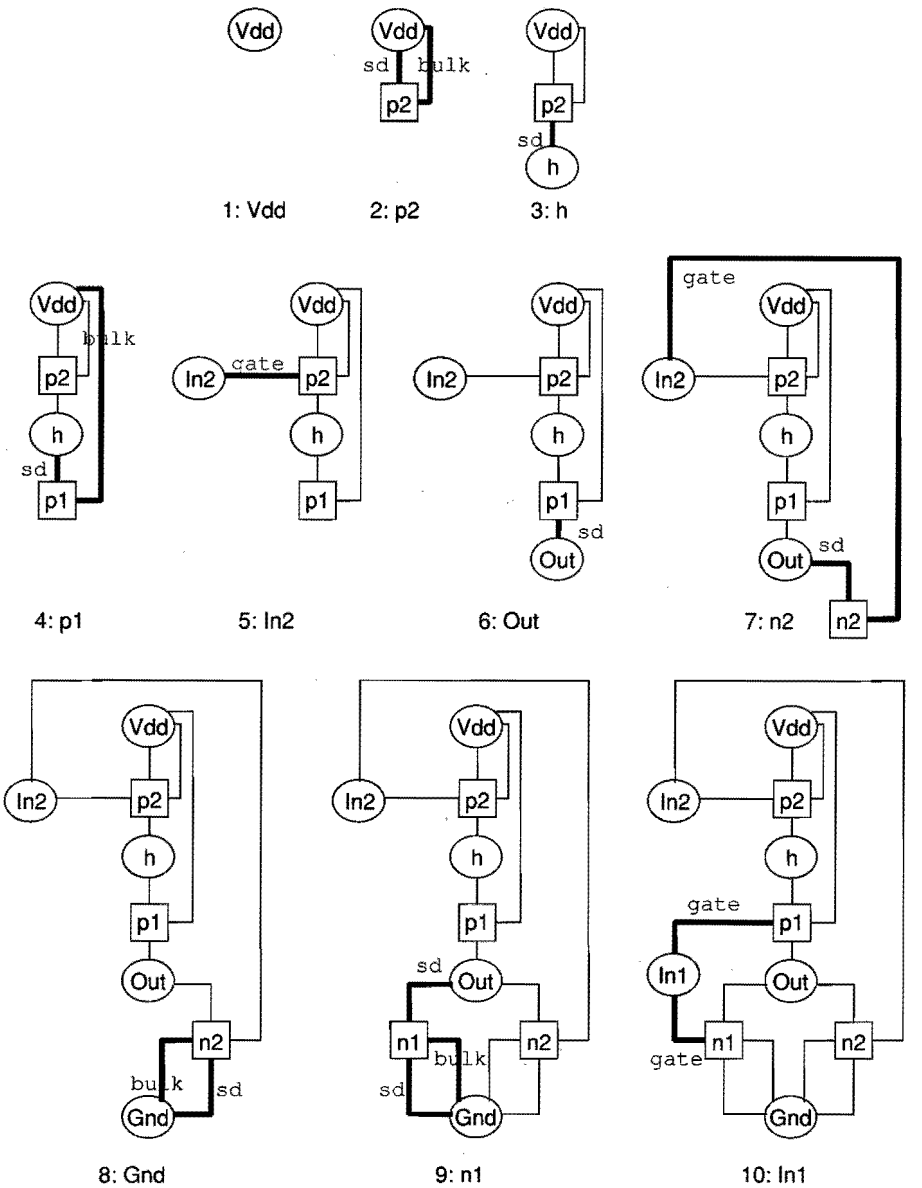


Figure 4.8: *The sub-circuits* $G|_{\mathbf{s}_j}$, $j = 1, \ldots, 10$ *of the NOR template circuit (Figure 4.7), when the search list is given by* $\mathbf{s} = \mathbf{s}_{10} =$ (Vdd, p2, h, p1, in2, out, n2, gnd, n1, in1). *Apart from the internal net h, each net is an external net. The new edges in* $G|_{\mathbf{s}_j}$ *with respect to* $G|_{\mathbf{s}_{j-1}}$, *are drawn in bold lines. At the bottom of each graph,* j *and* v_j *are given.*

Decomposition
The decomposition of $\mathbf{D}$ into set $\mathbf{d} = (d_1, \ldots, d_k)$ is described, starting from the above definitions. Based on partial search lists $\mathbf{s}_j$, $j = 1, \ldots, k$, the decomposition will correspond to the sub-circuits $G|_{\mathbf{s}_j}$ of the template circuit. Figure 4.8 shows a search list and the sub-circuits for the template circuit of Figure 4.7. It shows that each proceeding vertex v_j introduces new edges. Based on these new edges, we will associate with each search tree level j a demand function $d_j : V_j \rightarrow \{True, False\}$, that checks for a candidate $w_j \in \mathbf{V}$ the following properties.

- With regard to the labeled new edges between v_j and the elements of $\mathbf{s}_{j-1} = (v_1, \ldots v_{j-1})$, corresponding edges must be present between w_j and $w_1, \ldots, w_{j-1}$. This part of d_j is called the connectivity demand. Figure 4.8 shows, per level j, the relevant edges of the template in bold lines. For example, for $j = 3$, a candidate net w_3 that matches $v_3 = h$ should be connected to match w_2 of $v_2 = p2$, with an *sd* labeled edge.

- The attributes $\mathbf{A}(w_j)$ must be equivalent to the attributes $A(v_j)$, and corresponding degree per class must be similar. Also for a component, the types must be equal. For example, for a match w_2 of $v_2 = p2$, d_2 checks whether $type = MOS$, $width = 1e-6$, and $model = PTYPE$. For $v_3 = h$, d_3 checks whether w_3 has exactly two *sd* connections, since $h \in NI$. This part of d_j is called the *local demand*, since it is independent of the search list order.

Therefore each function d_j will be composed of a connectivity demand function F_j and a local demand function L_{v_j}. Also the relation to the isomorphism predicate $\mathbf{S}$ is explained.

Definition 4.36 *Local Demand Function*
For each $v \in V$, a Local Demand Function $L_v : \mathbf{V} \rightarrow \{True, False\}$ has the following definition for $w \in \mathbf{V}$:

$$L_v(w) = \begin{cases} (A(v) = \mathbf{A}(w)) \wedge (T(v) = \mathbf{T}(w)) & \text{if } v \in C \\ \forall c \in \tau : DEGREE(v, c) = \mathbf{DEGREE}(w, c) & \text{if } v \in NI \\ \forall c \in \tau : DEGREE(v, c) \leq \mathbf{DEGREE}(w, c) & \text{if } v \in NE \end{cases}$$

□

For a template component v, the local demand function L_v checks equality of attribute-name, attribute-value pairs and types of v with a component

candidate w. For a net $v \in N$ and $w \in \mathbf{N}$, $A(\mathrm{v}) = \mathbf{A}(w) = T(\mathrm{v}) = \mathbf{T}(w) = \emptyset$ by definition. With respect to isomorphism predicate **S**, this can be used to check Conditions 4.3 and 4.4 of Section 4.2.3. For a net candidate w, the local demand function checks the number of connections per class, which is different for internal nets versus external nets. Since the number of connections per component type is fixed, a component $v \in C$ and $w \in \mathbf{C}$ having $T(\mathrm{v}) = \mathbf{T}(w)$ imply that $DEGREE(v, c) = \mathbf{DEGREE}(w, c)$ for any class c. Therefore, L_v can be used to check Conditions 4.6 and 4.7 (Section 4.2.3). When $v \in C$ and $w \in \mathbf{N}$, or $v \in N$ and $w \in \mathbf{C}$, $L_v(w) = False$.
In summary, each L_v can be used to check Conditions 4.6, 4.7, 4.3 and 4.4 of **S** for the given main circuit vertex. As will be shown next, the remaining two Conditions 4.2 and 4.5 of **S** can be checked by the connectivity demand functions.

Definition 4.37 *Connectivity Demand Function*
For each $j = 1, \ldots, k$, a Connectivity Demand Function $F_j : \mathbf{V}^j \rightarrow \{True, False\}$ is defined for a search tree vertex $\mathbf{w}_j = (w_1, \ldots, w_j)$ (see Definition 4.35) by:

$$F_j(\mathbf{w}_{j-1}, w_j) = \begin{cases} True & \text{if } j = 1 \\ \forall i \in \{1, \ldots, j-1\} : & \text{if } 1 < j \leq k \\ (TCS(v_j, v_i) = \mathbf{TCS}(w_j, w_i)) \ \wedge & \\ \qquad (w_j \notin \mathbf{w}_{j-1}). & \end{cases}$$

F_j depends on the ordering of **s**. □

F_j checks whether the new edges between v_j and $\mathbf{s}_{j-1}$, as indicated in Figure 4.8 have corresponding edges between w_j and $\mathbf{w}_{j-1}$. Figure 4.9 illustrates the definition of $F_j(\mathbf{w}_{j-1}, w_j)$. Since components and nets are treated equally, they are both drawn by bullets in the figure. Based on L_j and F_j, the demand functions can now be defined.

Definition 4.38 *Demand Function*
For $j = 1, \ldots, k$ a Demand Function $d_j : \mathbf{V}^j \rightarrow \{True, False\}$ is defined for a search tree vertex $\mathbf{w}_j = (w_1, \ldots, w_j)$ at level j by

$$d_j(\mathbf{w}_{j-1}, w_j) \equiv F_j(\mathbf{w}_{j-1}, w_j) \wedge L_{v_j}(w_j).$$

□

When the backtracking method as defined in Definition 4.31 is applied with the current decomposition $\mathbf{d} = (d_1, \ldots, d_k)$, the relation between $G_{\mathbf{s}}$, d_j,

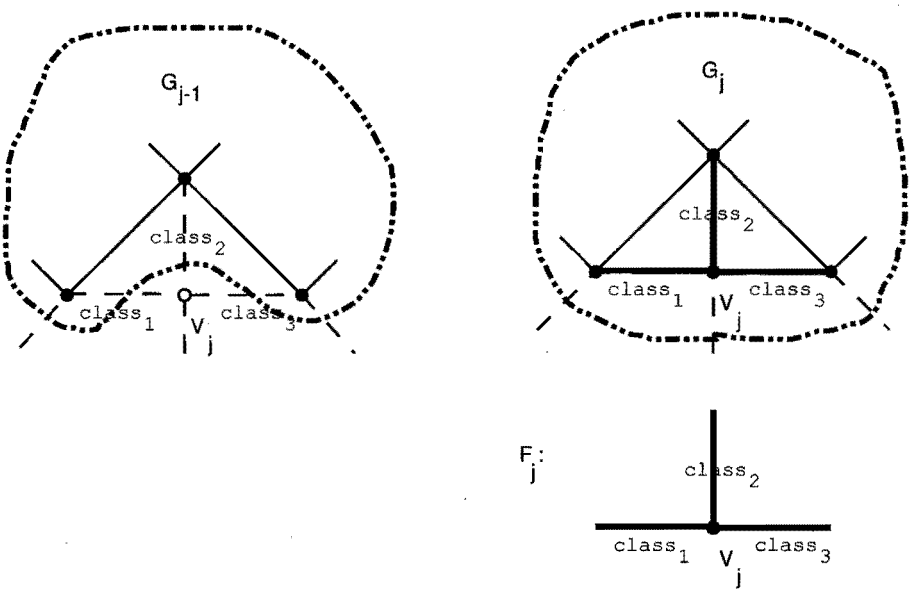


Figure 4.9: *Relation between the outlined restricted template graphs G_{j-1}, G_j and connectivity demand function F_j. Both components and nets are represented by small bullets, because they are treated equally.*

F_j and $\mathbf{S}$ is as follows.

Let $j \in \{1, \ldots, k\}$ be a level in the search tree $G_{\mathbf{S}}$.
For $j = 1$, the sub-circuit $G|_{\mathbf{s}_1}$ of the template, consists of a single vertex v_1. No new connectivity is associated with the traversal from the root vertex $\emptyset$ of the search tree to level 1 vertices, so $F_1 = True\_function$, and matches of $G|_{\mathbf{s}_1}$ should only satisfy L_{v_1}.
For $j > 1$, a unique path from root $\emptyset$ to the vertex $\mathbf{w}_{j-1} = (w_1, \ldots, w_{j-1})$ is traversed according to Definition 4.31 of the sub-tree of $G_{\mathbf{S}}$, for which all $d_i(\mathbf{w}_i) = True$ $(i < j)$. In other words, $F_i(\mathbf{w}_i) = L_{v_i}(w_i) = True$ for $i = 1, \ldots, j-1$, and every $w_i \in \mathbf{V}$ is a corresponding candidate of v_i. Hence $\mathbf{w}_{j-1} \in \mathcal{D}_{j-1}$. Let G_{j-1} denote the sub-circuit $G|_{\mathbf{s}_{j-1}}$ of the template. Let the function ϕ_{j-1}, associated with G_{j-1}, be equal to the pair function $\mathbf{s}_{j-1} \bullet \mathbf{w}_{j-1}$. Since $w_i \notin \mathbf{w}_{i-1}$ according to Definition 4.37, ϕ_{j-1} is one-to-one. $F_i(\mathbf{w}_i) = True$ implies that the connectivity Condition 4.5 and Condition 4.2 of $\mathbf{S}$ are met by ϕ_{j-1}. Since also all $L_{v_i}(w_i) = True$, ϕ_{j-1} is an isomorphism of the template sub-circuit G_{j-1}. In Figure 4.9, the dotted line encircles on the left the connections between every $v \in \mathbf{s}_{j-1}$. On the right

it also includes connections to v_j. The difference is represented in F_j. F_j checks for a candidate $w_j \in \mathbf{V}$ whether the corresponding connectivity per terminal-class between v_j and its predecessors is present. In other words, F_j checks the Conditions 4.2 and 4.5 of $\mathbf{S}$ related to the edges between v_j and $\mathbf{s}_{j-1}$.

We see that when for a $w_j \in \mathbf{V}^j$, $L_{v_j} = F_j = \text{True}$, all conditions of $\mathbf{S}$ are met with respect to G_j. This implies that $\phi_j = \mathbf{s_j \bullet w}_j$ is an isomorphism of G_j. By using recursion, it follows that for $j = k$, $\phi_k = \mathbf{s}_k \bullet \mathbf{w}_k = \mathbf{s} \bullet \mathbf{w}$ is an isomorphism of the complete template G. This concludes the decomposition of $\mathbf{D}$ into the given set of demands $\mathbf{d} = (d_1, \ldots, d_k)$, composing of a local demand functions L_v, for every $v \in V$, and search list order dependent set of connectivity functions F_j, $j = 1, \ldots, k$.

4.3.5 Search tree traversal

This section focusses on search tree traversal and efficiency. Two known tree traversal methods that implement the backtracking method (Definition 4.31), called depth-first and breadth-first, are briefly introduced, to provide a motivation for why depth-first has been selected as the most efficient method. Next, the depth-first search algorithm will be described.

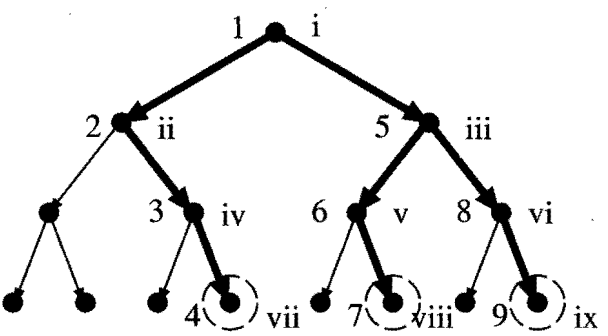


Figure 4.10: *Depth-first and breadth-first tree traversal. Depth-first and breadth-first order are respectively indicated by arabic numbers on the left side and roman numbers on the right side of each point.*

Breadth-first and depth-first traversal

Breadth-first and depth-first tree traversal are both shown in Figure 4.10.

When performing breadth-first search, the tree is traversed with "horizontal" preference, and for depth-first search, the tree is traversed with "vertical" preference. Starting from the root (the top), the preferred direction is advanced as far as possible, then the non-preferred direction is advanced one step and next the preferred direction is tried again, see also Figure 4.10. The traversal algorithm efficiency depends on the chosen data structure. When the graph is represented as adjacency lists (see Section 4.2.4), the traversal algorithm is linear in the size of the search tree for both methods[Sedge88]. However, depth-search and breadth-search differ in their storage use. Eventually after the traversal, every solution (match) of $\mathbf{D}_k$ given by a search tree vertex $\mathbf{w}_k = (w_1, \ldots, w_k)$ must have been stored. Each vertex $\mathbf{w}_k$ represents the unique path $\emptyset, \mathbf{w}_1, \ldots, \mathbf{w}_k$ connecting the root with the vertex.

Breadth-first traversal, traverses level by level. At level j, $0 \leq j < k$, all reached vertices must be remembered to know which paths should be continued at the next level $j+1$. Therefore, for each level j, the partial solution set $\mathcal{D}_j$ of $\mathbf{D}_j$ must be stored, including the vertices that will eventually not lead to a solution at level k.

For depth-first traversal, one path is considered at a time and advanced until it leads either to a match or it cannot be completed. Only one vertex, one partial solution, need be stored, together with the previously found matches.

Therefore, depth-first traversal is to be favored over breadth-first because it uses storage more efficiently, which also results in better run times. Although the argument suggests that depth-first is always better, sometimes breadth-first traversal is preferred for other backtracking problems. For instance, for the shortest path problem, the breadth-first method is preferred.

Depth-first search algorithm

In Figure 4.11 the pseudo-code for the recursive depth-first search algorithm is shown. STORE_PATH saves a full match satisfying $\mathbf{d}_k = (d_1, \ldots, d_k)$ into the global data structure. The function FIND_CANDIDATE_SET returns a set of candidates for which each element w, $d_{j+1}(\mathbf{w}_j, w)$ is $True$, i.e., it computes $\mathcal{Y}_{j+1}$ (Equation 4.21). It is fully explained in the next section. MARK and UNMARK allow a quick check at the proceeding recursion levels to prevent candidates from occurring more than once in a partial match. The recursion depth is obviously limited to $k = |V|$, and is

```
Procedure DEPTH_FIRST_SEARCH (j)

    if j = k
        then STORE_PATH(w_k)
        else for all cand ∈ FIND_CANDIDATE_SET(d_{j+1}, w_j)
                 w_{j+1}:=cand
                 MARK(cand)
                 DEPTH_FIRST_SEARCH(j+1)
                 UNMARK(cand)
             endfor
    endif

endproc
```

Figure 4.11: *The depth-first search algorithm.*

independent from the main circuit **G**.

4.3.6 Finding a candidate set for a demand

The previous section described the depth-first search algorithm for traversing the search tree, based on the FIND_CANDIDATE_SET algorithm, that implements the computation of the candidate set $\mathcal{Y}_{j+1}$ (see Equation 4.21). This section describes how FIND_CANDIDATE_SET can be computed efficiently. In this section the computational complexity will also be given, followed by an enhancement for a special case that often occurs.

FIND_CANDIDATE_SET

In the depth-first search algorithm of Figure 4.11, FIND_CANDIDATE_SET(d_{j+1},$\mathbf{w}_j$) must return all vertices of the main circuit for which the demand d_{j+1} holds, after arriving at vertex $\mathbf{w}_j \in \mathcal{D}_j$, for which the corresponding search tree path $\emptyset, \ldots, \mathbf{w}_j$ has been traversed. For the sub-circuit recognition problem $Y_{p_{j+1}} = \mathbf{V}$, $1 \leq j < k$, (see Definition 4.32), so the actual candidate set $\mathcal{Y}_{j+1}$ is given by

$$\mathcal{Y}_{j+1} = \{w \in \mathbf{V} \mid d_{j+1}(\mathbf{w}_j, w)\}. \tag{4.25}$$

A brute-force direct implementation of the above definition evaluates $d_{j+1}(\mathbf{w}_j, w)$ for every $w \in \mathbf{V}$, hence leading to $|\mathbf{V}|$ evaluations of d_{j+1} at every traversed vertex of the search tree. For reasonably sized circuits, this implementation becomes impracticable, so FIND_CANDIDATE_SET will be defined in a different way.

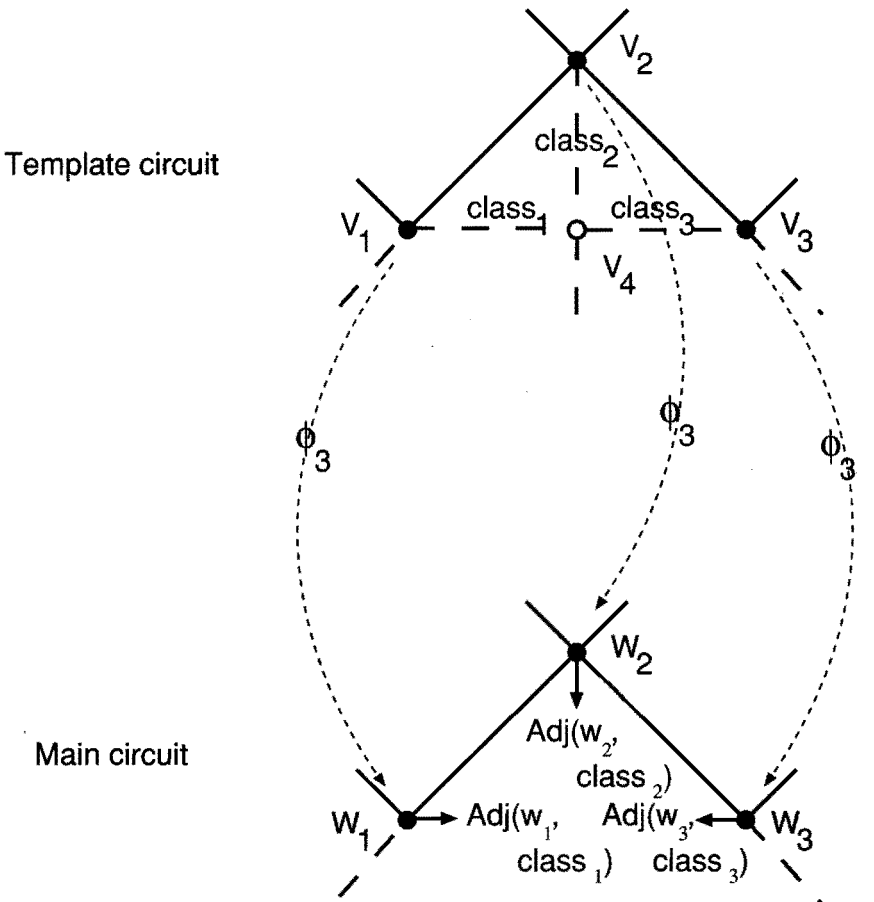


Figure 4.12: *Example of FIND_CANDIDATE_SET. Both components and nets are drawn by bullets. The multi-sets* $Adj(w_i, class_i)$, $i \in \{1,2,3\}$, *enumerate the vertices that are connected via edges to* w_i, *labeled* $class_i$. *Candidates matching* v_4 *must be part of the intersection of the multi-sets* $Adj(w_i, class_i)$ $(i = 1,2,3)$, *to have the required connections to all three vertices.*

An example

To introduce a better approach to compute $\mathcal{Y}_{j+1}$, consider the following typical example (see Figure 4.12). In the example, there are no multi-edges. Suppose that for $j = 3$, the depth-first search algorithm has arrived at the partial match $\mathbf{w}_3 = (w_1, w_2, w_3)$ for $\mathbf{s}_3 = (v_1, v_2, v_3)$. Obviously, for $i = 1, 2, 3$, $w_i = \phi_3(v_i)$. Given this context, we want to compute the set of candidates $\mathcal{Y}_4$ for v_4, satisfying d_4 for each candidate. First, we will consider connectivity. Let $\mathcal{F}_4$, called the tentative candidates, be the set of candidates that match the connectivity demand function F_4. Since v_4 is connected to v_1, v_2, v_3, with edges labeled class_1, class_2, class_3, each element $w \in \mathcal{F}_4$ should be connected to w_1, w_2, w_3, with corresponding labeled edges. According to the definition of the adjacency function of Section 4.2.4, Equation 4.25, each $Adj(w_i, class_i)$ $(i = 1, 2, 3)$ enumerates the multi-set of neighbor vertices of w_i, with label class_i. Therefore, for $i = 1, 2, 3$, $w \in Adj(w_i, class_i)$. Hence $\mathcal{F}_4 \subset (Adj(w_1, class_1) \cap Adj(w_2, class_2) \cap Adj(w_3, class_3))$. Each element w of the right-hand side that is not one of w_1, w_2, w_3, satisfies the connectivity demand, i.e., $F_4(\mathbf{w}_3, w) = True$. When also $L_{v_4}(w) = True$, $d_4(\mathbf{w}_3, w) = True$. Therefore,

$$\mathcal{F}_4 = \bigcap_{i=1,2,3} Adj(\phi(v_i), class_i) \setminus \mathbf{w}_3 \tag{4.26}$$

and

$$\mathcal{Y}_4 = \{w \in \mathcal{F}_4 \mid L_{v_4}(w)\} \tag{4.27}$$

and

$$\text{FIND\_CANDIDATE\_SET}(d_4, \mathbf{w}_3) = \mathcal{Y}_4. \tag{4.28}$$

This computation utilizes local information of the template graph, the main graph and the depth-search procedure, and the size of $\mathbf{V}$ is of no concern.

General case

With the previous example in mind, the FIND_CANDIDATE_SET algorithm will be described for the general case. For the general case, the depth-first search algorithm has arrived at a partial match $\mathbf{w}_j = (w_1, \ldots, w_j)$ of $\mathbf{s}_j$, given by $w_i = \phi_j(v_i)$ $(i = 1, \ldots, j)$. Now it is known which elements of $\mathbf{w}_j$ have to be connected to any unknown vertex w_{j+1} that matches v_{j+1}. For Figure 4.12, the vertices $\{v_1, v_2, v_3\}$ are connected to v_4, and the corresponding matching vertices $\{w_1, w_2, w_3\}$ have to be connected to any candidate w_4. In the general case, multi-edges are allowed, so the multiplicity must be taken into account as well.

Definition 4.39 *Template Neighbor Function*
For $j \in \{1, \ldots, k-1\}$, for each $i \in \{1, \ldots, j\}$, a Template Neighbor Function $b_i : V \rightarrow 2^{\mathbf{s}_i \times \Gamma \times \mathbb{N}^+}$ is defined by

$$b_i(v) = \{(u, c, m) \mid u \in \mathbf{s}_i, c \in \gamma(v) : m = \mu_u(Adj(v, class)) \wedge m > 0\}.$$

□

Each template neighbor function b_i enumerates for a template vertex argument v a set of (partial search list vertex, label, multiplicity) triples that represents the labeled multi-edges between v and $\mathbf{s}_i$. For example for Figure 4.12, $b_3(v_4) = \{(v_1, \text{class}_1, 1),\ (v_2, \text{class}_2, 1),\ (v_3, \text{class}_3, 1)\ \}$.

Definition 4.40 *Main Circuit Neighbor Set*
For a level $j \in \{1, \ldots, k-1\}$, a search tree vertex $\mathbf{w}_j = (w_1, \ldots, w_j) \in \mathcal{D}_j$ (see Definition 4.31), a function $\phi_j = s_j \bullet w_j$, the Main Circuit Neighbor Set is defined by

$$P_j = \{(\phi_j(u), c, m) \mid (u, c, m) \in b_j(v_{j+1})\}.$$

The template neighbor function b_j is defined by Definition 4.39. □

P_j is the corresponding main circuit neighbor set of $b_j(v_{j+1})$ describing the required edges of a candidate. P_j is a set of (main-circuit vertex, label, multiplicity) triples. For each triple $(w, c, m) \in P_j$, any candidate w_{j+1} should be adjacent to w via exactly m edges labeled with class c. For example in Figure 4.12, $P_4 = \{\ (w_1, \text{class}_1, 1),\ (w_2, \text{class}_2, 1),\ (w_3, \text{class}_3, 1)\ \}$. This is defined as follows.

Definition 4.41 *Tentative Candidate Set*
For a level $j \in \{1, \ldots, k-1\}$, a search tree vertex $\mathbf{w}_j \in \mathcal{D}_j$ and the main circuit neighbor set P_j (Definition 4.40), the Tentative Candidate Set $\mathcal{F}_{j+1}$, a subset of $\mathbf{V}$, is defined by

$$\mathcal{F}_{j+1} = \{\, w \in \bigcap_{(u,c,m) \in P_j} Adj(u, c) \backslash \mathbf{w}_j \mid \forall (u, c, m) \in P_j : m = \mu_w(Adj(u, c))\ \}\,.$$

□

The $m = \mu_w(Adj(u, c)$ part of Definition 4.41 checks the multiplicity of a candidate. $\mathcal{F}_{j+1}$ is the true-set of F_{j+1}, restricted to $\{\mathbf{w}_j\} \times \mathbf{V}$. For Figure 4.12, the tentative candidates for $\mathcal{F}_4$ are given by

$$\bigcap_{i=1,2,3} Adj(\phi(v_i), class_i, 1) \setminus \mathbf{w}_3,$$

since it contains no multi-edges. Only $L_{v_{j+1}}$ is left to be evaluated for each vertex in $\mathcal{F}_{j+1}$ to become a member of $\mathcal{D}_{j+1}$, so we arrive at a new equivalent definition of FIND_CANDIDATE_SET.

Definition 4.42 *FIND_CANDIDATE_SET*
For a level $j \in \{1, \ldots, k-1\}$, a search tree vertex $\mathbf{w}_j \in \mathcal{D}_j$, the tentative candidate set $\mathcal{F}_{j+1}$ (Definition 4.41), the algorithm for FIND_CANDIDATE_SET is defined by

$$\text{FIND\_CANDIDATE\_SET}(d_{j+1}, \mathbf{w}_j) = \{w \in \mathcal{F}_{j+1} \mid L_{v_{j+1}}(w)\}.$$

□

The computational complexity
The computation of FIND_CANDIDATE_SET(d_{j+1}) as described above takes $|\mathcal{F}_{j+1}|$ evaluations of $L_{v_{j+1}}$. The cost of computing the triples P_j for $\mathcal{F}_{j+1}$ depends only on the local graph structure of the neighbors of w_{j+1}, and the efficiency of the intersection operation, because the computation of the template neighbor functions $b_j(v_{j+1})$ only has to be done once for each level j, and computing $\phi_j(v)$ for $v \in \mathbf{s}_j$ has complexity $O(1)$. Since the candidates $\mathbf{w}_j$ are marked by the depth-first search procedure (Figure 4.11), they are efficiently omitted.

The relevant $Adj(w, class)$ are directly accessible in the circuit data structure (Section 4.2.4, Equation 4.25). The order of the FIND_CANDIDATE_SET algorithm is therefore given by

$$O(\sum_{(u,c,m)\in P_j} |Adj(p)|) = O(\sum_{(u,c,m)\in P_j} \mathbf{DEGREE}(p)), \tag{4.29}$$

i.e., the number of edges connected to the neighbors of v_i's matches. This result is much better than the direct FIND_CANDIDATE_SET implementation mentioned in the beginning of this section. For Figure 4.12, the algorithm is only $O(|Adj(v_1, class_1)| + |Adj(v_2, class_2)| + |Adj(v_3, class_3)|)$, and $|\mathbf{V}|$ is of no concern.

An enhancement
A simple but effective improvement is described next. It prevents the expensive computation of intersections of large sets in the formula of $\mathcal{F}_{j+1}$ where possible.
Suppose that for a certain FIND_CANDIDATE_SET computation, a set of

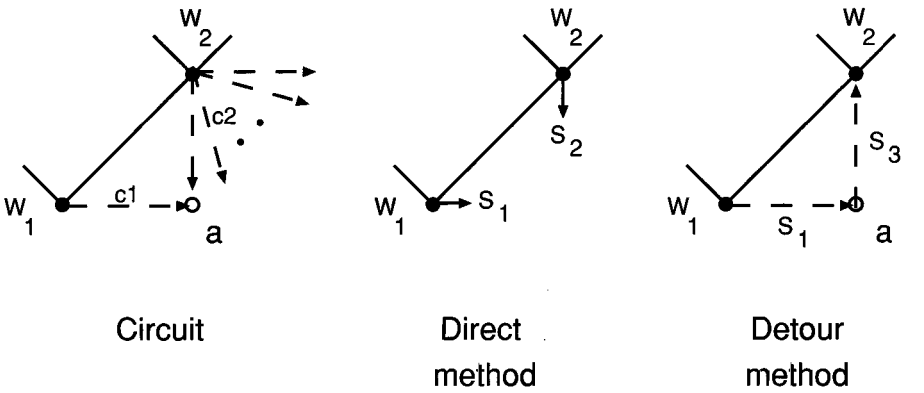


Figure 4.13: *Example of an improvement of FIND_CANDIDATE_SET, with* $Adj(w_1, c1) = S_1 = \{a\}$ *and* $Adj(w_1, c1) = S_1$, $|S_1| = 100$. *It shows that by using a detour, the fact that* $\{a\} = S_1 \cap S_2$ *can more efficiently be computed by checking whether* $w_2 \in Adj(a, c2)$, *when* $|Adj(w_2, c2)|$ *is large.*

candidates $\mathcal{F}_{j+1}$ is determined by two adjacency lists $S_1 = Adj(w_1, c1)$ and $S_2 = Adj(w_2, c2)$, as shown in Figure 4.13. Suppose that $S_1 = \{a\}$ and $|S_2| = 100$. The described method for FIND_CANDIDATE_SET would take the intersection of S_1 and S_2, a computation of on average 50 steps. This seems inefficient, knowing that only one vertex a might survive. Therefore, FIND_CANDIDATE_SET is further improved by making a detour in this case. Instead of intersecting S_1 and S_2, it simply checks whether $w_2 \in Adj(a, c_2)$. This is a computation of $|Adj(a, c_2)| = 1$ step for the example. The detour is only applied when for some $i < j : |S_j| << |S_i|$. In this way, the typical case order FIND_CANDIDATE_SET is strongly improved.

For example, when using the recognition algorithm for a circuit at the transistor level, sets that are nearly always bypassed via this detour, are the adjacency sets of ground and supply nets of the source-drain class, because their degrees are large. On the other hand, the degrees of the gate connections of these nets are usually low, so these sets are hardly ever bypassed.

4.3.7 The ordering of a search list

The previous sections describe the decomposition of the sub-circuit recognition problem into a set of demands for a given search list and efficient backtracking for a given set of demands. This section highlights issues concerning the best ordering method for demand set $\mathbf{d}$. We have argued in Section 4.3.2, that the size of the traversed part of the search tree is strongly influenced by the chosen permutation called the search order $\mathbf{p}$. For the sub-circuit recognition problem, the search order $\mathbf{p}$ has been transformed into the order of the search list $\mathbf{s}_k$. Indeed, in the actual sub-circuit recognition implementation, the search list ordering method has been the most decisive part for the efficiency. The outcome of the ordering method called search list generation, is a prefix connected ordering of the template vertices V (see Definition 4.34, Section 4.3.4), the actual search list $\mathbf{s}_k = (v_1, \ldots, v_k)$, $k = |V|$, which directly corresponds to the demand function set $\mathbf{d}$ as described in Section 4.3.4. In general, there are $k!$ orderings of V. For a typical template circuit, considering every order is therefore impracticable.

The aim of the ordering is to reduce the traversal size of the corresponding search tree (see Definition 4.35, Definition 4.30). By placing strongly selective demands at the beginning of the search list, and the barely selective demands towards the end, the traversed part of the search tree will start off narrow, and most branches deeper in the tree are likely to lead to a recognized instance of the template. For algorithms found in the literature, the search list is either determined by the user, or based on characteristics of the template circuit G only. The first option requires a very experienced user, while the later may lead to an algorithm that works for one main circuit and fails for another. In our approach, the search list depends on both the template G and the main circuit $\mathbf{G}$. This is realized by estimating heuristically the selectivity of each demand, based on G and $\mathbf{G}$. The selectivity of a demand will be formalized as the branching factor of a coordinate, i.e., a vertex. A high branching factor corresponds to low selectivity. The search list is therefore ordered from low branching factor values to high branching factor values.

The branching factor, an example

To introduce the branching factor, we reconsider the problem of Figure 4.5 and Figure 4.6. Suppose we must determine the best search order $\mathbf{p}$ for this problem. In analogy to assigning branching factors to vertices, we will

assign branching factor values to the variables y_1, y_2, y_3. Recall that for every search order $\mathbf{p}$, the search tree $G_{\mathbf{p}}(V_{\mathbf{p}}, E_{\mathbf{p}})$ (see Definition 4.35) is interpreted differently, and thus the traversal differs. The branching factors $\mathcal{U}_1(i)$, $(i = 1, 2, 3)$ are equal to the minimum traversal size of any search tree, starting from level 0, when $p_1 = i$. In order to compute $\mathcal{U}_1$, each variable y_i is considered at the first level, and the other two are considered at the next level. For $k = 3$, this leads to $3! = 6$ possibilities. However, since the problem is symmetrical w.r.t. y_1 and y_2, Figure 4.5 shows that $\mathcal{U}_1(3) = 6$ and Figure 4.6 shows that $\mathcal{U}_1(1) = \mathcal{U}_1(2) = 8$. Therefore, a decision based on y_3 at level 1 is preferred. After having selected $p_1 = 3$, our next concern is p_2. The branching factors $\mathcal{U}_2(i)$, $(i = 1, 2)$ are equal to the number of traversed edges of the search tree, starting from level 1, when $p_2 = i$, assuming $p_1 = 3$. Figure 4.6 shows that $\mathcal{U}_2(2)$=5, and from the symmetry between y_1 and y_2 follows that $\mathcal{U}_2(1)$=5. Indeed, the symmetry indicated directly that the choice for p_2 leads to the same number of traversed edges. After having chosen $p_2 = 2$ at random, there is only one alternative for p_3, $p_3 = 1$, making the evaluation of branching factor $\mathcal{U}_3(2)$ superfluous. So for the example, $\mathbf{p} = (3, 2, 1)$ is the final choice.

The example that has just been described, shows that every search tree should be traversed to determine an efficient search order. This method obviously puts the cart before the horse, since we want to use the search order to select a single search tree with a small traversal sub-tree. Therefore, one must rely on estimates of the branching factors that necessitate neither search tree enumeration nor search tree traversal.

For the given example, one may for example approximate the branching factor $\mathcal{U}_1(i)$, $i = 1, 2, 3$, by the number of traversed edges when going from level 0 to 1 only. Let an approximation of $\mathcal{U}_1(j)$ be denoted by $U_1(j)$. Now $U_1(3) = 1$, and $U_1(1) = U_1(2) = 2$ (see first level of Figure 4.5 and Figure 4.6). This leads to the same conclusion: $p_1 = 3$, and from the y_1, y_2 symmetry follows that $\mathbf{p} = (3, 2, 1)$ is a good choice, without traversing any search tree explicitly.

The branching factor functions

After informally having introduced the branching factor, the branching factor function will now be defined as follows.

Definition 4.43 *Branching Factor Function*
For a main circuit $\mathbf{G}$, a template circuit G, the search predicate $\mathbf{D}$ (Definition 4.33), a partial search order $\mathbf{s}_{j-1} = (v_1, \ldots, v_{j-1})$, $j \in \{1, \ldots, k\}$

which is a prefix connected subset of V, the Branching Factor Function $\mathcal{U}_j : V \setminus \mathbf{s}_{j-1} \rightarrow I\!N$ is defined for a vertex v as

$$\mathcal{U}_j(v) = \min_{\substack{\mathbf{s} \in \{\mathbf{s}_{j-1}\} \times \{v\} \times V^{k-j}, \\ \mathbf{s} \text{ is prefix connected}}} (\, |G_{\mathbf{s}}| - |G_{\mathbf{s}}|_j \,),$$

i.e., the minimum traversal size at levels $j, \ldots, k$, when $v_j = v$ in any search tree $G_{\mathbf{s}}$ having $\mathbf{s}_j = (v_1, \ldots, v_{j-1}, v)$. □

The domain of the function is the set of vertices excluding the vertices of the partial search order. In this definition, the number $|G_{\mathbf{s}}|_j$ is constant, since $\mathbf{s}_{j-1}$ is constant. When $\mathcal{U}_j(p) > \mathcal{U}_j(q)$ for a given $\mathbf{s}_{j-1}$ and for some $p, q \in V \setminus \mathbf{s}_{j-1}$, a search tree $G_{\mathbf{s0}}$, $\mathbf{s0}_j = (s_1, \ldots, s_{j-1}, q)$ with a smaller traversal size exists than the traversal size of any search tree $G_{\mathbf{s1}}$ with partial search order $\mathbf{s1}_j = (s_1, \ldots, s_{j-1}, p)$. In other words, the demand function d_j associated with template vertex q is more selective.

To compute a branching factor function value $\mathcal{U}_j(v)$ for some vertex at some level j, every search tree should be traversed. Since the computation is impracticable, approximate functions $U_j : V \setminus \mathbf{s}_{j-1} \rightarrow I\!R \bigcup \{0\}$ of the branching factor functions are used to efficiently find a search order with a search tree having a small traversal size. When $U_j(p) > U_j(q)$, it should imply that $\mathcal{U}_j(p) > \mathcal{U}_j(q)$, so only the relative function values of U_j should resemble $\mathcal{U}_j$, the function values of a U_j should maintain the order of $\mathcal{U}_j$. In the rest of this thesis, the notion branching factor will refer to U_j, i.e., an approximation of $\mathcal{U}_j$. The co-domain of U_j, the non-negative real numbers, is an extension of the co-domain of $\mathcal{U}_j$ to allow real number approximations of natural numbers.

The example and notion of branching factor that were shown above suggest a recursive approach to compute the search list $\mathbf{s} = \mathbf{s}_k$. Therefore, in the following section, the definition of U_1, called the initial branching factor estimate, is described first to determine v_1, including an efficient computation method. In the next section, the rest of the search list is determined during a recursive traversal of the template circuit, by defining the branching factors U_j, $j = 2, \ldots, k$ along the way. It is also shown how these can be computed efficiently. At every level j, the vertex having lowest branching factor is selected as v_j. The approximations of branching factors used are crucial for the eventual recognition algorithm and are therefore explained in detail, including supportive examples.

4.3.8 The first search list element

The ordering of a search list is guided by a set of functions called branching factor estimates, introduced in the last section. The subject of this section is the initial branching factor. An efficient computation method is included, based on equivalence sets. The vertex having the minimum initial branching factor is considered as having the most selective demand d_1, and is therefore selected to be v_1. Finally, an improvement of the initial branching factor is given, called the *clock heuristic.*

The initial branching factor U_1
According to the definition of the first demand function d_1 (Definition 4.38), $d_1 = L_{v_1}$. Therefore, we will consider the true-set function of the local demand functions L_v, $v \in V$, $\mathcal{L} : V \rightarrow 2^{\mathbf{V}}$.

Definition 4.44 *Initial Candidates Function*
For a main circuit G, template circuit $\mathbf{G}$, the local demand functions L_v for $v \in V$ defined according to Definition 4.36, the Initial Candidates Function $\mathcal{L} : V \rightarrow 2^{\mathbf{V}}$ is defined for a template vertex v by

$$\mathcal{L}(v) = \{\ w \in \mathbf{V} \mid L_v(w)\ \}.$$

□

$\mathcal{L}(v)$ assigns the elements of $\mathbf{V}$ to a template vertex v that satisfy L_v. Since $|\mathcal{L}(v)|$ equals the number of traversed edges when going from level 0 to 1, assuming $v_1 = v$, a reasonable estimate for $\mathcal{U}_1$ is

$$U_1(v) = |\mathcal{L}(v)| \tag{4.30}$$

Whether this function can be computed efficiently is addressed next.

An efficient computation for U_1
The computation of U_1 is dominated by the initial candidates function $\mathcal{L}$. The computation of $\mathcal{L}$ takes $O(|V| * |\mathbf{V}|)$ operations when implemented as evaluating every L_v ($v \in V$) for every $w \in \mathbf{V}$, and assuming that $L_v(w)$ can be evaluated in $O(1)$ operations. However, as will described next, the computation can be done much more efficiently. According to Definition 4.36, two local demand functions L_p and L_q represent the same relation, when the attribute function A, the type function T and the degree function for every class of p and q are equal. In that case, $\mathcal{L}(p) = \mathcal{L}(q)$, and the

computation of $U_1(p)$ makes computing $U_1(q)$ superfluous. According to Definition 4.3, Section 4.2.1, the quotient set $V/\mathcal{L}$ is a partition of V into equivalence sets. The vertices in an equivalence set have equal local demand functions L_v. Hence only one computation of $\mathcal{L}(v)$ per set of the quotient set $V/\mathcal{L}$ is necessary.

According to Definition 4.36, every L_v function, $v \in V$ is uniquely defined by the 3-tuple $(T(v), A(v), \{\ (c, DEGREE(v,c)) \mid c \in \gamma(v)\ \}\ )$. Therefore, by computing every 3-tuple per $v \in V$, the quotient set $V/\mathcal{L}$ can be computed without explicitly evaluating any $\mathcal{L}(w)$. This results in $O(|G|) = O(|V| + |E|)$ operations to compute $V/\mathcal{L}$.

To compute U_1, we must compute one $\mathcal{L}(v)$ explicitly per set of the quotient set $V/\mathcal{L}$. Since for electronic circuits many similar constructions are used in a design, the number of equivalence sets in $V/\mathcal{L}$ is small, typically $O(\log|V|)$ or even less. For instance, a CMOS template of an adder will only contain PMOS and NMOS transistors, with only a few variations in width and length attributes. The number of different degree values for nets is also limited, because of fan-in and fan-out restrictions.

In total, the computation of U_1, involves computing $V/\mathcal{L}$, and for each set we must compute one $\mathcal{L}(v)$. Therefore, the number of operations is $O(|G|) + O(|V/\mathcal{L}| * |\mathbf{V}|)$. By assuming that $|G| < |\mathbf{V}|$ and $O(|V/\mathcal{L}|) \approx O(\log|V|)$, this reduces to

$$O(U_1) \approx O(\log|V| * |\mathbf{V}|), \tag{4.31}$$

which is much better than $O(|V| * |\mathbf{V}|)$. For the implemented recognition algorithm, this computation forms a major part of the run time, so the computational speedup by using equivalence sets is of great importance.

The search list initialization algorithm

The algorithm that initializes the search list generation process and computes the initial branching factor U_1 is given in Figure 4.14. It computes a partitioning of V over equivalence sets, $V/\mathcal{L}$, the initial branching factors $U_1(v)$, the 3-tuple per vertex, the start vertex v_1 and its candidates $\mathcal{Y}_1$.
In the first for-loop the equivalence sets with respect to the initial candidates $\mathcal{L}$, being the true-set of L_v, are computed. The 3-tuples associating a vertex to its local demand L_v are also computed.
In the second for-loop, the candidates for any vertex of an equivalent set are computed. The function ANY applied to a set returns one item of the set. The first vertex of the search list, v_1 is set to a vertex having a minimum branching factor U_1. When $U_1(v_1) = 0$ for this vertex, the number of

Procedure INITIALIZE_SEARCH_LIST_GENERATION

for all $v \in V$ do /* Compute $V/\mathcal{L}$ */

$3\_tuple[v] := (\ T(v),\ A(v),\ \{(c, DEGREE(v,c)) \mid c \in \gamma(v)\}\ )$
$eqset[3\_tuple[v]] := eqset[3\_tuple[v]] \cup \{v\}$

endfor

$U_{min} := +\infty$

for all $set \in eqset$ do /* Compute $Y = \mathcal{L}(v)$ and U_1
per equivalence set and ...*/

v :=ANY(set)
$L := L_v$ /* assign a function */
$Y := \{w \in \mathbf{V} | L(w)\}$
$U_1 := |Y|$

if $U_1 < U_{min}$
then /* ... select v_1 and $\mathcal{Y}_1$ in passing. */
$U_{min} := U_1$
$v_1 := v$
$\mathcal{Y}_1 := Y$
endif

endfor

if $U_1 = 0$ then exit("No matches")

endprocedure

Figure 4.14: *Initialization for search list generation.*

possible matches is zero, and the recognition process exits. The actual set of candidates $\mathcal{Y}_1$ for v_1, $\mathcal{L}(v_1)$, is also stored for later use in the depth-first search process.

The clock heuristic

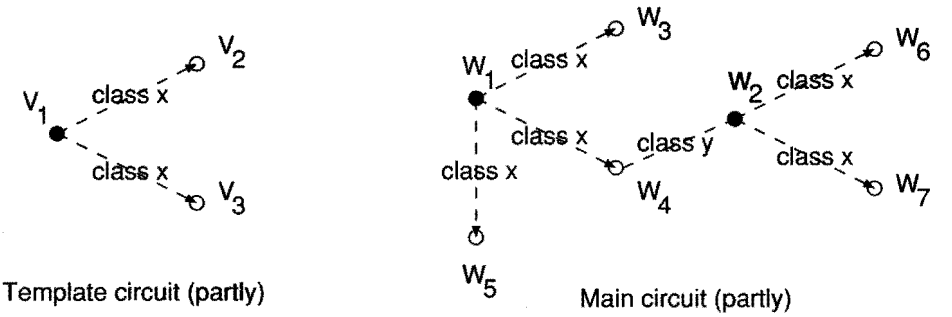
An improvement for the U_1 estimate, called the clock heuristic, is now described. The improvement was introduced as a result of inefficient recognition, that was caused by a circuits clock net.

Although Equation 4.30 seems a natural choice for estimating the relative size of the traversal sub-tree, it ignores the different traversal sizes of the search trees as a result of connectivity permutations, as shown in Figure 4.15. When the template vertex v_1 is selected for s_1 in this figure, both the main circuit vertices w_1 and w_2 are a matching candidate. In the figure, only the indices of the main circuit vertices are indicated. The traversal size of search tree sprouting from w_1 is 3 times larger (6 : 2) than for w_2. This directly relates to the permutation of *class* x connections between v_1, v_2 and v_1, v_3. Starting from w_1, $\binom{3}{2} = 3$ combinations are possible, i.e., (w_3, w_4), (w_3, w_5), (w_4, w_5) for (v_2, v_3). Starting from w_2, only $\binom{2}{2} = 1$ continuation is possible, i.e., (w_6, w_7) for (v_2, v_3). Since different labels (classes) are independent, the branching effect is multiplied for different classes. Therefore, each candidate in $\mathbf{V}$ is weighted according to the product of the number of possible connections for each class in the improved definition for the initial branching factor function:

$$U_1(v) = \sum_{w \in \mathcal{L}(v)} \prod_{c \in \gamma(v)} \binom{\mathbf{DEGREE}(w, c)}{DEGREE(v, c)} \tag{4.32}$$

For components and internal nets, the weight is equal to 1, since the degree per class, d, in template and main circuit must be the same and $\binom{d}{d} = 1$. Therefore, the weight factor only enhances the branching factor for external net vertices.

In the algorithm of Figure 4.14, the addition of the clock heuristic is simple. Only the right-hand side of the assignment of U_1 should be replaced by the right-hand side of Equation 4.32. The efficiency of the algorithm is hardly affected by the improvement.



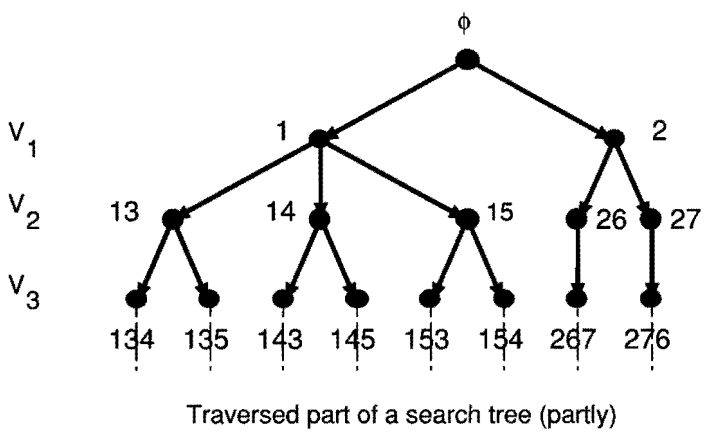


Figure 4.15: *The clock heuristic factor. Part of the traversed search tree is shown for* $s_3 = (v_1, v_2, v_3)$. *The numbers in the tree denote the main circuit vertices, i.e., 13 denotes* (w_1, w_3). *When* $s_3 = (v_1, v_3, v_2)$, *a similar partial tree would result. Since the traversed part sprouting from* w_1 *is* $\binom{3}{2} = 3$ *times larger than the traversed part from* w_2, *the contribution of candidate* w_1 *to the initial branching factor* U_1 *should be 3 times larger than the contribution of candidate* w_2.

4.3.9 Ordering the rest of the search list

Starting from v_1, the rest of the search list is determined by traversing recursively the rest of template circuit. At every recursion level j, the next, as yet unselected, vertex having minimum branching factor U_{j+1} is selected. The focus of this section is on the definition of the branching factors U_{j+1}. An efficient algorithm for computing the factors is described in the next section.

After defining the induction step for finding the next v_{j+1} in global terms, we will describe two estimation heuristics, called the search list connection count heuristic, and the parallel heuristic. Since the heuristics are essential for the efficiency of the recognition algorithm, they are explained in detail. Based on these heuristics and on the the already defined initial branching factors $U_1(v)$, the branching factors $U_{j+1}(v)$ definition will be given.

The induction step

This part describes the global scheme of the recursive search list generation, based on the selection of v_1 as described in the previous section. The induction step entails finding a suitable v_{j+1}. Candidates for selection are defined as follows.

Definition 4.45 *Border Set*

For a template circuit G, for $j \in \{1, \ldots, k-1\}$, a partial search list $\mathbf{s}_j$, the Border Set B_j, $B_j \subset V$ is defined by

$$B_j = \bigcup_{v \in \mathbf{s}_j} \bigcup_{c \in \gamma(v)} (\, Adj(v,c) \setminus \mathbf{s}_j \,).$$

The adjacency function Adj of a circuit is defined in Definition 4.25. The present classes function function γ of a circuit is defined in Definition 4.24. □

The border set B_j enumerates all neighbor vertices adjacent to the vertices of the partial search list $\mathbf{s}_j$. Since every $v \in \mathbf{s}$ must be connected to at least one of its predecessors, v_{j+1} must be a member of B_j.

Suppose the definition for the branching factor function $U_{j+1} : B_j \rightarrow \mathbb{R}^+ \cup \{0\}$ is given. Since U_{j+1} estimates the minimum traversal size for level $j+1, \ldots, k$. the induction step computes v_{j+1} by selecting an element of B_j having minimum U_{j+1}.

The rest of this section concerns the actual definition of the branching factors U_{j+1}, $j = 1, \ldots, k-1$. Two estimation heuristics form the basis of U_{j+1}, called the search list connection count heuristic and the parallel heuristic. They are described first.

The search list connection count heuristic

In the backtracking algorithm, the FIND_CANDIDATE_SET call mainly computes intersections of adjacency sets of the main circuit set P_j, (see Section 4.3.6, Definitions 4.40, 4.42). Since the intersection of two sets is in general smaller than the original sets, the number of intersected sets has a negative correlation with the size of the resulting set. The number of intersected sets is equal to the number of pairs in P_j. So statistically one may assume that the more elements in P_j, the less branching may be expected. However, P_j itself is unknown, since this heuristic is applied during the determination of the search list, i.e., before the backtracking starts. This can be solved since $|P_j| = |b_j(v_{j+1})|$ by definition (Definition 4.39), so the number of connections to $\mathbf{s}_j$ for a vertex v, called the search list connection count, is equal to $|b_j(v)|$.

The parallel heuristic

A second estimation heuristic is called the parallel heuristic. For this heuristic, the border set B_j (Definition 4.45) is partitioned into sets, called parallel sets. A parallel set has the property that the elements in a parallel set, being a member of B_j are not only candidate for v_{j+1}, but all lead to the same demand function d_{j+1}. This is similar to the partitioning $V \setminus \mathcal{L}$ of the previous section, in which each set also contained vertices with equal demand function d_1. Since two vertices in a parallel set lead to the same demand function, they have equal local demand functions, and they have equal connectivity to $\mathbf{s}_j$ (see Definitions 4.38, 4.36, 4.37). This explains why the sets are called parallel sets, since they appear to be parallel for the neighbors in the partial search list $\mathbf{s}_j$. The following example introduces both the notion of a parallel set, and the effect for traversed sub-tree of a search tree.

Example

Consider the example of Figure 4.16, showing a template circuit, a main circuit, and the traversed sub-trees of two search trees corresponding to two search list orders. The identification of the search tree vertices are

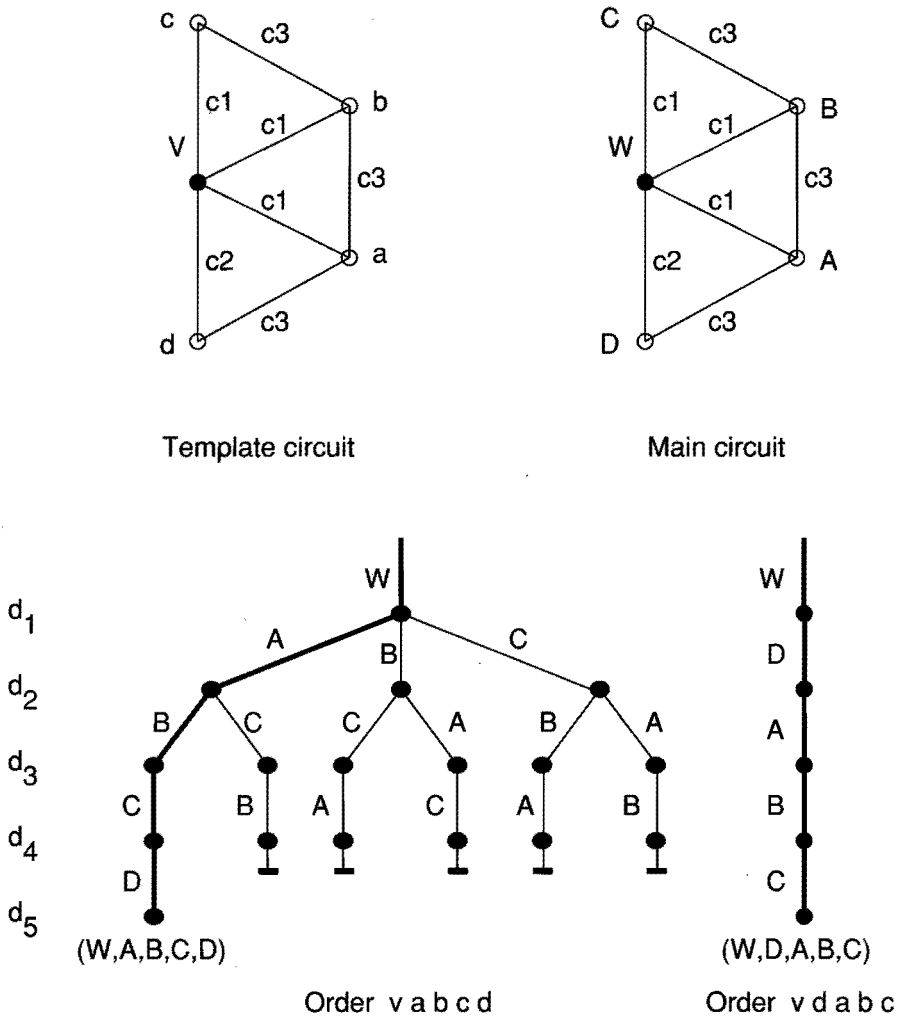


Figure 4.16: *Example showing the influence of the parallel heuristic. At the bottom, the traversed part of the search trees are shown for two search orders. The two bold lines correspond to the resulting match. The second search list needs less branching to find the match.*

only given explicitly for the equivalent solutions (W, A, B, C, D) and (W, D, A, B, C). The vertices are equal to the names enumerated along the paths of search tree. Suppose the search list for the template drawn at the top left-hand side is ordered (v, a, b, c, d). Starting from v mapped to W, the search tree will grow rapidly during the backtracking process as shown on the bottom left-hand side, since all vertices A,B,C are equally acceptable, although in the end only one branch will result in a match (W, A, B, C, D). Coming from W, the vertices A,B,C are not distinguishable, since they are all connected in parallel. Therefore, the branching size is equal to 3!, the factorial of the number of parallel neighbors (a,b,c) of v. Suppose the search list is ordered (v, d, a, b, c), as is drawn in Figure 4.16 (right-hand bottom). Starting from v, the match (W, D, A, B, C) is found without superfluous branching, because after having selected d, c has different connections to $\mathbf{s}_2$ than a and b, and will therefore not lead to parallel branching. The aim of the parallel heuristic is therefore, to select non-parallel neighbors of the selected partial search list $\mathbf{s}_j$ first, to omit branching caused by parallel neighbors. For the current example, the parallel heuristic should lead to the second search list.

Definition of the parallel heuristic

With this example in mind, the parallel heuristic will be defined for a general case. The general definition that will be given is applied again to the example of Figure 4.16. We want to partition the border set elements B_j of $\mathbf{s}_j$ into sets of vertices leading to equal demand functions d_{j+1}, to be able to count the number of parallel neighbors. In Section 4.3.6, we have seen that connectivity demand part of d_{j+1}, F_{j+1}, is determined by the template neighbor function call $b_j(v_{j+1})$ (Definition 4.39, 4.40, 4.41), enumerating the relevant labeled (multi-)edges between v_{j+1} and $\mathbf{s}_j$. Recall the definitions of the search tree vertices $V_j = \mathbf{V}^j$ (Section 4.3.4). Since v_{j+1} has not yet been chosen, d_{j+1} (see Definition 4.38), is not yet defined. Therefore, we define proto-demand functions as follows.

Definition 4.46 *Proto-demand Function*

For a main circuit $\mathbf{G}$, a template circuit G with external net set NE, for a $j \in \{1, \ldots, k-1\}$, a partial search list $\mathbf{s}_j$, a vertex $v \in B_j$, the Proto-demand function $d_{(j+1,v)} : \mathbf{V}^j \rightarrow \{True, False\}$ is defined to be equal to the function d_{j+1}, when $v_{j+1} = v$. □

The definition of a demand function d_{j+1} is determined by the functions L_v and F_{j+1}, see Definitions 4.36, 4.37. L_v is determined by the 3-tuple

$$(T(v), A(v), \{\ (c, DEGREE(v,c)) \mid c \in \gamma(v)\ \}\ ),$$

as described in the previous section. The definition of F_{j+1} is determined by the template neighbor function call $b_j(v)$, Definition 4.39, which enumerates the relevant labeled (multi-)edges between v and $\mathbf{s}_j$. Therefore, a function $d_{(j+1,v)}$, is completely determined by the 3-tuple of L_v and $b_j(v)$.

Definition 4.47 *Parallel Function*
Let a main circuit $\mathbf{G}$, a template circuit G with external net set NE, a $j \in \{1, \ldots, k-1\}$, a partial search list $\mathbf{s}_j$ be given. Let the true-set function $\delta_{j+1} : B_j \to 2^{\mathbf{V}^{j+1}}$ assign the true-set of $d_{(j+1,v)}$ to $\delta_{j+1}(v)$ for a vertex $v \in B_j$, i.e., the set of all search tree vertices at level $j+1$ satisfying $d_{(j+1,v)}$. The quotient set B_j/δ_{j+1} partitions B_j into sets of equal $d_{(j+1,v)}$ functions (see Section 4.2.1). The canonical function $Par_j : B_j \to B_j/\delta_{j+1}$, called the Parallel Function, is defined by

$$Par_j(v) = \{u \in B_j \mid \forall \mathbf{w} \in \mathbf{V}^{j+1} : d_{(j+1,v)}(\mathbf{w}) = d_{(j+1,u)}(\mathbf{w})\}.$$

□

In analogy to the quotient set $V/\mathcal{L}$ (Section 4.3.7) partitions V into sets of vertices having equivalent local demand functions L_v, Par_j assigns to each border set element v a subset of the border set where the elements have equivalent proto-demand functions.

For a vertex $v \in B_j$, $|Par_j(v)|$ indicates the number of indistinguishable vertices, with respect to $d_{(j+1,v)}$. Therefore, the higher $|Par_j(v)|$, the larger the traversal size that may be expected.

Example reviewed
Figure 4.16 illustrates the above definitions. In Figure 4.16, the initial partial search list $\mathbf{s}_1 = (v)$ and $B_1 = \{a, b, c, d\}$. The template neighbor function is given by $b_1(a) = b_1(b) = b_1(c) = \{(v, c_1, 1)\}$ and $b_1(d) = \{(v, c_2, 1)\}$. The parallel set functions are given by $Par_1(a) = Par_1(b) = Par_1(c) = \{a, b, c\}$ and $Par_1(d) = \{d\}$. The template vertices a, b, c are connected in parallel to v, hence $B_1/\delta_2 = \{\{a, b, c\}, \{d\}\}$. With the parallel heuristic, for the first induction step to compute v_2, it is noted that $|Par_1(d)| < |Par_1(x)|$ for $x = a, b, c$, hence selection $\mathbf{s}_2 = (v, d)$ is made.

At this point, $B_2 = \{a, b, c\}$ and $B_2/\delta_3 = \{\{b, c\}, \{a\}\}$. Vertex a is not equivalent to b and c, since it is also connected to $v_2 = d$. In the next induction step, we compute v_3. Now $|Par_2(a)| < |Par_2(x)|$ for $x = b, c$, and $\mathbf{s}_3 = (v, d, a)$.
Repeating this process twice more results in $s_5 = (v, d, a, b, c)$. On the left side of Figure 4.16 we see that this complete search order prevents branching due to temporal parallelism.

The branching factor function estimates
After having defined several estimation parameters, we will now define the branching factor functions U_{j+1}. From the discussion so far, the following conclusions can be drawn. For $v \in B_j$, the branching factors $U_{j+1}(v)$ should be **increased** when the initial branching factor $U_1(v)$ is high, and when the number of vertices in the associated parallel set, $|Par_j(v)|$, is large. The branching factors $U_{j+1}(v)$ should be **decreased** when the number of connections to predecessors, $|b_j(v)|$, is large. The following definition agrees with these requirements.

Definition 4.48 *Branching Factor Estimate*
Let a main circuit $\mathbf{G}$, a template circuit G with external net set NE, a $j \in \{1, \ldots, k-1\}$, a partial search list $\mathbf{s}_j$ be given. The template neighbor function b_j is defined according to Definition 4.39, the parallel function Par_j is defined according to Definition 4.47, the initial branching estimate U_1 is defined according to Equation 4.32. The Branching Factor Estimate $U_{j+1} : B_j \rightarrow I\!R^+ \cup \{0\}$ is defined for each border set vertex as

$$U_{j+1}(v) = \begin{cases} U_1/|b_j(v)| & \text{if } |Par_j(v)| = 1 \\ U_1 * |Par_j(v)|! & \text{if } |Par_j(v)| > 1 \end{cases}$$

□

The following section describes the algorithm to generate the search list. It will be shown that despite their complex definitions, the factors $|b_j(v)|$ and $|Par_j(v)|$ can be computed efficiently.

4.3.10 The iterative search list generation algorithm

This section describes in more detail how the rest of the search list is ordered according to the description of Section 4.3.9, and focusses on an efficient implementation to compute the branching factor functions U_{j+1}, $j = 1, \ldots, k-1$. After the ordering algorithm is described the special case of template vertices having only one initial candidate is briefly explained.

Introduction

After having selected $\mathbf{s}_1 = (v_1)$ according to the algorithm in Figure 4.14, Section 4.3.8, the rest of the search list can be generated in a for-loop, $j = 1, \ldots, k-1$, whereby each iteration produces the next element of the search list, v_{j+1}, and its demand function d_{j+1}. Before selecting the next vertex v_{j+1} and d_{j+1}, the information needed to compute the required branching factor estimate U_{j+1} (see Definition 4.48), the functions b_j (see Definition 4.39) and Par_j (see Definition 4.47) are computed with respect to the current $\mathbf{s}_j$.

When describing the initialization of the search list generation (Section 4.3.8), we have shown that the quotient set $V/\mathcal{L}$ is efficiently computable by using selection based on the 3_tuple $(T(v), A(v), \{\ (c, DEGREE(v,c)) \mid c \in \gamma(v)\})$ per $v \in V$, without explicitly evaluating any $\mathcal{L}(w)$. Likewise, the quotient set B_j/δ_{j+1} (see Definition 4.47), and thus Par_j, can be computed efficiently by selecting on a 4_tuple_j, consisting of $b_j(v)$ and the (already computed) 3_tuple, per $v \in B_j$, without explicitly evaluating any $\delta_{j+1}(v)$.

The algorithm

The total search list generation algorithm is described in Figure 4.17, and is illustrated in Figures 4.18 and 4.18a. The notions B_j, Par_j, b_j, 4_tuple_{j+1} for different j values, all share respectively the same variables, B, Par, b, 4_tuple in the algorithm. $Par_j(v)$ is implemented by an indirect reference $Par[4\_tuple[v]]$. After the initialization of the search list generation (Section 4.3.8, Figure 4.14), the borderset B variable, the argument-value tables of the parallel set functions Par and template neighbor functions b are initialized. The $4\_tuple_1(v)$ are defined by assignment to ($\emptyset$,3_tuple(v)) for all $v \in V$. The 3_tuple(v) values, $\mathbf{s}_1 = (v_1)$, U_1, and the initial candidates of v_1 Y_1, have been defined by the INITIALIZE_SEARCH_LIST_GENERATION procedure (see Figure 4.14). In the main for-loop, $v_2, \ldots, v_k$ and $d_2, \ldots, d_k$ are determined.

The following notions are computed incrementally in the body of the loop:

- the border set B_j, stored in variable B,
- the template neighbor function b_j, stored in variable b,
- the parallel sets function Par_j, stored in variable Par,

Procedure SEARCH_LIST_GENERATION(G, $\mathbf{G}$)

INITIALIZE_SEARCH_LIST_GENERATION /* See text */

$\mathtt{B} := \emptyset$ /* Var initializations */
$\mathtt{b} := \emptyset\_table$
$\mathtt{Par} := \emptyset\_table$

for $v \in V$ do /* Assign 4_tuple$_1$ */

$\mathtt{4\_tuple}[v] := (\emptyset,\ 3\_tuple[v])$

endfor

for $j = 1$ to $k - 1$ do /* Main for loop */

$\mathtt{Par}[\mathtt{4\_tuple}[v_j]] := \mathtt{Par}[\mathtt{4\_tuple}[v_j]] \setminus \{v_j\}$
$\Delta\mathtt{B} := \bigcup_{c \in \gamma(v_j)} Adj(v_j, c) \setminus \mathtt{s}_j$

for all $v \in \Delta\mathtt{B}$ do

$\mathtt{Par}[\mathtt{4\_tuple}[v]] := \mathtt{Par}[\mathtt{4\_tuple}[v]] \setminus \{v\}$
$\mathtt{b}[v] := \mathtt{b}[v] \cup \{(v_j, c, m) \mid c \in \gamma(v) \wedge m = \mu_{v_j}(Adj(v, c)) \wedge m > 0\}$
$\mathtt{4\_tuple}[v] := (\mathtt{b}[v],\ 3\_tuple[v])$
$\mathtt{Par}[\mathtt{4\_tuple}[v]] := \mathtt{Par}[\mathtt{4\_tuple}[v]] \cup \{v\}$

endfor

$\mathtt{B} := \mathtt{B} \cup \Delta\mathtt{B} \setminus \{v_j\}$
if $\mathtt{B} = \emptyset$ then exit("G is not connected")
$v_{j+1} := ANY(\min_U(\mathtt{B}))$ /* see Equation 4.48 */
$d[j+1]$:=MAKE_FUNCTION($\mathtt{4\_tuple}[v_{j+1}]$)

endfor

endproc

Figure 4.17: *Search list generation algorithm.*

- the 4_tuple_{j+1} relation (corresponding to δ_{j+1}), stored in variable 4_tuple.

This means that each computation is based on the results of the previous step, B_{j-1}, b_{j-1}, Par_{j-1} and 4_tuple_j. This approach utilizes the local character of every update, as illustrated in Figures 4.18 and 4.18a, showing all variable values during the execution of the algorithm, for the example of Figure 4.16. In Figures 4.18 and 4.18a, the bold dots indicate the search list, and the hollow dots indicate the border set B_j. To show the effect of the connectivity, the example ignores the *3_tuple*, so that $4\_tuple_{j+1} \equiv b_j$. For Par_j, first the vertex v_j is removed from the set corresponding with the old value of *4_tuple*.Each j^{th} main loop iteration must compute b_j, 4_tuple_j and Par_j for every $v \in B_j$. But many of the argument-value pairs of the functions of step j are equal to the argument-value pairs computed in previous steps. Only for the vertices adjacent to v_j, and of course v_j itself, need the function value be recomputed. Therefore, we will only consider the vertices of the set variable ΔB. The variable ΔB enumerates the affected neighbors of s_j. By using the functions $\gamma(v_j)$ and $Adj(v_j, c)$ (see Section 4.2.1), ΔB is quickly found based on local data only. In the tables on the right side of Figures 4.18 and 4.18a, the border edges that are not connected to the previously selected v_j are surrounded by square brackets, since their *4_tuple* and b values are unchanged and therefore not recomputed. For example in step j=2, $\Delta B = \{a\}$, so *4_tuple*[b], *4_tuple*[c], b[b], b[c] are not recomputed. Although the example shows only a few unchanged vertices, most of the border sets are unchanged for a typical template, because the average template is larger.

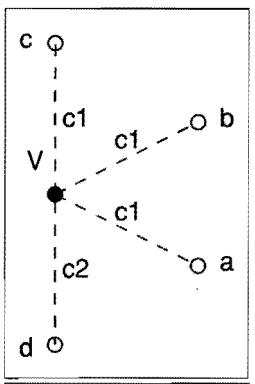
Next, the variable ΔB is used in the second level for-loop to update only the relevant argument-value pairs of b_j, 4_tuple_{j+1}, and Par_j. Figures 4.18 and 4.18a show the values for the current example.

After the loop, the border set is constructed, from which the next search list vertex v_{j+1} will be selected. When B_j becomes empty before $j = k-1$, the template graph G is not a connected graph, which is a violation of the sub-graph recognition precondition, leading to a premature exit of the program.

Now, the branching factors U_{j+1} can be computed using Equation 4.48. For example in Figures 4.18 and 4.18a, in step j=2 both cases of Equation 4.48 are used. For a, $|Par_2(a)| = 1$, hence $U_3(a) = U_1(a)/|b_2(a)| = 3/2$. For b, $|Par_2(b)| = 2$, hence $U_3(b) = U_1(b) * |Par_2(b)|! = 3 * 2! = 6$. Similarly for c, $U_3(c) = 6$. Although it has not been elaborated in the algorithm,

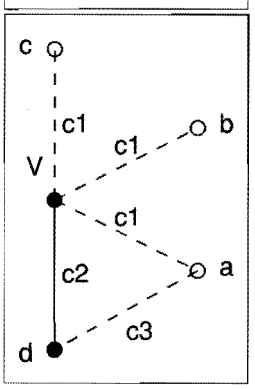
obviously only the branching factors of vertices with changed b_j and Par_j parameters need to be recomputed.

Hereafter, the next v_{j+1} is computed with the function $\min_U$, returning the set of vertices having minimum branching factor. MAKE_FUNCTION creates the corresponding demand function d_{j+1} from a 4_tuple. d_{j+1} will be used by the FIND_CANDIDATE_SET function in the backtracking algorithm (Figure 4.11).



Step j=1

$v \in B_1$:	$a\ b\ c$	d
$4\_tuple_2(v)$:	$\{(v, c1, 1)\}$	$\{(v, c2, 1)\}$
$Par_1(v)$:	$\{a, b, c\}$	$\{d\}$
$U_2(v)$:	$3 \times 3! = 18$	$1/2$
$\vdash \mathbf{s}_2 = (v, d)$		



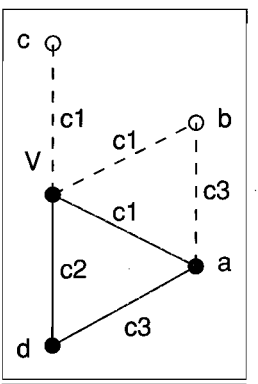
Step j=2

$v \in B_2$:	a	$[b]\ [c]$
$4\_tuple_3(v)$:	$\{(v, c1, 1), (d, c3, 1)\}$	$\{(v, c1, 1)\}$
$Par_2(v)$:	$\{a\}$	$\{b, c\}$
$U_3(v)$:	$3/2$	$3 \times 2! = 6$
$\vdash \mathbf{s}_3 = (v, d, a)$		

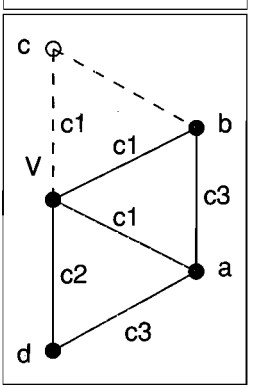
Figure 4.18: *The iteration steps for the search list generation algorithm of Figure 4.17. The figure is continued in Figure 4.18a. The iteration starts from* $v_1 = v$, *with* $4\_tuple_j \equiv b_j$ *and assumes* $U_1(a) = U_1(b) = U_1(c) = 3$ *and* $U_1(d) = 1$. *The selected partial search lists* $\mathbf{s}_j$ *for each step* j *are indicated by bold dots, open dots are members of* B_j. *The uninterrupted lines are the edges of partial template graph* G_j. *The dashed lines represent the template neighbor function* b_j. *The 4_tuple values of vertices between square brackets are unchanged in an iteration.*

Singly initial candidates

Although it has not been included in the pseudo-code of Figure 4.17, vertices that have only one initial candidate after the initialization are better processed slightly differently. Since they don't cause branching in the traversed part of the search tree, they can be successively chosen at the beginning of the search list. For these vertices, the demand that each ver-



Step j=3		
$v \in B_3$:	b	$[c]$
$4\_tuple_4(v)$:	$\{(v, c1, 1), (a, c3, 1)\}$	$\{(v, c1, 1)\}$
$Par_3(v)$:	$\{b\}$	$\{c\}$
$U_4(v)$:	$3/2$	$3/1 = 3$
$\vdash \mathbf{s}_4 = (v, d, a, b)$		



Step j=4	
$v \in B_4$:	c
$4\_tuple_5(v)$:	$\{(v, c1, 1), (b, c3, 1)\}$
$Par_4(v)$:	$\{c\}$
$U_5(v)$:	$3/2$
$\vdash \mathbf{s}_5 = (v, d, a, b, c)$	

Figure 4.18a: *Figure 4.18 continued.*

tex must be connected to at least one of its predecessors is dropped. The demand was only useful for being able to apply FIND_CANDIDATE_SET in an efficient way (see Section 4.3.4). In this case, the only possible candidate for a vertex has already been identified, and their mutual connectivity is checked immediately. This means that the singly matching vertices and their candidates are checked during search list generation, and are therefore removed from the search list on which depth-first search will be applied later. The v_1 and Y_1 relate to the first multiple matching vertex having minimum initial branching factor U_1, with more than one candidate.

4.3.11 The primary algorithm, an overview

The primary sub-circuit recognition algorithm will be completed in this section, after reviewing briefly the algorithms developed so far.

The algorithm in retrospect

The primary sub-circuit recognition problem is specified by a main circuit and template circuit. The solution set of the problem, called the matches, is a set of sub-circuits of the main circuit, that is implicitly defined by the true-set of the characteristic function called the isomorphism predicate (Definition 4.20). We have chosen backtracking by depth-first search as

```
Procedure PRIMARY_RECOGNIZE (G, G)

    SEARCH_LIST_GENERATION (G, G)

    for all cand ∈ 𝒴1

        w1:=cand
        MARK(cand)
        DEPTH_FIRST_SEARCH(1)
        UNMARK(cand)

    endfor

endproc
```

Figure 4.19: *The primary recognition algorithm. The backtracking top level is performed here, since the search list generation algorithm computes the top level candidates $\mathcal{Y}_1$ for v_1.*

the main method to compute the matches explicitly, since the method allows skipping of large parts of the search space (i.e., the domain of the isomorphism predicate). The translation of sub-circuit recognition into a backtracking problem, i.e., an ordered set of demand functions based on a search list, has been described in detail in Sections 4.3.3, 4.3.4 and 4.3.5. During backtracking, the demand functions are evaluated one by one, lead-

ing to a partial traversal of the associated search tree. The importance of the search list order for efficiency was emphasized. In Section 4.3.6 an efficient method and algorithm for finding candidates at every level in the search tree has been described. The subsequent section, Section 4.3.7, described how an efficient search list order can be achieved, by reducing the estimated traversal size of the search tree. In the ordering algorithm, the computations are organized to identify equivalent calculations before performing them, thus preventing the execution of equal computations where possible.

The remaining envelop algorithm

Now the parts can be put in place to make the overall PRIMARY_RECOGNIZE algorithm, see Figure 4.19. In the algorithm, the search list generation algorithm not only computes the search list, but also handles vertices that match only once (see previous section). Since the search list generation algorithm also computes the set of candidates for v_1, $\mathcal{Y}_1$ (see Figure 4.14), the algorithm also performs the first level of backtracking. The depth-first search process continues from v_2 onwards. Eventually, the complete set of isomorphisms is stored in the data structure, as the set of complete paths of the search tree. To get the solution set $\mathcal{M}$, the set of isomorphic sub-circuits must be derived from the set of isomorphisms, as described in Section 4.2.3, Definitions 4.21, 4.23. This operation is one of the topics of the next section.

4.4 Post-processing

In the next sections, several extensions will be discussed that enhance the primary algorithm from a fast but rigid algorithm into a flexible, fast and complete tool. This section describes the transformation of the set of isomorphism functions $\mathcal{S}$ (Definition 4.20, Section 4.2.3) into a set of matches $\mathcal{M}$. $\mathcal{S}$ is represented as a set of complete paths of the search tree, resulting from the primary recognize algorithm. According to the primary recognition problem definition, a match $\mathcal{G}$ is equal to the sub-circuit $\mathbf{G}|_{f(V)}$ of the main circuit $\mathbf{G}$ for an isomorphism $f \in \mathcal{S}$. Hence the set of all matches $\mathcal{M}$ of the template circuit G in the main circuit $\mathbf{G}$ is given by $\mathcal{M} = \{\mathbf{G}|_{\phi(V)} \mid \phi \in \mathcal{S}\}$. However, different isomorphisms may result into the same sub-circuit, i.e., $|\mathcal{M}| = |\mathcal{S}|$ need not be true. In general, two matches in $\mathcal{M}$ that may be partially, or completely overlapping. Suppose that the recognition algorithm is used to partition the main circuit into a set of sub-circuits, then clearly common vertices in overlapping matches complicate the partitioning. This section classifies this phenomenon, and shows classified matches may be processed depending on a simple user control mechanism. Furthermore, the run time consequences of the different cases are briefly described.

Definition 4.49 *Overlap*
Let $\mathcal{G}_1$, $\mathcal{G}_2$ be elements of the set of matches $\mathcal{M}$, with sets of component vertices $\mathcal{C}_1$, $\mathcal{C}_2$. The matches $\mathcal{G}_1$ and $\mathcal{G}_2$ Overlap when some of the component vertices in $\mathcal{G}_1$ are also component vertices of $\mathcal{G}_2$, i.e., $\mathcal{C}_1 \cap \mathcal{C}_2 \neq \emptyset$. The set $\mathcal{C}_1 \cap \mathcal{C}_2$ is called the *common components set.* □

Definition 4.50 *Touch*
Two matches $\mathcal{G}_1$, $\mathcal{G}_2 \in \mathcal{M}$ Touch when some of the net vertices $\mathcal{N}_1$ in $\mathcal{G}_1$ are also net vertices $\mathcal{N}_2$ of G_2, i.e., $\mathcal{N}_1 \cap \mathcal{N}_2 \neq \emptyset$. □

Since a template circuit is connected, a match is also connected. When every component vertex is connected to at least one net vertex, the following observations are valid.

1. When two matches $\mathcal{G}_1$ and $\mathcal{G}_2$ overlap, they also touch, because the nets connected to any component, as prescribed via its type property, are also part of the match.

2. When two matches $\mathcal{G}_1$ and $\mathcal{G}_2$ touch, they may or may not overlap.

In this respect, overlap is a stronger property than touch.

The remainder of this section describes the following two cases. Firstly, we describe the case when two matches overlap completely, i.e., the matches are identical. Secondly, we describe the case when the matches overlap partially.

4.4.1 Automorphisms

Definition 4.51 *Automorphisms*
An Automorphism is an isomorphism of a graph onto itself [Harar72.1]. The *automorphism set* of a graph is the total set of different isomorphisms of a graph onto itself. □

For example, PRIMARY_RECOGNIZE(G, G) computes the set of automorphisms of a template circuit G. When the automorphism set of G contains more than one element, G contains symmetry. Since a template G and all its matches $\mathcal{G} \in \mathcal{M}$ are isomorphic, it is obvious that if G contains symmetry, all matches contain symmetry as well. Let l be the number of automorphisms of G. Obviously each match of G in $\mathbf{G}$ is also found exactly l times. So the set $\mathcal{S}$ of isomorphisms can be partitioned into $|\mathcal{S}|/l$ subsets of which each subset leads to one identical match, G_j, $j = 1, \ldots, |\mathcal{M}|/l$. Depending on the application of the sub-circuit recognizer, the user might be interested not only in the set of matches, but also in $\mathcal{S}$, i.e., including all automorphisms. The user indicates his preference in the fourth list of the restrictions (see Figure 4.3), where a `"nil"` indicates no deletion of isomorphisms, and a `"T"` indicates deletion of all but one automorphism in the isomorphism set. In the latter case, `"T"`, the post-processing procedure partitions the isomorphisms according to equal component sets, and erases all but one element of these sets. Equal component sets imply equal net vertices sets, because the net connections are prescribed by the template circuit.

Now, the run time effect of automorphisms is briefly described. The sub-circuit recognizer will be slow when many automorphisms are present. In general, the size of the automorphism set can be large. The largest sets relative to the number of vertices are given by complete graphs [Harar72.2]. A complete graph is a graph in which each vertex is connected to every other vertex. A complete graph of n vertices has an automorphism set size of $n!$. Fortunately, electrical circuit template circuits are rarely complete graphs.

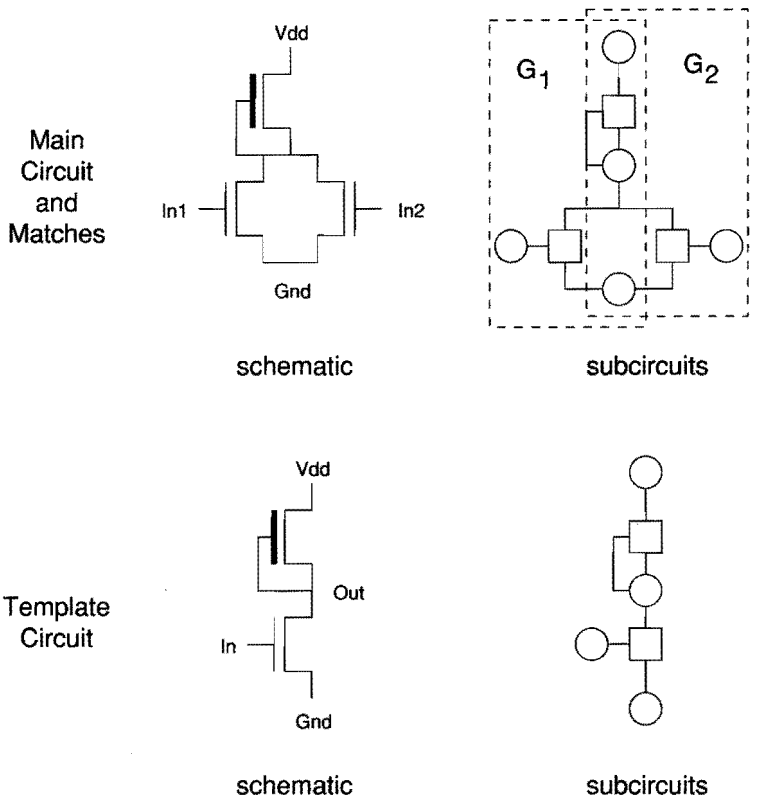


Figure 4.20: *Example of overlapping matches $\mathcal{G}_1$ and $\mathcal{G}_2$, when searching for an NMOS inverter in an NMOS NOR*

4.4.2 Partially overlapping matches

Fully overlapping matches have been described in the last section. Partially overlapping matches are the subject of this section. Partially overlapping matches are the second class of common vertices.

Definition 4.52 *Partial overlap*
Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be matches of a template G, with component sets $\mathcal{C}_1, \mathcal{C}_2$ respectively. $\mathcal{G}_1$ and $\mathcal{G}_2$ Partly Overlap when $\mathcal{C}_1 \neq \mathcal{C}_2$ and $\mathcal{C}_1 \cap \mathcal{C}_2 \neq \emptyset$. □

Depending on the intention of the recognition user, the partially overlapping matches might either be rejected or accepted.

Two cases, i.e., rejection and acceptance, of common components are now described by an example. In Figure 4.20, a rejection situation is de-

picted. The matches $\mathcal{G}_1$ and $\mathcal{G}_2$ can be identified in an NMOS NOR gate, but they should not be identified as NMOS inverters because they have a common NMOS transistor. Therefore, the post-processing should eliminate both matches for having a forbidden overlap.

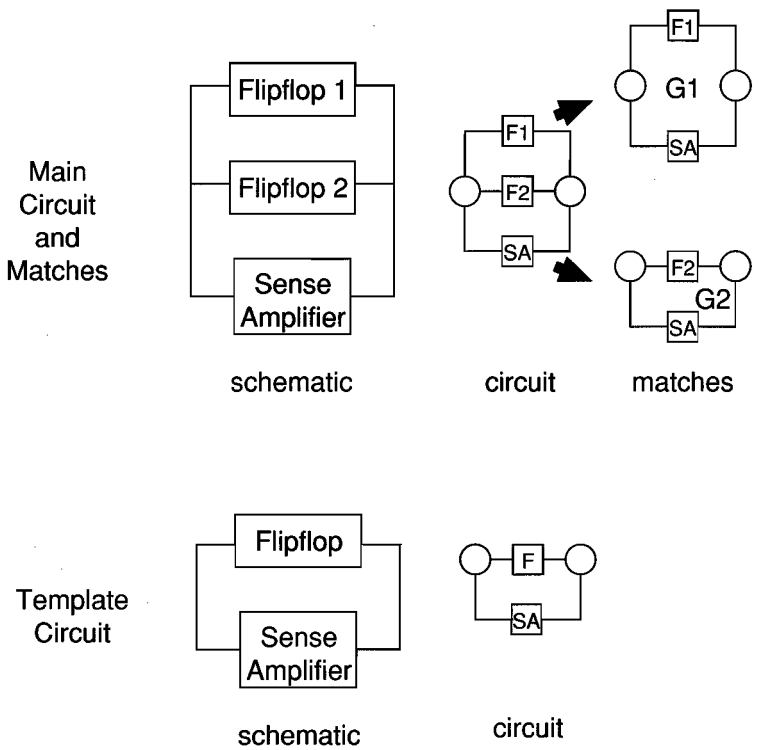


Figure 4.21: *Example of overlapping matches $\mathcal{G}_1$ and $\mathcal{G}_2$, when searching for a memory cell*

In other cases, overlap is allowed. Figure 4.21 shows a situation with accepted overlap. A flipflop in combination with a sense-amplifier forms a single bit memory-cell in a large memory block. In order to distinguish from ordinary flipflops in a design, one can add the sense-amplifier in the description of a memory-cell. Different memory-cells share the sense-amplifier, so the two matches $\mathcal{G}_1$ and $\mathcal{G}_2$ may overlap with the sense-amplifier component.

The user sets his requirement with respect to rejection or acceptance of overlap in the **third** list of the restrictions (see Figure 4.3, Section

4.2.3). This list enumerates the allowed *common components*. They describe whether for different matches, the matching main circuit components of template components allow overlap. Other component matches should not overlap. The algorithm that rejects or accepts a match on the basis of whether the overlapping components are all members of the common components is straightforward. For example, for two (primary recognize) matches $\mathcal{G}_1$ and $\mathcal{G}_2$ with component sets components $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively, a set of common components CC might be specified. During the post-processing, the following condition is tested:

$$(\mathcal{C}_1 \cap \mathcal{C}_2) \subset CC. \tag{4.33}$$

When Equation 4.33 is true, both $\mathcal{G}_1$ and $\mathcal{G}_2$ are accepted, otherwise, they are rejected as a match.

4.5 Extensions to the primary algorithm

This section describes the following extensions, adding more flexibility to the primary algorithm.

- Partially prescribed matches.
- External net merging.
- Exchangeability of terminal-classes groups.
- Diagnosis feedback.

The first extension allows the remaining part of a partially prescribed match to be found. The second and third extensions allow recognition of a combination of different versions of similar template circuits simultaneously. A family of templates can be recognized in a single execution, starting from one specification. The last extension shows how valuable feedback can be given when the actual recognition result is different from the expected recognition result. The extensions have proven to be very useful in practice.

4.5.1 Partially prescribed matches

A first extension to the primary algorithm is to allow a partial match to be prescribed beforehand. When a partial match is known, and one is interested in the full match, this is fully supported by the algorithm. In

a component type definition, net vertices present in the first list of a restriction, called *fixed nets* indicate that a template net vertex matches a main circuit net vertex with the same name, see Figure 4.3 (Section 4.2.3). Without describing it in more detail, any partial match can be prescribed and the algorithm can find the rest of the matches.

The adaptation of the PRIMARY_RECOGNIZE implementation to realize this extension is trivial, and is not included.

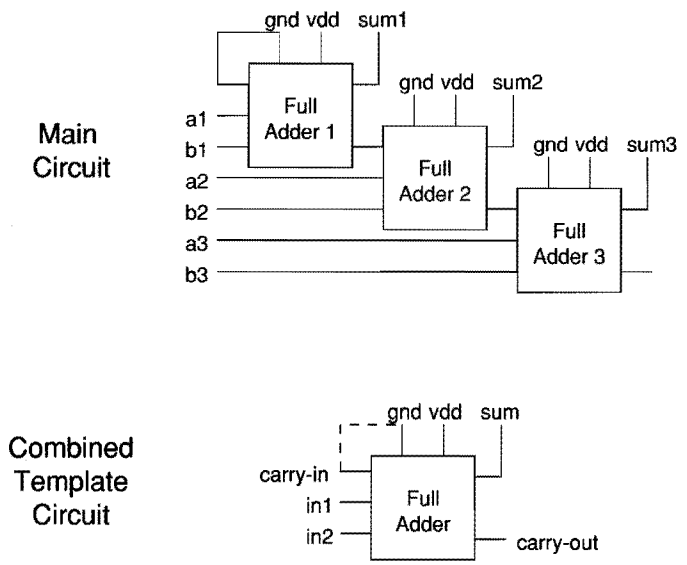


Figure 4.22: *Example of a merged net vertex in a 3-bits adder component. The carry-in and the gnd net are connected (merged) for Full Adder 1 and separate for the other components.*

4.5.2 External net merging

A second extension to the primary algorithm allows merged nets. A typical example of a situation where nets are merged, is given in Figure 4.22, showing a simple n-bits adder, composed of 3 full-adder cells. Because the carry-in of the least significant bit is connected to the ground net named gnd, this structure deviates from the other full-adder cells. When using the described primary algorithm only, one would have to define one template G_1 for the least significant bit full-adder, and another template G_2 for the other full-adders of the n-bits adder. It would be beneficial if this deviation

could be denoted more easily, and also if the recognition algorithm could handle both cases simultaneously.

The net vertices of a template, consisting of external net vertices NE and internal net vertices NI, are therefore grouped into sets of mergable nets. In a template definition, the external nets are defined in the second part of the restrictions, and directly grouped to indicate an optional merging. For Figure 4.22, the external nets groups are given by `((carry-in gnd)(vdd)(sum)(in1)(in2))`. Figure 4.3, Section 4.2.3, shows another example of inputs `(in1, in2)` that might be merged.

Implementation

The following indicates how the extensions to the primary algorithm to incorporate optional merged net vertices can be implemented. The merged nets option entails extension of the FIND_CANDIDATES algorithm, that is called by the the DEPTH_FIRST_SEARCH algorithm. In the primary version, any vertex of the main circuit was only allowed to be a candidate for one template vertex in a match. This was indicated by marking every candidate during the search tree traversal. However, when a potential candidate *cand* of a vertex v_j is already matched by a predecessor template vertex v_i $(i < j)$, but v_j and v_i are members of the same external nets group, *cand* is still acceptable as a candidate according to the current extension, provided it has the required neighbors, and the local demand call $L_{v_j}(cand)$ returns true. The implementation of this extension only requires a small adaptation of the FIND-CANDIDATES routine, but makes the use of the sub-circuit recognizer more general.

The number of circuits when merging nets

The number of different template circuits that are recognized simultaneously when merging nets can be very large. Figure 4.23 shows an example, only for a cmos inverter, of all 12 circuits that are recognized simultaneously when the external net vertices $\{in, out, gnd, vdd\}$ are considered as one merge-able group. In general, k merged net vertices in a group of n leads to $\binom{n}{k}$ different circuits, when the circuits do not contain symmetry. In total, including the original unmerged circuit, the number of different circuits equals

$$\sum_{k=2}^{n} \binom{n}{k} + 1 = 2^n - n. \tag{4.34}$$

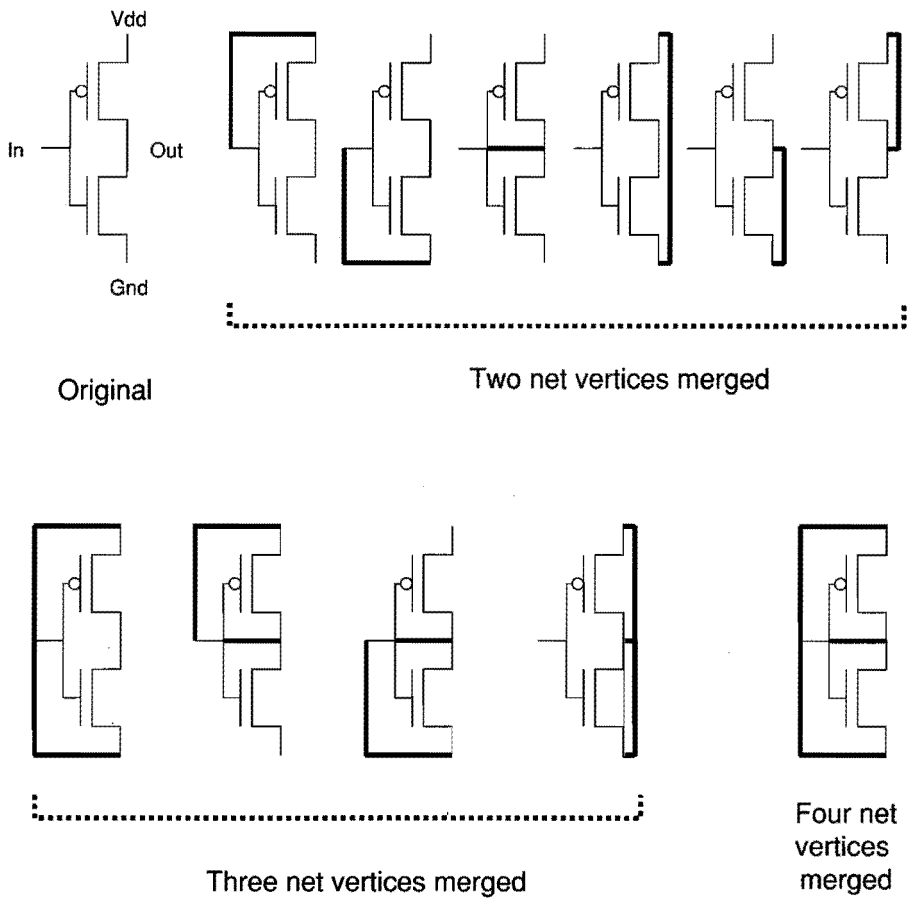


Figure 4.23: *The set of represented circuits when merging all net vertices of an inverter circuit. The bold lines are the short-cuts, to merge the nets. The circuits are subdivided into 3 groups having 2, 3 or 4 merged nets.*

For Figure 4.23, the external vertices are $\{in, out, gnd, vdd\}$, so n equals 4, and therefore 12 different circuits are shown. Equation 4.34 indicates that one must be selective when using merged nets. When a circuit contains symmetry, the number of different circuits is less than indicated in Equation 4.34.

4.5.3 Exchangeable terminal groups

The third extension concerns groups of exchangeable connections. It will be introduced by first reviewing the possibilities of exchangeability offered by the terminal classes definition, as included in the primary algorithm. Next, an example will show exchangeability that cannot be modelled by terminal classes. Hereafter the exchangeable terminal groups extension is defined, and the additions to the algorithm are discussed.

Review of terminal classes

As has been described in Section 4.2, several connections to a component can have the same terminal class. For example, a MOS (Figure 4.2, Section 4.2.2) may have an SD terminal class, relating to both the source and the drain connection. Which of the two connections is actually the source or the drain is not specified; they are exchangeable. A second example is shown

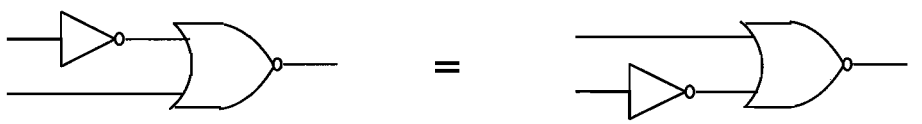


Figure 4.24: *Example of exchangeable inputs*

in Figure 4.3 (Section 4.2.3), describing a NOR having both inputs exchangeable. Both circuits in Figure 4.24 are equivalent by definition. This illustrates a terminal class that relates to multiple exchangeable terminals. The definition allows different sets of multiple exchangeable terminals for a component type.

Another exchangeability example

A component type whose connection exchangeability cannot be modelled by the terminal classes is a set-reset flip-flop (SRFF) for example. For a SRFF (see Figure 4.25), the R terminal and S terminal can be exchanged (switched) only when the Q terminal and Q-not terminal are switched at the same time. The ordered terminal sets {R, Q} and {S, Q-not} are exchangeable. The lower level implementation (Figure 4.26) of the SRFF shows why only pair-wise exchanging the SRFF connections is allowed: in this way the same graphs are found at the implementation level.

Definition 4.53 *Terminal-groups*

For a component type $t \in \tau$ with type terminal classes $TTC(t)$ according

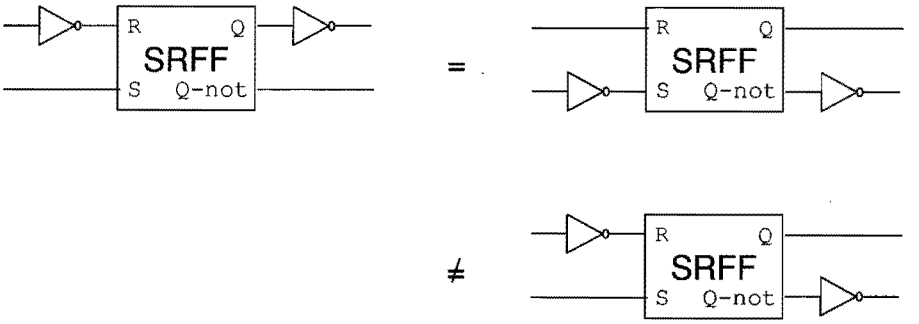


Figure 4.25: *Allowed (=) and forbidden (≠) exchangeability of a set-reset flip-flop*

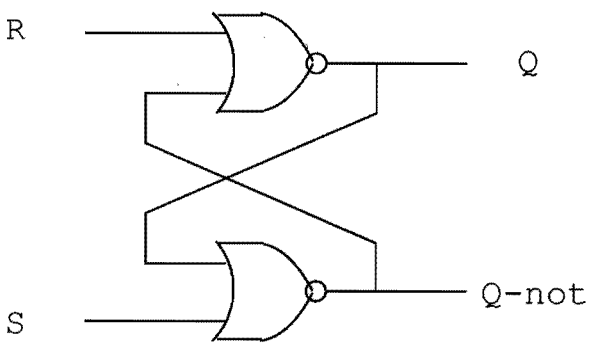


Figure 4.26: *Implementation of a set-reset flip-flop*

to Definition 4.15, a Terminal Group is an ordered subset of $TTC(t)$. A *terminal grouping* of component type t is an unordered set of equally sized terminal groups. A *terminal grouping set* of t is an unordered set of terminal groupings. For any two terminal groupings A and B of t, the union of all terminal classes in the terminal groups of terminal grouping A must be disjoint from the union of all terminal classes in the terminal groups of terminal grouping B. The total number of occurrences of any terminal class c in a terminal grouping set may not exceed the number of occurrences of c in $TTC(t)$. □

In this way, exchangeability described by A is independent from the exchangeability described by B. For the set-reset flip-flop (SRFF) example of Figure 4.27, the terminal groups set consists of only one terminal grouping, `((R S) (Q Q-not))`. Terminal group exchangeability can be generalized

```
(SRFF
   (Terminal-names        (R S Q Q-not)
    Terminal-classes      (R S Q Q-not)
    Terminal-groups-sets (
                          ((R S) (Q Q-not))
                         )
    Network               ((NOR N1 R Q-not Q)
                          (NOR N2 S Q Q-not))
    Restrictions          (()
                          ((R)(S)(Q)(Q-not))
                          ()
                          T)
   )
)
```

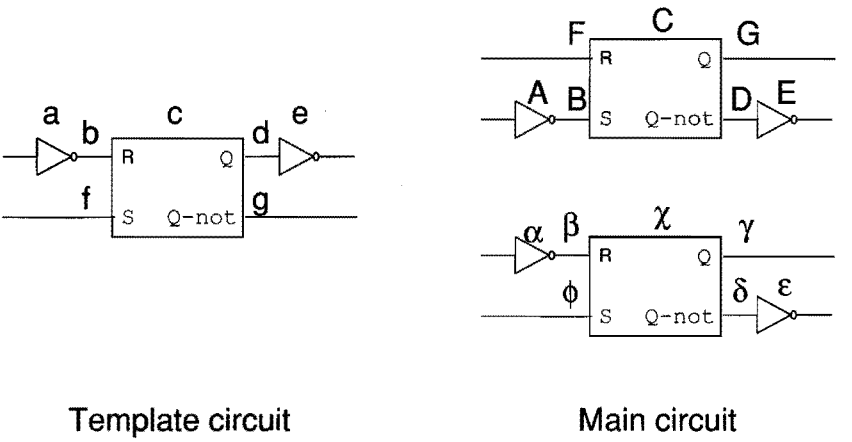
Figure 4.27: *The set-reset flip-flop component type, showing an example of the terminal groups sets property.*

to exchangeability of terminal groupings, and beyond.

The implementation

The extension to the basic recognition algorithm to handle terminal group exchangeability is sketched below. Basically, there are two possible directions to incorporate the extension. In one direction, the data representation should be changed in such a way that terminal group exchangeability is dealt with automatically, just like the current data representation solves the exchangeability of connections having equal terminal classes. In the second direction, the algorithm should model explicitly the terminal group exchangeability, by iterating over all allowable connection cases as described by the terminal grouping sets of all components.

A solution in the first direction, a convenient data representation, invariant to exchanging terminal groups, is still unknown at present. Another data representation would also demand modification of the current data representation of both the main circuit and the template circuit. It is very likely that the modified data representation will require more computer storage than used for the current data representation, and storage is a very important item for current applications.



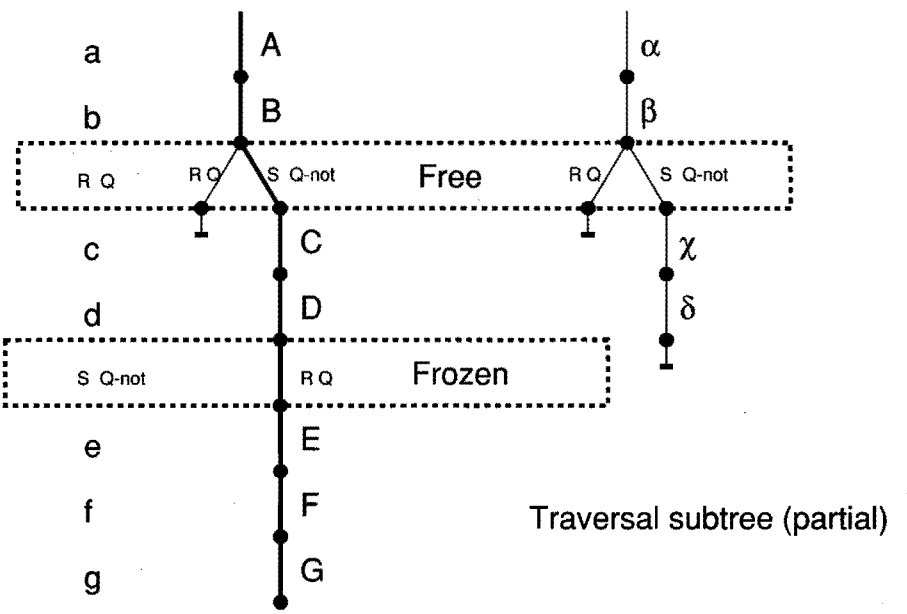


Figure 4.28: *Partial traversal sub-tree for exchangeable terminal groups. The demand functions associated with each level are related to the template vertices on the left side. Note the difference between the free (branching) terminal group* $\{R, Q\}$ *and the frozen (non-branching) terminal group* $\{S, Q - not\}$.

A solution in the second direction has a clear disadvantage: it deteriorates the current run time efficiency. The number of terminal class combinations may be very large. Suppose a component v of type t has j terminal groupings $\{TG_i(v) \mid i = 1, \ldots, j\}$. In the worst case, every component v adds the following multiplication factor to the run time:

$$\prod_{i=1}^{n} |TG_i(v)|! \tag{4.35}$$

Therefore, the exchangeability must be modelled as economically as possible, and the worst case must be omitted whenever possible. Figure 4.28 shows how the iteration is incorporated during depth-first search. When a component has exchangeable terminal groups, an extra level for each terminal group is added to the depth-first search algorithm when any of the connections in a terminal group is under consideration (see Figure 4.28). The figure shows that in this search order, {R, Q} is free, but {S, Q-not} is already frozen, since the other combination is already in use. This implies that {R, Q} creates extra branching, unlike {S, Q-not}.

The recognition algorithm consists mainly of the search list generation part and of the depth-first search part. The number of traversed edges during depth-first search strongly depends on the search list order. The search list order is determined by the procedure GENERATE_SEARCH_LIST, based on the branching factor functions U_j, $j = 1, \ldots, k$ (see Sections 4.3.7 and 4.3.9).

For the current extension, the formula given in Equation 4.48 that computes U_j is extended with an extra factor that computes the potential growth of the traversal size, introduced by the terminal group exchangeability. In this way, 'frozen' terminal groups have no effect on the old method of determining the search list order, but freely exchangeable terminal groups tend to be found at the end of the search list. This effect is stronger when the terminal groups are larger.

The depth-first search part depends strongly on effective selection of candidates at every level in the search tree. The algorithm to find candidates for a terminal group of a component uses the connectivity information of the related nets and component, both in the template and the main circuit, and resembles the FIND_CANDIDATE_SET of the primary-recognition algorithm.

4.6 Diagnosis feedback

In the event of a search for a particular template using the recognize algorithm not resulting in the number of matches that was expected, the reason why the recognition seems to have failed should be explained by the system as well. This section describes how valuable information can be retrieved after the execution of the recognition algorithm. Some examples of how this information is presented are shown as well.

Before discussing how diagnosis feedback is given when no matches are found, the fact that the situation when fewer (but more than zero) matches than expected are found can be translated into the problem when no matches are found.

The recognition algorithm consists of search list generation (see Figure 4.14 and Figure 4.17), a depth-first search procedure (see Figure 4.11 and Figure 4.19) and post-processing (see Chapter 4.4). The diagnosis feedback will be described in the following paragraphs for each step.

Diagnosis when finding fewer matches than expected
When fewer matches are found by the recognition algorithm than expected, the recognition monitor indirectly offers help. The recognition monitor only displays information in the case when zero matches are found, so a direct invocation doesn't work. If one removes the matching components first, before applying the same recognition again, the recognition monitor will then be able to display the diagnostics and suggestions as to why no more matches are found. The removal of matching components is a standard command in the implementation.

Diagnosis for the search list generation phase
The algorithm of search list generation (Figure 4.17) consists of an initialization (Figure 4.14) that computes the number of candidate main circuit vertices per template vertex, when considering the local demand function of the template vertex only, and a procedure to order the vertices into a search list. As indicated in the initialization algorithm (Figure 4.14, the line `if` $U_1 = 0$ `then exit("No matches")` indicates that when a template vertex has no candidate vertex satisfying the local demand function, the recognition algorithm can be aborted. Obviously, in this case the diagnosis of why no match was found should feed back the vertex under consideration to the user. As an example, the following could be the result of a failure.

Note that in the actual implementation the notions "node" and "element" are used for the notions "net" and "component" of this thesis.

```
> (rg-monitor 'why)

DIAGNOSTICS:
------------
- The used template is equal to the network of
  type-description: DFFL
- At least for the template element N1 no initial
  match could be found.

SUGGESTIONS:
------------
- Substitute all attribute-values of template
  element N1 by ''?''.
```

The information displayed above is the result of the "recognition monitor" call given in the first line, after a recognition resulted into zero matches. The information is split in a "diagnostics" part explaining why the recognition resulted in zero matches, and a "suggestions" part, containing suggestions how the template should be changed to come to a match.

Diagnosis for the depth-first search phase

During the depth-first search procedure (see Figure 4.11 and Figure 4.19), the search tree is traversed. At every preceeding level $j+1$ in the search tree, a partial isomorphism is extended with another (template vertex, main circuit vertex) pair, provided that the required connectivity demands and local demand are valid. When such an extension does not exist, the current computation of `FIND_CANDIDATE_SET` (see Equation 4.42) results in an empty set, implying that either Equation 4.41, called the tentative candidates, result in an empty set, or no element of the tentative candidates $\mathcal{F}_{j+1}$ satisfied the local demand $L_{v_{j+1}}$. When a partial isomorphism cannot be extended, backtracking occurs, and at a later stage, another partial isomorphism might be extendable beyond level $j+1$. Therefore, when no match is forthcoming, the partial isomorphism mapping relating to the maximum level reached, i.e., the one which comes closest to a possible match, is the most interesting one. The diagnosis information should display the largest partial isomorphism, and it should explain for the non-matchable vertex,

which of the conditions (see Equations 4.42, 4.41, 4.36 and 4.37) cannot be satisfied. As an example, the following is the result of a failure.

```
> (rg-monitor 'why)

DIAGNOSTICS:
------------
- The used template is equal to the network
  of type-description: DFFL
- The template could not be matched completely.
  The biggest match found was up to and including
  template object number 9 of a total of 19 objects
  (i.e. 47.4 percent is matched).
- The last template element that could be matched
  was: P2 ---> MP2
- The last template node that could not be matched
  was: N18
- To see the list of template objects type:
  (rg-monitor 'search_list)

SUGGESTIONS:
------------
- The connectivity of the template node N18 does
  not match with the connectivity in the network.
  Check connectivity using (rg-monitor 'candidates)
  or remove node from template and recognize again.
- In the restrictions node N18 is not specified as
  an external node. Specify it as an external node.
```

In addition, the partial isomorphism can also be retrieved, by using the recognition monitor again.

```
> (rg-monitor 'search_list)

 Nr.  Type     Template     Netlist  Remarks
-------------------------------------------------------------
   1  node     PHI1     ---> PHI1            fixed object
   2  node     PHI2     ---> PHI2            fixed object
   3  node     VDD      ---> VDD             fixed object
   4  node     GND      ---> GND             fixed object
   5  element  N7       ---> MN7
   6  element  N5       ---> MN5
   7  node     DIN      ---> MDIN
   8  node     N17      ---> MN17
   9  element  P2       ---> MP2
  10  node     N18      ---> matching failed
  11  element  N2       ---> -
  12  element  N1       ---> -
                etc.
```

Finally, a detailed explanation of why element N18 failed to be matched is given by enumerating all candidates that failed, i.e., all vertices connected to the partial template G_{10} (see Definition 4.21). G_9 is the sub-circuit of the template with respect to the partial search list s_9. The failed candidates are sorted according to the number of conditions that are fulfilled, in this way the most likely intended candidates are shown first. This information is presented by the recognition monitor as follows:

```
> (rg-monitor 'candidates)

  Objects between {} are not matched.
  Objects between <> are matched but to the wrong objects.
-------------------------------------------------------
prob. node  nr-con. class    elements
-------------------------------------------------------
      {N18}       2 MOS~GATE P2 {N2}
                  3 MOS~SD   N7 {P1} {N1}
=======================================================
0.90  {MN18}      2 MOS~GATE MP2 {MN2}
                  4 MOS~SD   MN7 <MN7> {MP1} {MN1}
```

The "probability" measures the number of conditions that are fulfilled. The first vertex description describes the template vertex, the main circuit

vertices are displayed below the "===" line. The example only shows one main circuit vertex. Every line shows the $Adj(vertex, class)$ value for a class, i.e., the number of connections, the class and the adjacent vertices. In this example, main circuit vertex MN18 has one SD connection too many (4 instead of 3), and a wrong connection to MN7. Apparently when removing the connection to MN7, this match seems to be correct.

Diagnosis for the post-processing phase
In the post-processing phase (see Section 4.4), zero matches can only be the result of partially overlapping matches. Although symmetry conditions can lead to a rejection of matches, at least one survives in that case, so it cannot be the cause of zero matches found. The following diagnosis and suggestions are given by the recognition monitor resulting from the partially overlapping matches occurring for the example of Figure 4.20, Section 4.4.1.

```
> (rg-monitor 'why)

DIAGNOSTICS:
------------
- The used template is equal to the network of
  type-description: INVERTER
- All 2 initial matches where rejected due to common elements.
  At least the following template elements where found
  to be common but not defined as such: T1

SUGGESTIONS:
------------
- Add elements: T1
  to the common elements in the restrictions.
```

4.7 Results

In this section, some results of the recognition algorithm will be shown. The function implementing the algorithm is called RECOGNIZE, and is part of the Vera environment. No standard benchmark set exists for sub-circuit recognition, so we must select appropriate recognition examples ourselves. Many designs have been verified by our hierarchy reconstruction method, so we could have taken the largest (over a million transistors), show the set of run times for every recognition during the hierarchy reconstruction

process (about 2 hours in total on a HP-9000/750) and argue that the recognition algorithm is fast enough. However, such an example would not be representative, since designs having large numbers of transistors are usually dominated by a large Random Access Memory part. As will be shown in this section, RAM-cells are easily recognized, so large designs give an optimistic view for the average case. Furthermore, a non-representative example would reveal little about the run times of different parts of the recognition algorithm. Therefore, a moderate, representative, industrial design has been chosen, that contains many different modules.

The selected design, called the TDA-1307 [Deloor92], is composed of 130 000 transistors (see Figure E.2), and is introduced more elaborately in Chapter 6. The hierarchy reconstruction verification process has been applied to this design, and the 134 recognition calls in the non-parameterized hierarchy reconstruction process will be analyzed paying special attention to the top 36 in CPU time. In the hierarchy reconstruction process, many other recognition calls are applied as well, being part of a parameterized hierarchy reconstruction process. Although large in number, these calls are not considered here, since they depend strongly on the context of usage, and they are usually so fast that accurate timing information can hardly be obtained.

The recognition algorithm consists of two parts, the primary recognition algorithm (Section 4.3.11) and post-processing (Section 4.4). Both algorithms are extended according to Section 4.5. As shown in Section 4.3.11, Figure 4.19, the primary recognition algorithm consists of search list generation and a depth-first search procedure. The search list generation procedure itself consists of an initialization (the algorithm of Figure 4.14) and the actual search list generation algorithm (Figure 4.17). The initialization computes mainly $\mathcal{L}(v)$, i.e., the set of main circuit candidates for each template vertex v, that satisfy the local connectivity demand function L_v. So for the run time analysis, the following parts of the recognition algorithm will be distinguished:

- Initialization of search list generation (ISG),
- Actual search list generation (ASG),
- Depth-first search (DFS),
- Post-processing (Post).

Template	|V|	|V|	Match count	Total (sec.)	ISG (sec.)	ASG (sec.)	DFS (sec.)	Post (sec.)
Noiseshaper	80014	1103	2	120.75	20.02	2.07	96.78	1.72
Nor3	56182	14	0	86.81	0.92	0.01	85.87	0.00
Nand4	78744	18	0	76.98	1.63	0.01	75.32	0.01
Dobm_ori	83564	1085	1	63.29	27.96	2.33	32.25	0.60
Axu_ori	84543	690	1	30.73	21.02	0.56	7.28	1.78
Pint_ori	82356	472	1	24.95	21.98	1.01	1.78	0.09
Iisinput_ori	77736	323	1	14.48	13.95	0.42	0.04	0.02
Myespa_ctrl	83914	375	1	13.80	13.13	0.33	0.22	0.05
Decoder2	47644	15	43	13.47	1.35	0.01	12.10	0.01
Ram-cell	173132	13	6300	13.35	3.98	0.00	7.91	1.43
Decoder3	47386	18	35	12.77	1.41	0.01	11.33	0.01
Decoder4	47106	21	50	11.14	1.47	0.01	9.64	0.01
Cs_29_sum	12729	27	255	10.79	0.32	0.01	10.36	0.08
Alignment	77868	125	1	10.06	9.61	0.39	0.02	0.01
Tcb_ori	84769	240	1	9.10	8.66	0.16	0.20	0.03
C3_nand	103351	10	673	9.02	1.77	0.01	7.16	0.07
Nand2	79861	10	256	7.46	1.42	0.00	6.00	0.03
C3_inv	119333	6	15472	6.39	1.59	0.00	3.66	1.11
Clockdiv	78699	101	1	6.09	6.02	0.04	0.00	0.01
Synchron	77812	90	1	6.06	5.96	0.06	0.02	0.00
Decoder1	46606	12	16	5.63	1.20	0.00	3.56	0.86
Invertor	92331	6	10936	4.71	1.20	0.00	2.75	0.74
Nand3	78828	14	12	4.65	1.51	0.01	3.10	0.01
Glbclock	103798	93	1	4.49	4.41	0.04	0.01	0.01
Cuinc	58423	73	1	4.34	4.29	0.03	0.01	0.00
Cs_24_mux	12905	14	20	4.16	0.29	0.01	3.86	0.00
Clockphins	103861	69	1	4.10	4.03	0.03	0.00	0.00
Acuslice	56624	53	16	3.92	3.77	0.02	0.06	0.06
C3_hlatch	91097	22	238	3.55	3.24	0.01	0.24	0.04
Dtn12tac	103711	36	12	3.29	3.24	0.01	0.02	0.00
C3_mux	97376	13	433	3.10	2.22	0.00	0.82	0.05
C3_latch	88003	20	294	3.01	2.66	0.00	0.27	0.07
Nand2_cinp	100659	8	3	2.86	1.55	0.00	1.29	0.00
Nand5	78744	22	6	2.83	1.74	0.01	1.05	0.01
Muxreg-top	39587	44	1	2.83	2.76	0.03	0.02	0.00
Reg-cell	52524	18	964	2.58	1.38	0.01	0.27	0.90

Table 4.1: *Performance of recognition algorithm.*

Apart from the cpu run-times associated with the algorithmic parts, Table 4.1 enumerates per recognition call the component type, the number of main circuit vertices ($|\mathbf{V}|$), the number of template circuit vertices ($|V|$), the number of found matches (Match count) and the total recognition time. Since the run time samples have been taken during a hierarchy reconstruction process, the number of main circuit vertices decreases gradually. Table 4.1 is ordered according to decreasing run-times, and only the 36 worst cases out of 134 are shown. The total recognition time for all component types is 707 seconds. The hierarchy reconstruction job ran on a HP-9000/735. The following trends can be derived:

1. Most recognition calls (most of them are not enumerated in the table) take less than a second, few (<3%) take several minutes.

2. 5% of the recognition calls take 50% of the run time.

3. On average, the search list generation (ISG + ASG) consumes more cpu-time than the depth-first search tree traversal, although for the worst cases, the depth-first search is dominant. These worst cases are found mainly in the top 15. On the average it seems that a reasonable balance between ordering heuristic and tree traversal computations has been obtained.

4. The search list generation initialization (ISG) dominates the total search list generation process.

5. The number of matches has little correlation with the run times e.g. 26000 inverters (C3_inv and Inverter) are found in 11 seconds, 6300 memory-cells are found in 13 seconds, whereas to establish that Nor3 and Nand4 have no matches takes 154 seconds.

6. Large templates ($|V|$) correlate with longer run times.

7. Post-processing time is never substantial.

Although the algorithm performs very well on the average, the Nand4 and Nor3 component types illustrate that the heuristics of the algorithm show a large variation in effectiveness. When comparing the run times with others [Bolse89], [Spick88], [Nebel87], [Papas88], [Hirsch88], the algorithm is at least an order of magnitude better.

4.8 Conclusions

A recognition algorithm to find all occurrences of a template circuit in a main circuit has been described in detail.

The algorithm is based on a depth-first search backtracking. Most of the benefits of the algorithm result from a careful ordering of the search list. The algorithm to order the search list has been described in detail. It is based on estimates of the branching factor function. In this way the size of the traversed part of every search tree during the actual depth-first search process is predicted, before actually performing the traversal. This allows the selection of a search tree that needs little traversal. The computations of the branching factor function values are performed efficiently by grouping the computations first, before actually computing the values. In this way only one computation is needed per group.

The post-processing step, after the backtracking process, handles matches that overlap depending on the user requirements. Finally, some important extensions of the algorithm are described that enhance the flexibility. A self-explanatory help facility called the recognize monitor provides clear diagnostic information when less is recognized than expected.

The worst case run time efficiency of the algorithm has been shown for recognitions that occurred during a typical hierarchy reconstruction job. The results indicate that a reasonable balance between ordering heuristic and tree traversal computations has been obtained. In general, the performance of the algorithm is very good for recognizing electrical sub-circuits present in IC-designs.

Chapter 5

The hierarchy reconstruction implementation

In Chapter 3, the required tools and libraries have been derived that are needed to make the hierarchy reconstruction method operational, as summarized in the operational model, Figure 3.3. The main tool, the sub-circuit recognizer, has been described separately in Chapter 4. This chapter describes the implementation of the hierarchy reconstruction method as it has been embedded in the Vera environment. Since the other tools are not as spectacular as the recognition tool, their description will be less elaborate. The sub-circuit recognizer will also be recapitulated, to summarize it's functionality and to show the coherence with the other tools and the Vera environment.

Vera [Koste89], [Koste88], [Deloor90] is an acronym for VERification Assistant, a rule-based environment for the analysis and manipulation of electrical circuit designs. The schematic representation of Vera's architecture is given in Figure 5.1. As will be shown in this chapter, Figure 5.1 is the implementation view of the operational model (Figure 3.3).

The inputs for Vera consist of a network description, component type descriptions and a rule base, all supervised by the user.

The *Network Description* (see Figure 5.1) describes the main circuit which will be verified or modified, see Definition 4.18, Section 4.2.2. The description is component-oriented and based on the component types de-

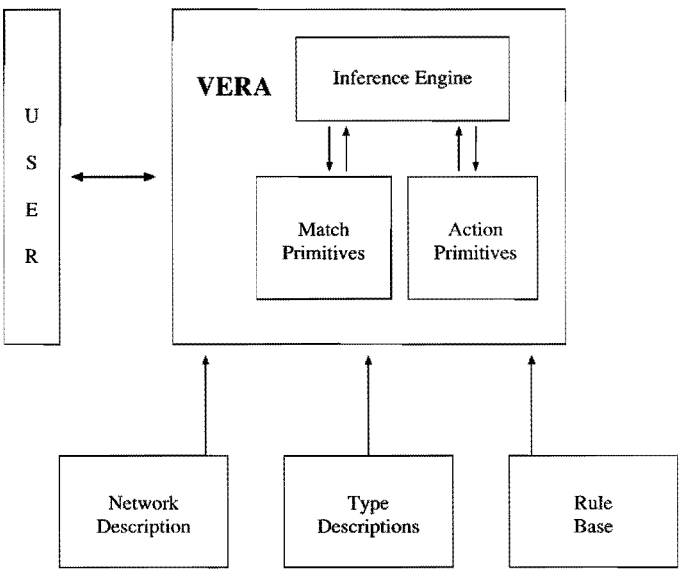


Figure 5.1: *Vera's architecture*

fined in the type descriptions.

The *Type Descriptions* (see figure 5.1) contain the general characteristics of component types, including the template circuit. Examples are given in Figures 4.2 (Section 4.2.2) and 4.3 (Section 4.2.3). In general, a type description enumerates a number of properties and their values, see Sections 4.2.2 and 4.2.3. The *Rule Base* contains definitions of rules. A rule defines a situation (IF part) and the action to be taken when such a situation occurs (THEN part). Examples of rules are given in the next section. For our application, the rules describe the control of the reconstruction process. Rules are defined hierarchically in terms of other (lower level) rules and actions. At the lowest level, rules are based on a flexible and easily extendible set of primitives such as: a sub-circuit recognizer, an abstracter, etc.

Internally, Vera consists of an inference engine and a separate set of tools called primitives (see Figure 5.1). The inference engine handles the interaction between the rule base, the primitives and the user. With respect to the primitives, a distinction is made between match primitives, which are used in the IF-part of a rule, and action primitives which are used in the THEN-part. Match primitives search for the presence of facts or patterns in a circuit, while primitive actions modify, add or delete circuit compo-

nents/nets or generate messages. Vera is implemented in Common_LISP. More details on Vera can be found in [Koste91], [Koste92.2], [Koste92.3].

5.1 The RECOGNIZE primitive

The sub-circuit recognizer primitive is called RECOGNIZE. In addition to chapter 4, where the tool is described formally and in-depth, this section briefly reviews the functionality. In this way, the embedding of the algorithm in the Vera environment will become clear.

In general, the recognize primitive finds all occurrences of a specified template in the current main circuit and returns them to the inference engine. The template is specified by two arguments, the circuit and the restrictions.

A **circuit** describes most of the template. It's components and nets indicate how a matching cluster of components and nets must be connected, and the values per attribute that a component match should have. For a component type, the `network`-entry defines the circuit, see Figure 4.3.

A set of four **restrictions** prescribe the remaining details of the template. See also Figure 4.3.

1. The **fixed-nets** is a list of those template nets that must match the identically named actual nets in the circuit, thus reducing the set of acceptable occurrences. For more details, see Section 4.5.1.

2. **External-net-groups**. By default, nets are *internal*, meaning that a matching net must have exactly the same connectivity as its counterpart in the template. When a net has more actual connections than is indicated, it is *external*. This is indicated when the net is mentioned in any *external-net-group*. Furthermore, the user can indicate that different external-nets may be short-circuited by grouping possibly connected nets together. For example, in an `inverter`, the `in` net and `gnd` may be connected in some cases. For more details, see Section 4.5.2.

3. **Common-components**. By default, when occurring matches partly overlap, they are all rejected. For instance, if the user defines a memory cell as a flipflop plus a sense-amplifier, the sense-amplifier may be the same for a whole column of different flipflops and thus all

memory cells are rejected. When however the overlapping components (like the sense-amplifier) are mentioned in the list of common-components, no rejection occurs. In this way, a casual flipflop without sense-amplifier, used for its driving capability is not recognized as a memory-cell. Common-components allow the facility of adding some external environment to the circuit template thus making it more specific. For more details, see Section 4.4.

4. **Symmetry** may be given two main values: `T` (true) and `nil` (false). Symmetry occurs when a template maps several times onto exactly the same components but in a different order, being a permutation of the same match. This phenomenon is called automorphism. For example a circuit of two parallel resistors can be interchanged and still reflect the same circuit. When *symmetry* is true, only one match is returned of the completely overlapping set, otherwise all permutations are returned. For more details, see Section 4.4.

5.2 Hierarchy reconstruction for various hierarchy categories

In this section, the implementation of hierarchy reconstruction will be described for all hierarchy categories enumerated in Section 3.2.

Category 0: non-parameterized modules
For the architecture reconstruction of non-parameterized modules, the requirements mentioned in Chapter 3 are met as follows. The non-parameterized module library is stored in type descriptions. The pattern matching is performed by the *recognize* primitive described above. The next step is to replace the recognized pattern by a higher level component. This is handled by the action primitive *Abstract*. The following rule called FIND-AND-ABSTRACT performs the reconstruction.

```
RULE FIND-AND-ABSTRACT (abelt) (type)
 IF RECOGNIZE (circuit)                    {1}
              (NETWORK(type),
               RESTRICTIONS(type))
    FIND-NAME (abelt)(type)                {2}
 THEN ABSTRACT (circuit, abelt)(type)      {3}
END
```

When `find-and-abstract` is activated by the user as follows:

```
ACTIVATE (FIND-AND-ABSTRACT ('inverter)),
```

this rule will replace all transistors in the network description forming an inverter network, by inverter components (see Figure 3.1). The `activate` gives `type` the actual value `inverter`. The `abelt` is not used by the activate call. When the rule is used hierarchically as a match call, the `(abelt)` in the first line acts as an external variable, with which information can be passed to and from other rules or primitives. In the current example it stores the names of the abstracted components.
In the IF part of the rule, the network of the inverter is used as a template by the match primitive `recognize` {1} which will find all matching sub-circuits. For every match the primitive `find-name` {2} will generate a name for the new inverter component. The action part is applied to the resulting set of [sub-circuit, abelt] pairs, so the `abstract` primitive {3} will replace all sub-circuits by corresponding inverter components.
Hereafter the same rule can be activated again with a different `type` such as a `memorycell`, whose `network` is described in terms of inverters, etc., thus raising the circuit level gradually up to a description in terms of modules. In this way the complexity of the circuit can be reduced enormously.

Category 1: singly-parameterized modules
For Category 1 modules, a structure repetition detector and parameterized template generators are required. The Vera primitive matches CHAIN and FORK detect respectively serial and parallel repetition in connectivity. Also, two related abstract action primitives have been created called ABSTRACT-CHAIN and ABSTRACT-FORK which generate the appropriate component type description, and abstract the repetitive structures. The rule `fork-and-abstract` can abstract parallel structures:

```
RULE FORK-AND-ABSTRACT (abelt, nr)(typ, con)
 IF FORK (eset, nr) (typ, con)                  {1}
    FIND-NAME (abelt) ( concat (typ, nr, '-)) {2}
 THEN ABSTRACT-FORK (abelt, eset)( typ, con)   {3}
END
```

The rule `fork-and-abstract` strongly resembles `find-and-abstract`. When the rule is activated as follows:

```
ACTIVATE (FORK-AND-ABSTRACT ('memorycell '(r/w)))
```

the result is that memorycell (= typ) components are considered that have the terminal r/w (= con) connected in parallel. In addition, the abelt and nr are only important in the case of hierarchical rules, where they are used to pass information between rules and primitives. The memory cell is a component that may have been abstracted by find-and-abstract in previous steps. In the IF-part, fork {1} will find sets of memorycell components that are all connected to the same net via a r/w terminal. The sets are accumulated in eset, and the corresponding repetition parameter (the word length) in nr. Per set, find-name {2} will generate a name, e.g., memorycell3-1. In the THEN-part, abstract-fork {3} will replace every set by a new higher level component of type FORKMEMORYCELL-<nr> generated by abstract-fork itself, derived from nr, typ and con. It combines type description generation and non-parameterized module abstraction. For the example of Figure 3.2, the result will be a component of type FORKMEMORYCELL-3 called MEMORYCELL3-1.

Category 2: multiple parameterized modules

Multiple parameterized hierarchy reconstruction is performed by repeating singly parameterized hierarchy reconstruction. Although no additional requirements are formulated, the rules that are used are more complex. A typical example is the hierarchy reconstruction of a memory core (see Figure 3.2). It can be reconstructed by the hierarchical rule find-mem-core which calls the fork-and-abstract rule twice:

```
RULE FIND-MEM-CORE (mc, wl, wn) ()
 IF FORK-AND-ABSTRACT (regw, wl)          {1}
                      ('memorycell,
                       '(r/w))
    FORK-AND-ABSTRACT (mc, wn)            {2}
      ( concat ('forkmemorycell-, wl),
        r-l-a-i ('(in out), wl))
 THEN MESSAGE (memory core mc             {3}
               is abstracted)
END
```

When the rule is activated as follows:

```
ACTIVATE (FIND-MEM-CORE)
```

the memory cells are grouped and merged into register words which are themselves grouped and merged into memory cores. Register words regw

consist of wl memory-cells connected in parallel via the r/w terminal, as explained in the previous section. Memory cores mc are register words connected in parallel via all in<k> and out<k> terminals ($k = 0, \ldots, wl-1$). The iteration is performed by r-l-a-i (Repeat-list-and-increment, more details are found in Section 5.3). The word length parameter wl is recovered in the first fork-and-abstract, and the number-of-words wn is recovered in the second step.

A complete memory (a RAM) also has address decoders and buffers. They have not yet been abstracted. Category 1 modules often consist also of a repetitive core plus some extra circuitry. How these parts are abstracted in a following reconstruction step, is explained in the next section.

The reconstruction of modules consisting of repetitive structures and extra sub-blocks.

The hierarchy reconstruction of modules consisting of repetitive structures and extra sub-blocks is explained by the reconstruction of a bit slice processor (Figure 5.2).

A (bit)slice is a component that is composed of full-adders, shifters, registers, multiplexors, etc. These components may have been abstracted by find-and-abstract beforehand. The extra sub-block is control. Before describing the automatic reconstruction, a simpler, but manual, method is first explained. Starting from situation (a), the command

```
ACTIVATE (FORK-AND-ABSTRACT ('slice '(c0 c1)))
```

performs the operations to get to situation (b). The slice components connected in parallel via c0 and c1 terminals are reconstructed to a fork-slice-4 block, similar to the reconstruction of a memory word. To get from (b) to (c) by using the rule find-and-abstract is not possible, since the rule only operates for simple non-parameterized type descriptions. However, when parameter values are known, parameterized type descriptions can generate a simple non-parameterized instance. During the step from (a) to (b), the parameter value or repetition number is detected, so find-and-abstract can be used after all. The user can start the operation interactively by the command

```
ACTIVATE
  (FIND-AND-ABSTRACT
       ( LOAD-TD ( PROCESSOR (4) ))),
```

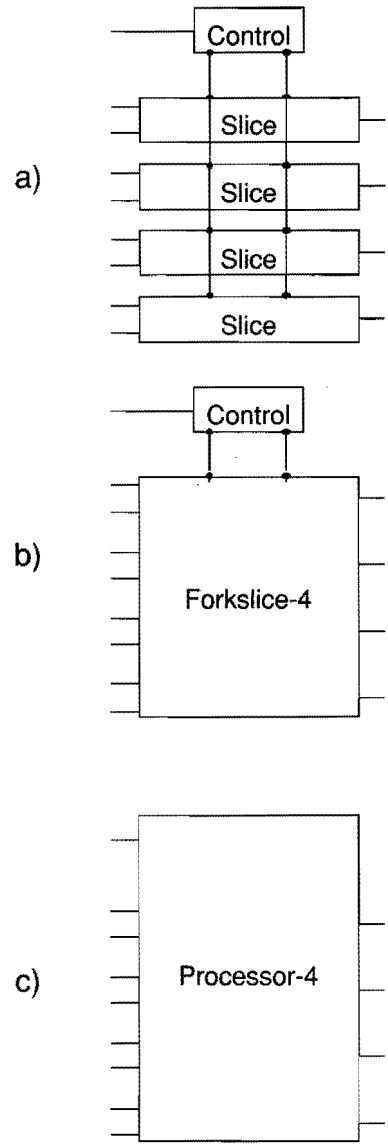


Figure 5.2: *Hierarchy reconstruction of bit slice processor a) after Category 0 abstraction b) after* `fork-and-abstract` *c) after* `find-and-abstract`.

in which `load-td` adds the instance `processor-4`, generated by the parameterized type description `processor (n)`, to the non-parameterized type descriptions. Hereafter, `Find-and-abstract` performs the reconstruction operation.

The method described above requires manual interventions by the user during the reconstruction process. It can be completely automated by the construction of a hierarchical rule `f-and-a-processor` which controls all reconstruction steps, from (a) to (c). It is defined as follows:

```
RULE F-AND-A-PROCESSOR (proc-e, nr)
 IF FORK-AND-ABSTRACT (slice, nr)  {1}
                      ('slice,
                       '(c0 c1))
    FIND-AND-ABSTRACT  (proc-e)    {2}
      (LOAD-TD ( PROCESSOR (nr)))
 THEN MESSAGE (processor proc-e    {3}
               is abstracted)
END
```

The major difference with the manual method is that the repetition numbers `nr` detected by `fork-and-abstract` are automatically passed to the `find-and-abstract` by Vera's inference engine. The activation

```
ACTIVATE (F-AND-A-PROCESSOR)
```

reconstructs any number of processors of any parameter value, and `message` signals to the user which processors are abstracted. The parameterized type description `processor (n)` is explained in Section 5.3.

Category 3: Parametrized modules having a variable functionality

Category 3 contains the hardest modules for hierarchy reconstruction, because their structure is difficult to predict. In general it is no longer a one-dimensional repetition of a fixed structure. For PLA and ROM modules, the various decoders can be reconstructed by using the Category 2 reconstruction method. The number of addresses, inputs and outputs is derived during this stage. The table extractor is a primitive which derives the contents of the module core. Internally it uses the recognize primitive to notice the presence or absence of transistors, and fills the table accordingly. When the module is reconstructed, the functionality in the table is

automatically compared to the intended functionality. This method has been elaborated in [Kuppe89.1] and [Kuppe89.2].

For Category 3 modules that are not described by a table, a different and dedicated solution might be needed.

5.3 An example of a parameterized type description

In the previous section, parameterized type descriptions were used for hierarchy reconstruction of macro-cells. In this section an example of a parameterized type description implementation is given. Parameterized type descriptions are regarded as non-parameterized type description generators. From a parameterized type description, a simple, non-parameterized type description can be instantiated. In Vera, a parameterized type description is a function which looks like a simple type description template. It is written in Common_LISP format. Often occurring constructs are supported by Vera functions.

The parameterized processor of Section 5.2 is defined as follows:

```
Function PROCESSOR (nr)
`(,(concat `processor nr)
    (terminal-classes (cin ,@(r-l-a-i `(x y out)
                                      nr))
     network ((control element1 cin c0 c1)
              (,(concat `forkslice- nr) f c0 c1
                        ,@(r-l-a-i `(x y out)
                                   nr)))
     restrictions (()
                   !
                   ()
                   ())
    )
 )
```

Some remarks on details of the generator definition:
The parameterized processor has a variable number of terminals and consists of a control element and a fork-slice-<n>. The backquote "`" indicates that everything is meant literally, except for the expressions starting with a comma ",", which must be evaluated. `Concat (s)` is a function for string

concatenation. The at operator "@" removes the outer set of parenthesis in the outcome of an expression. The function `r-l-a-i (l n)` is short for *Repeat-List-And-Increment*, which returns a list containing n copies of l, concatenated with the iterative number. For example: `r-l-a-i ('(in out) 2)` returns `(in0 out0 in1 out1)`. For Figure 5.2 the following non-parameterized type will be generated:

```
(processor4
  (terminal-classes (cin x0 y0 out0
                         x1 y1 out1
                         x2 y2 out2
                         x3 y3 out3)
   network ((control cin c0 c1)
            (forkslice-4 f c0 c1
                         x0 y0 out0
                         x1 y1 out1
                         x2 y2 out2
                         x3 y3 out3))
   restrictions (()
                 !
                 ()
                 ())
  )
)
```

Parameterized type descriptions are straightforward and easy to write, for they only contain connectivity information.

5.4 Reconstruction order and hidden hierarchy

After having described different reconstruction methods for various hierarchy constructs in the last section, this section discusses the problem of reconstruction order and hierarchy in the type descriptions. Since any reconstruction rule only searches occurrences of one (parameterized) template at a time, a given set of templates should be ordered, to apply the reconstruct rules one-by-one. One speaks of **hidden hierarchy** when a template of a component type contains structures that are a template of another component type.

An example

The following example will show that the reconstruction order, the type descriptions and hidden hierarchy are strongly related.

Suppose one starts with the following set of type descriptions.

```
(INVERTER
  (Terminal-names   (in out)
   Terminal-classes (in out)
   Network   ((mos t1 out in vdd vdd ? ? ptype)
              (mos t2 out in gnd gnd ? ? ntype))
   Global-nets  (vdd gnd)
   Restrictions ((vdd gnd) ((in)(out)(vdd)(gnd)) () ())))
(MEMORY-CELL
  (Terminal-names   (b binv r/w)
   Terminal-classes (b b    r/w)     ; b and binv permutable
   Network ((mos t1 b  r/w b1   gnd ? ? ntype)  ; pass trans.
            (mos t2 b2 b1  vdd  vdd ? ? ptype)  ; inverter p
            (mos t3 b2 b1  gnd  gnd ? ? ntype)  ; inverter n
            (mos t4 b1 b2  vdd  vdd ? ? ptype)  ; inverter p
            (mos t5 b1 b2  gnd  gnd ? ? ntype)  ; inverter n
            (mos t6 b2 r/w binv gnd ? ? ntype)) ; pass trans.
   Global-nets (vdd gnd)
   Restrictions ((vdd gnd) ((b)(binv)(r/w)(vdd)(gnd)) () T)))
```

In this case, both the inverter and the memory-cell type have the same hierarchical level, since their template consists of transistors only. Since two inverters (t2,t3 and t4,t5) can be recognized in the template of the memory-cell, this set of type descriptions contains hidden hierarchy. There are two ways to proceed.

The first solution is to reconstruct the memory-cells in a main circuit before reconstructing the inverters, i.e., by using the above type descriptions, and the following

```
reconstruction order (memory-cell , inverter).
```

A second solution is to remove the hidden hierarchy, i.e., change the type description into

```
(INVERTER
  (Terminal-names   (in out)
   Terminal-classes (in out)
```

```
    Network  ((mos t1 out in vdd vdd ? ? ptype)
              (mos t2 out in gnd gnd ? ? ntype))
    Global-nets  (vdd gnd)
    Restrictions ((vdd gnd) ((in)(out)(vdd)(gnd)) () ()))))
(MEMORY-CELL
  (Terminal-names    (b binv r/w)
   Terminal-classes (b b    r/w)      ; b and binv permutable
   Network ((mos t1 b  r/w b1   gnd ? ? ntype)  ; pass trans.
            (inverter i1 b2 b1)
            (inverter i2 b1 b2)
            (mos t6 b2 r/w binv gnd ? ? ntype)) ; pass trans.
   Global-nets (gnd)
   Restrictions ((gnd) ((b)(binv)(r/w)(gnd)) () T)))
```

and use the

```
reconstruction order (inverter , memory-cell).
```

This example shows that for a set of hierarchy reconstructions, the levels used in the templates and the reconstruction order are strongly related, and should be chosen with care.

Ambiguity

As a first step to get an reconstruction order, a component type should be matched only after its children in the hierarchy (i.e., the component types in its network entry) have been reconstructed. This leads to a partial order. Secondly, if more than one type has all its children reconstructed, the types with the highest component number should be selected first. In this way, smaller templates are not accidently recognized in larger templates.

However, there is no guarantee, even if we use the above guidelines, that the intended hierarchy of a correct main circuit is reconstructed. The reason is that the matching process is ambiguous because the result may be different when different reconstruction orders are applied. This means that when a verification by hierarchy reconstruction is not successful, the main circuit may or may not contain an error. The following ambiguity problems might be encountered.

1. *Hidden hierarchy.* In this case isomorphic structures might have different hierarchical levels. For example, a type might contain an inverter as a component, and two transistors forming a component next

to it. When this type is searched for, after the reconstruction of an inverter, the type is not reconstructible, since reconstruction of the two inverters leaves a structure non-isomorphic to this type. Alternatively, when the type is ordered before the reconstruction of inverters, no inverters are present, also preventing the recognition of the structure.
As another example, different types might have isomorphic templates, i.e., have isomorphic network and restriction entries.

2. *Partially overlapping matches.* A structure C might be recognizable at the borders of a structure A and B. When C is reconstructed first, neither A nor B are reconstructible. When either A or B are reconstructed first, C is not recognizable.

The first ambiguity problem is solvable, by transforming the type templates and reconstruction order according to order-irreducibility [Pelz91], [Pelz94]. For a given hierarchy of n templates H, described by a set of type descriptions, and a reconstruction order $O = (t_1, \ldots, t_n)$, order-irreducibility means that every template $t_i \in O$ is not isomorphic to any of the templates $t_{i+1}, \ldots, t_n$. In other words, there should not be a hidden match in the remainder of the reconstruction order. In the inverter/memory-cell example shown above, both indicated solutions are order-irreducible. Similarly, the examples for the first ambiguity problem can be solved by changing the types, or by removing isomorphic types completely.

In Vera, an analysis tool called the TD-MONITOR [Koste92.1] is present, capable of generating an irreducible order for the current type templates, or of interactively helping to remove hidden hierarchy from the type descriptions. In addition, a given reconstruction order can be checked for irreducibility. A complication, however, is the possibility of having isomorphic templates, where one might be more rigidly defined than the other. For example, the template networks

```
((mos p1 out1 in1 vdd vdd ? ? pmos)
 (mos n1 gnd in1 out1 gnd ? ? pmos))
```

and

```
((mos p1 out1 in1 vdd vdd ? 1e-6 pmos)
 (mos n1 gnd in1 out1 gnd ? 1e-6 pmos))
```

are isomorphic, but the first one matches to more structures (including those with different length values) than the second. In a reconstruction

order, the more specific structures are obviously put before the more general structures. The same strategy applies to templates having equal networks, but different restrictions.

The implementation of the TD-MONITOR is based on the RECOGNIZE primitive, and it uses signaturing to prevent searching for every template in every other template. The run times of this tool are very good, even with type-description libraries of hundreds of types.

The second ambiguity problem however, the case of partially overlapping matches, cannot be solved easily. Although one could think of an exhaustive algorithm, searching for any template in any imaginable combination of templates, this algorithm would have very large run times. Fortunately, only several combinations of components makes a reasonable circuit, so the problem seldomly arises. The experience with applying the hierarchy reconstruction method so far has confirmed the rare occurrence of and the limited harm caused by this problem.

5.5 Layout positions and very large designs

This section shows a simple approach to handle layout positions during hierarchy reconstruction. The starting point for this approach is the inclusion of layout positions per component in a network description. This information is also the starting point for a simple but powerful extension of the method to reconstruct hierarchy by using parallel hierarchy reconstruction processes, each operating on small sub-designs.

Layout positions in a network description

Layout extraction generates a transistor network from the layout. The layout extractor computes various attributes per MOS-transistor. For example, for a component type definition

```
(MOS
        (   Terminal-classes   (sd gate sd bulk)
            Attribute-names    (width length model)
        )
)
```

the width, length and model of every MOS is extracted from layout. By adding an extra attribute, named `position`, to the MOS type

```
(MOS
        (   Terminal-classes    (sd gate sd bulk)
            Attribute-names     (width length model position)
        )
),
```

and all other component types, any information can be stored as an attribute-value for the position. The position of each component can be indicated in the form of a pair representing the x and y position of a point, or a 4-tuple representing the x1, y1, x2, y2 position of a rectangle. Obviously, this form is primitive, since L-shapes etc. cannot be modeled correctly. It does however serve as a first approach to gain experience when coupling layout with network information. For the transistor network, the position is computed by the layout extractor.

Layout positions and hierarchy reconstruction

From a global point of view, hierarchy reconstruction consists of matching, and abstraction. In the Vera environment, abstraction is performed by the primitives `abstract`, `abstract-fork`, `abstract-chain`, etc. Every abstraction replaces a set of matched components by a single component of a higher level. By computing the bounding box of each set of matched components, and assigning this rectangle as position to the associated new component, the layout positions can be taken into account as well. Figure

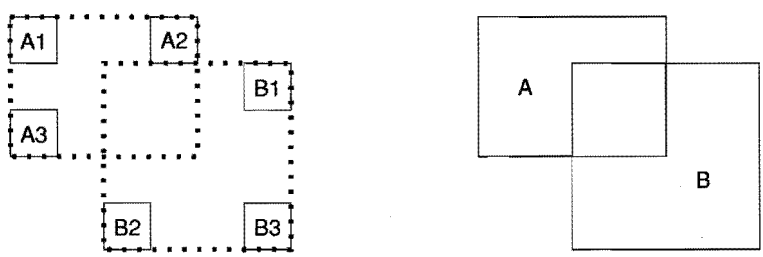


Figure 5.3: *Abstraction including position information based on rectangular shapes*

5.3 shows a small example of how `A1, A2, A3` and `B1, B2, B3` are replaced by `A` and `B`. The figure also indicates the main disadvantage caused by the rectangular shapes. When lower level components do not overlap, the abstracted components might overlap. Despite its simplicity, the approach

still indicates the layout position the abstracted components stem from rather well.

Layout positions and large designs

In the current implementation on an HP 9000, a transistor network of 250 000 transistors can be stored in main memory. For larger designs the execution speed drops dramatically because of a drastic increase of the number of page faults, eventually leading to thrashing, i.e., the situation when every memory access leads to a page fault.

One solution could be to cut the network into chunks of 200 000 transistors, before starting hierarchy reconstruction. In that case the following problems might arise during reconstruction into a chunk:

1. A match of a template might not be found because it is split up. It is even possible that no matches of any template can be found.

2. A template might lead to an invalid match, because the part that would prevent the match from recognition in the total network is part of a different chunk.

By splitting the network based on position information, the first problem can be solved. Since a designer knows the intended positions of the original high level network (Figure 1.3), one can cut along the borders of the high level cells, where low level layout cells are not split up. This way of splitting a design also solves the second problem partly. However, the matching net of an internal net should have an equal number of connections per class. If one knows that a net appears in one chunk only, this is guaranteed. Therefore, during the network splitting, nets appearing in different network chunks are stored in a separate file. Figure 5.4 shows the method for a two chunk case. First the design is split into two chunks and nets present in both chunks are administered. Next, two separate hierarchy reconstruction processes are run, that both use the information present in the administration. This results in two reconstructed chunks. The merge operation combines the separate parts into one network. The advantage of this method are that the result is identical to reconstruction of the total network, only thrashing is prevented. If required, the reconstruction processes can run in parallel, on separate CPUs.

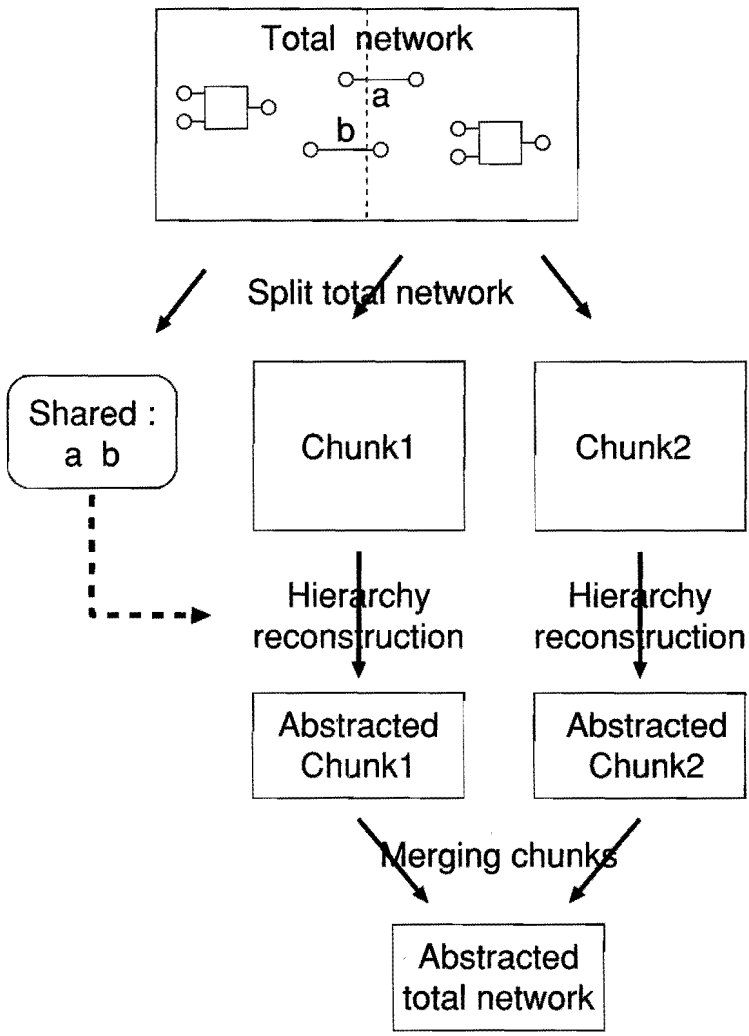


Figure 5.4: *Splitting a large design and applying hierarchy reconstruction*

Chapter 6

Results of the hierarchy reconstruction method

The verification method presented, has been used for many IC-designs in many design environments. This chapter discusses the results of hierarchy reconstruction method. In the first section, the hierarchy reconstruction process is taken of a representative design, to illustrate the method. The properties of the hierarchy reconstruction method, are then described in the second section.

6.1 The hierarchy reconstruction process for the TDA-1307

In the introduction, Chapter 1, the design process was partitioned into a high level design step, resulting in a high level structure, and a layout design step. The layout design step was shown in Figure 1.3, Section 1.3. In Figure 6.1, this figure has been redrawn, this time with the emphasis on the verification process of the layout generation part. According to Figure 6.1, the consistency between the layout and the high level structure is verified in three steps.

1. The first step consists of the extraction of the layout into a transistor level netlist.

2. Secondly all macro-cell instances must be reconstructed by using hierarchy reconstruction. If an instance is reconstructible, it's connectivity is correct.

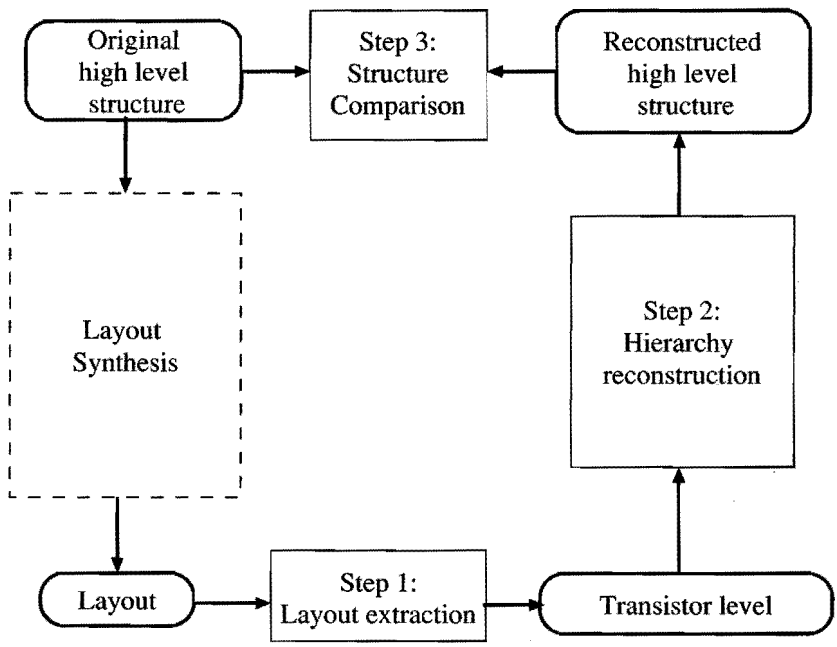


Figure 6.1: *Verification of layout versus original high level structure*

3. Finally, the connectivity between the high level components is verified by comparing the original, with the hierarchically reconstructed, high level structure.

When these steps have been performed successfully, the complete top-down trajectory from the highest structure level to layout is verified fully automatically.

To illustrate these steps, we have taken the design of the TDA-1307. The TDA-1307 performs the digital filtering of a decoded stereo digital audio signal prior to D/A conversion. It generates the 1-bit stereo input signal for a bit-stream DAC, for application in Compact Disk, Digital Compact Cassette, and Digital Audio Broadcasting. A detailed description of the design process of the TDA-1307 can be found in [Deloor92]. The layout of the design is given in Figure E.2 (see Appendix E). The TDA-1307 was designed by using the PIRAMID silicon compiler [Wouds90]. The TDA-1307 is a commercial product of Philips Semiconductors.

In the following, all jobs that will be mentioned ran on an HP-9000/735. The original high level structure of the TDA-1307 that has been mapped onto the layout, consists of 320 high level components.

1. The extraction of the TDA-1307 layout resulted in a network of 130 000 transistors, and took 45 minutes CPU-time.

2. The hierarchy reconstruction of the TDA-1307 takes in total 30 minutes CPU-time.

3. Comparing the connectivity of the original high level structure with the hierarchically reconstructed high level structure, in this case by using RECOGNIZE once again, takes 40 seconds CPU-time.

In the following, step 2 above, the hierarchy reconstruction from the transistor network to the high level structure is described in more detail. The reconstruction of a macro-cell instance usually consists of a mixture of non-parameterized and parameterized cell reconstruction. Just as has been described for the example in Figure 5.2, Section 5.2, the hierarchy reconstruction process starts with a number of non-parameterized hierarchy reconstructions by using FIND-AND-ABSTRACT, then continues with the parameterized reconstructions by using rules, and finally some FIND-AND-ABSTRACTs are applied, to collect the various high level sub-blocks that make up the macro-cells. In the following, the hierarchy reconstruction

is first described per parameterized macro-cell. Next, the hierarchy reconstruction is further analyzed by focussing on the non-parameterized reconstruction, i.e., all FIND-AND-ABSTRACT runs.

Results per parameterized macro-cell

As indicated in Chapter 3, one needs a parameterized cell library and a controller to perform the reconstruction for macro-cells of Category 1, 2 and 3. As described in Chapter 5, the controller is implemented in the

macro-cell generator	**category** (0,1,2 or 3)	**number of parameters**	**man weeks required**	
			layout view	**verification rule base**
RAM	2	5	104	3
AXU	1	1	3	0.4
CU/INC	1	1	5	0.4
ACU	3	2	52	1
ALU	2	8	52	2
MUX Register	2	4	52	2
ROM	3	3	52	2
PLA	3	3	15	3
MAC	2	4	52	2

Table 6.1: *Characteristics of the macro-cell generators and man weeks required for the generation of their layout view (including electrical design) and verification rule base*

Vera environment as a rule base. For the PIRAMID environment, a rule base has been developed for each macro-cell generator. Table 6.1 shows the characteristics per macro-cell type, the category, the number of parameters and the effort spent on each macro-cell generator. Note that this table does NOT enumerate the effort spent on the TDA-1307, but rather the effort spent on the PIRAMID macro-cell generators, that have been reused many times for different designs. The table shows that the extra effort needed to

implement the verification method, ranges from a day up to some weeks for complex parameterized macro-cells. In comparison with the effort spent on making the macro-cell generator's layout view, this effort is negligible.

Table 6.2 shows the run time effort spent on the verification of the actual TDA-1307 design. It enumerates the order of reconstruction, the number of instances reconstructed, the number of calls to RECOGNIZE, the number of components left after the current reconstruction step and the CPU time required per step. During this process, RECOGNIZE was

reconstruction order	**number of instances**	**calls to recognize**	**components left**	**CPU** (sec.)
Transistors			129804	
standard cells	6313	42	70511	479
RAM	3	486	46283	260
AXU	18	25	42144	12
CU/INC	1	8	41824	15
ACU	2	32	40457	49
ALU	2	62	38537	122
MUX Register	15	364	30197	513
ROM	4	697	12530	113
PLA	4	5841	10141	181
MAC	1	31	956	36
Analog, I/O cells	43	13	320	4
total		7601		1784

Table 6.2: *Hierarchy reconstruction process per macro-cell for TDA-1307.*

called 7601 times. Of these calls 6645 did not find matches. In total, RECOGNIZE consumed 1257 seconds CPU.

The following can be derived from Tables 6.2 and 6.1.

1. The total reconstruction time is less than the layout extraction time.

2. The total reconstruction time is dominated by the RECOGNIZE calls.
3. The run times correlate neither with the number of parameters per macro, the category per macro, nor with the number of recognize calls.
4. The table extractors of the PLA and ROM have the largest RECOGNIZE call count.
5. Most of the calls do not lead to matches.

The resulting high level structure is identical to the original high level structure, thereby proving the correctness of the implementation. Fabrication of the IC resulted in first time right silicon.

As has been described in Chapter 5, the reconstruction also records, per component, the bounding box, allowing a layout-like view of the reconstructed high level structure. In Figure 6.2 the bounding box views of the remaining network is shown. Obviously, the bounding box reconstruction is a simplified way of positioning the macro-cells, but even then Figures 6.2 and E.2 (see Appendix E) look rather similar. Since the bounding box extractors did not support table-extractor rules, the PLAs and ROMs are not shown in the figure. The many overlapping bounding boxes just below the center of the figure are the result of an extensive placement optimization, that led to a layout in which parts of different cells are strongly interleaved.

Results for non-parameterized hierarchy reconstruction

In order to analyze the results even further, this paragraph describes the results with respect to the rule `FIND-AND-ABSTRACT` (see Section 5), that has been applied at different places during the hierarchy reconstruction process. For the current design, Table 6.3 enumerates the worst case part of the run times relating to 144 component types to which `FIND-AND-ABSTRACT` has been applied. Like Table 4.1, the items are ordered according to the recognition effort. The total reconstruction time for non-parameterized cells is 949 seconds. The following remarks can be made.

1. A comparison between the total `FIND-AND-ABSTRACT` reconstruction time (949 seconds) and the total recognition time (707 seconds) illustrates that the recognition dominates the hierarchy reconstruction process.
2. Only for a few commonly occurring component types, such as RAM-cells and inverters, the recognition is not the dominating factor.

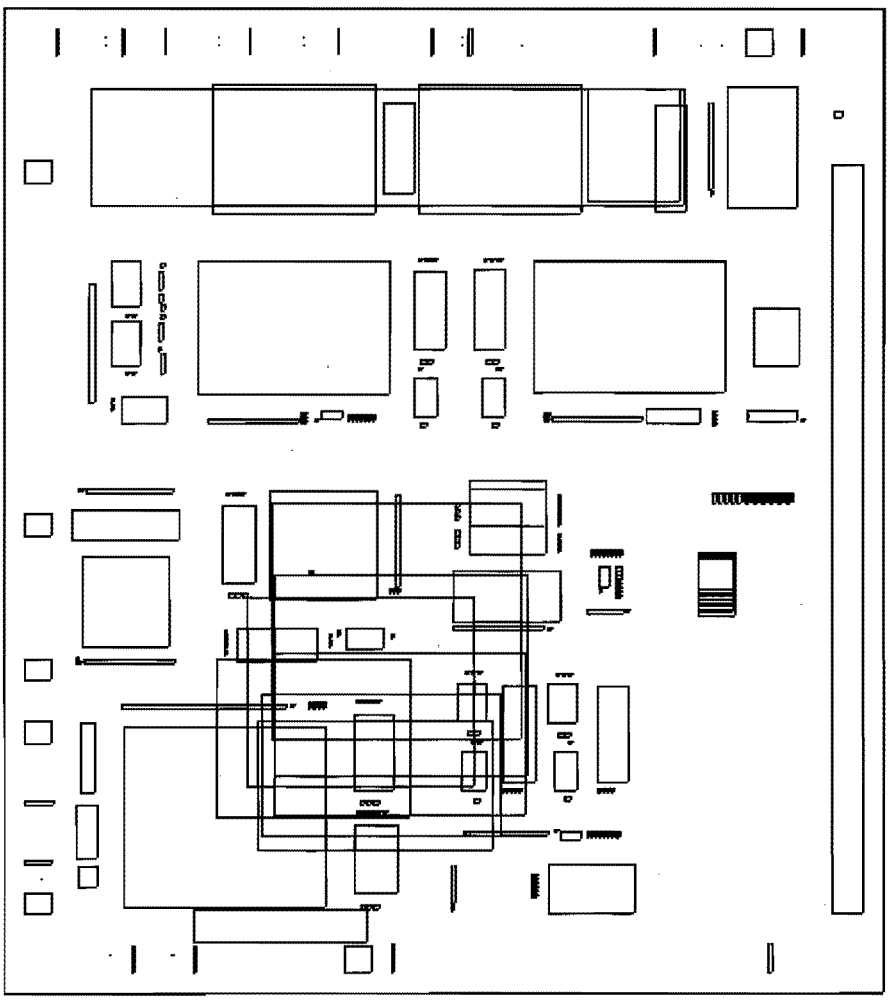


Figure 6.2: *Reconstructed high level structure of TDA-1307.*

Component Type	|V|	|V|	Match count	Recognize (sec.)	Reconstruction (sec.)
Noiseshaper	80014	1103	2	120.75	120.98
Nor3	56182	14	0	86.81	86.82
Nand4	78744	18	0	76.98	76.98
Dobm_ori	83564	1085	1	63.29	63.48
Axu_ori	84543	690	1	30.73	30.88
Pint_ori	82356	472	1	24.95	25.00
Iisinput_ori	77736	323	1	14.48	14.54
Myespa_ctrl	83914	375	1	13.80	13.84
Decoder2	47644	15	43	13.47	16.52
Ram-cell	173132	13	6300	13.35	38.87
Decoder3	47386	18	35	12.77	15.50
Decoder4	47106	21	50	11.14	15.48
Cs_29_sum	12729	27	255	10.79	12.28
Alignment	77868	125	1	10.06	10.09
Tcb_ori	84769	240	1	9.10	9.13
C3_nand	103351	10	673	9.02	12.44
Nand2	79861	10	256	7.46	8.71
C3_inv	119333	6	15472	6.39	64.10
Clockdiv	78699	101	1	6.09	6.10
Synchron	77812	90	1	6.06	6.07
Decoder1	46606	12	16	5.63	6.49
Invertor	92331	6	10936	4.71	66.15
Nand3	78828	14	12	4.65	5.40
Glbclock	103798	93	1	4.49	5.45
Cuinc	58423	73	1	4.34	4.69
Cs_24_mux	12905	14	20	4.16	4.48
Clockphins	103861	69	1	4.10	4.87
Acuslice	56624	53	16	3.92	4.03
C3_hlatch	91097	22	238	3.55	5.00
Dtn12tac	103711	36	12	3.29	5.78
C3_mux	97376	13	433	3.10	6.20
C3_latch	88003	20	294	3.01	4.73
Nand2_cinp	100659	8	3	2.86	2.94
Nand5	78744	22	6	2.83	3.65
Muxreg-top	39587	44	1	2.83	2.85
Reg-cell	52524	18	964	2.58	8.01

Table 6.3: *Non-parameterized hierarchy reconstruction results.*

3. In the total reconstruction time (30 minutes), the `FIND-AND-ABSTRACT` calls are dominant.

6.2 Error location

Error location is an important issue for verification. It will therefore be explained in more detail in this section.

A connectivity error can appear in the following ways.

1. Within a macro-cell, preventing it from being recognized as a macro-cell.

2. Within a macro-cell, preventing it from being recognized as the right macro-cell.

3. At the top-level, as a connectivity error among macro-cells.

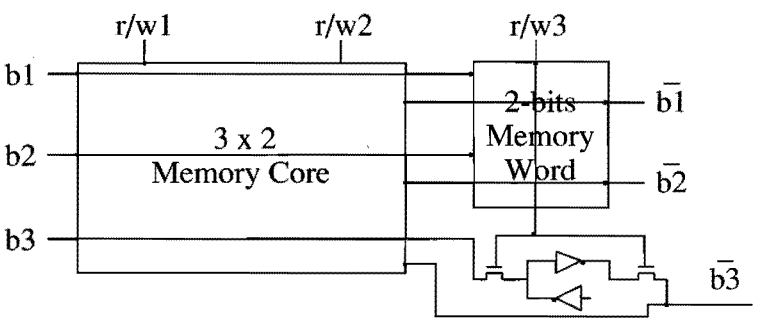


Figure 6.3: *Error location*

The first case is the most frequent one, because the number of connections inside all macro-cells is the largest, and macro-cells generally are not very much alike. This also makes the second case unlikely. The hierarchical approach of the reconstruction process is essential in enabling location of the first kind of errors quickly. Figure 6.3 shows a reconstruction result of a 3x3 memory core containing an error illustrating the first two cases. The error induces both the recognition of an unexpected macro-cell (a 2x3 memory core instead of a 3x3 memory core) and a small part of the circuit

remaining at intermediate levels: a 2-bit word, some inverters and transistors. It can quickly be seen that one inverter input is not connected, because all understandable parts are represented by high level elements. In this context it is easy to see that the inverter must be part of a memory cell, and the correct connection is now obvious (see also Figure 3.1).

Errors are located at the lowest level at which the reconstruction is stuck. This property is called hierarchical convergence.
Case 2 can be solved by comparing the number of expected instances per macro-cell to the occurring number.
For the third case, hierarchical convergence doesn't improve error location, because all hierarchy is already extracted. In this case errors can be found by netlist comparison at the highest structure level, since a reference network description is available at this level. Only a small number of macro-cell instances are left, compared to the number of transistors at the transistor level. It is therefore much easier to do the netlist comparison at this level.

With respect to PIRAMID, errors were found in the macro-cell generator descriptions, the macro-cell generator router, the leaf cell layouts and the highest level structure. In other environments, many errors have been detected as well, in all parts of the design trajectory. In general the hierarchy reconstruction method has proven to be both an efficient method of guaranteeing the absence of connectivity errors, and to be an efficient method of detecting connectivity errors.

6.3 Properties of the hierarchy reconstruction method

In this section the properties of the presented verification method are summarized. Structure verification by hierarchy reconstruction has been applied to many designs in different design environments. Based on this experience, a summary of properties resulted, described as follows:

- The error coverage of the verification method is complete, provided that the rule base and type descriptions are correct. By separating the writing of the macro-cell layout generator code and the writing of the macro-cell verification rule base, the probability that the same error occurs in both descriptions and thus escapes detection by the verification process, is very small.

- The verification process can be performed automatically. The problems relating to hidden hierarchy (see Section 5.4) are rarely encountered. When compared with simulation, this method is not burdened with the generation and interpretation of input and output patterns.
- The run times are less than the run times needed for layout extraction, even for very large circuits.
- Both digital and analog cells can be abstracted.
- The rule base is insensitive to technology changes (layout rules, electrical parameters), because the bottom-up description of a macro-cell is purely graph based.
- A verification rule base is fairly simple to construct, as it is independent of topology and other layout aspects. The effort spent in writing the rule base (up to a few weeks) is orders of magnitude smaller than the effort needed to create the layout generator view (several years).
- The hierarchical partitioning used during the reconstruction is independent from the hierarchical partitioning used for generating the layout. This means that any hierarchical view that may be imposed can be reconstructed. For the application, this is employed to impose a functional subdivision during the reconstruction, instead of a rectangular/layout like approach.
- Error location is straightforward due to the "hierarchical convergence" principle. Also, the RECOGNIZE primitive produces effective feedback when no matches are found.

Chapter 7

Final conclusions and future work

Verification has become a major bottleneck of the design process, if based on traditional techniques. Therefore, a recently proposed verification method called hierarchy reconstruction has been elaborated in this thesis. Starting from a layout, the method reconstructs hierarchically the highest structure level, in a bottom-up process.

The hierarchy reconstruction method can prove 100% correctness with respect to connectivity, in a very short time. It allows the automation of the verification process for a given design environment. Error location in the result of the verification process is easy due to an intrinsic property of the method called hierarchical convergence. In addition, the structure recognizer produces effective feedback when matching fails. The effort needed to set up the required rule base and component libraries is negligible compared to the effort in creating the layout view of a macro-cell generator.

The method has been implemented in the Vera environment. The flexibility and performance of this environment are crucial to the successful implementation. The efficiency of the method is mainly determined by the structure recognizer. The recognition algorithm is the core of the method, and its description forms the core of this thesis.

The application of the method to many designs, in several IC design environments, has highlighted the above mentioned advantages. So far, no connectivity error has passed the verification. The execution of a reconstruction job takes less time than the execution of the layout extraction job, that was done anyway to check the layout design rules. Designs containing

over a million transistors are verified without any difficulties. Numerous design errors have been signalled efficiently. With respect to the digital parts of a design, alternative verification means, such as switch-level simulation and transistor level netlist comparison, have been abandoned, or are applied only rarely in design environments that use hierarchy reconstruction. The method has become an important means to design correctly operating, complex IC-designs with limited effort.

The Vera environment in general combines structure analysis, recognition and structure manipulation. Therefore, it also supports structure synthesis, and additional electrical rule checking at any intermediate level. So far, this has only been applied from time to time. In the future, this might become more important.

The current trend in design is for the relation between layout and structure to become more and more important. Not only does the technological shrinking of typical layout dimensions contribute to a stronger relation between layout and structure, but also the design of combined analog and digital functions imposes strong demands between the network and layout of a design. Therefore, a future requirement will be to combine layout and structure recognition, enabling the checking of complex rules composed of layout and structure demands.

Other verification needs relate to high level design. In this area behavioral verification is at present mainly done by using a partial simulation. In the future, techniques must be developed to reduce the high level verification bottle-neck as well. Some formal verification techniques have proven very useful [Koste93], [Malik88], but still much work has to be done in this area.

Bibliography

[Veend92.1] H.J.M. Veendrick,
MOS ICs: from Basics to ASICS,
ISBN 1-56081-197-8, VHC Publishers Inc., 1992, p. 338.

[Veend92.2] H.J.M. Veendrick,
MOS ICs: from Basics to ASICS,
ISBN 1-56081-197-8, VHC Publishers Inc., 1992, p. 264-271.

[Veend92.3] H.J.M. Veendrick,
MOS ICs: from Basics to ASICS,
ISBN 1-56081-197-8, VHC Publishers Inc., 1992, p. 242.

[Graaf89] H.C. de Graaf and F.M. Klaassen,
Compact Transistor Modelling for Circuit Design
Springer-Verlag, New York - Wien, ISBN 0-387-82136-8.

[Nagel75] L.W. Nagel,
Spice2: A Computer Program to Simulate Semiconductor Circuits,
Memorandum No. ERL-M520, Electronic Research Laboratory, College of Engineering University of California, Berkeley USA, 9 may 1975.

[Feldm92] U. Feldmann et. al.,
Algorithms for Modern Circuit Simulation,
Allgemeine Electrische Übertragungen (Hirzel-Verlag Stuttgart), Vol. 46, 1992, no. 4, p. 274-285.

[Kleih94] E. Kleihorst,
Frequency Domain Analysis for Nonlinear Electronic Circuits,
Thesis TU-Delft, ISBN 90-74445-09-8.

[VHDL93] A. Hohl,
Proceedings of VHDL-forum for CAD in Europe,
Spring 1993, IFIP WG 10.2/10.5 ECIP Esprit 2072, Innsbruck Austria.

[Bryant87] R. Bryant,
Boolean analysis of MOS circuits,
IEEE Transactions on CAD, Vol. 6, no. 4, July 1987, p.634-649.

[Chua75] L.O. Chua and P. Lin,
Computer Aided Analysis of Electronic Circuits: algorithms & computational techniques,
Prentice-Hall, Inc., ISBN 0-13-165415-2, 1975.

[Jones94] L.G. Jones,
A cache-based method for accelerating switch-level simulation,
IEEE Transactions on CAD, Vol. 13, no. 2, February 1994, p.211-218.

[Apte82] R.M. Apte et. al.,
Logic function extraction for NMOS circuits,
Proc. Int. Conf. Circuits and Computers, October 1992, p. 324-327.

[Boehn88] M. Boehner,
An automatic logic extractor from transistor to gate level for CMOS technology,
Proc. IEEE DAC 1988, p. 517-522.

[Bolse89] I. Bolsens,
Electrical correctness verification of MOS VLSI digital circuits using expert system and symbolic analysis techniques,
PhD thesis 1989, IMEC/ Katholieke Universiteit Leuven.

[Ramme92] W. de Rammelaere,
Static electrical verification of synchronous digital MOS circuits,
PhD thesis 1992, IMEC/ Katholieke Universiteit Leuven.

[Malik88] S. Malik, A.R. Wang et. al.,
Logic Verification using Binary decision Diagrams in a logic

synthesis environment,
Proceedings of ICCAD'88.

[AFMC89] Internal Workshop on Applied Formal Methods For Correct VLSI Design,
IMEC-IFIP, Nov. 1989, Leuven Belgium.

[TPCD92] Internal Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience,
IFIP TC10/WG 10.2, June 1992, Nijmegen The Netherlands

[CHAR93] Correct Hardware Design and Verification Methods,
IFIP WG 10.2, CHARME '93, May 1993, Arles France.

[Genoe92] M. Genoe, L. Claesen et. al.,
Formal Verification of High Level Synthesis by means of SFG-Tracing.
High Level Synthesis Workshop 1992.

[Dever92] F. Deverchere, J.C. Madre et. al.,
Functional abstraction and formal proof of digital circuits,
Proceedings of EDAC '92, p. 458-462.

[Verli92] E. Verlind, L. Claesen, et. al.,
Partial strength ordering applid to symbolic switch level analysis,
Proceedings of EDAC '92, July 1992, p.388-392.

[Ebeli83] C. Ebeling and O. Zajicek,
Validating VLSI circuit layout by wirelist comparison,
Proceedings of ICCAD 1983, p. 172-173.

[Ebeli88] C. Ebeling,
Gemini II, a second generation layout validation program,
Proceedings of ICCAD 1988, p. 322-325.

[Waten83] T. Watanabe, et. al.,
A new automatic logic interconnection verification system for VLSI design,
IEEE transactions on CAD, vol. CAD-2, p. 70-81, April 1983.

[Spick83] R.L. Spickelmier and A.R. Newton,
WOMBAT: A new connectivity verification program,
Proceedings of ICCAD 1983, p. 170-171.

[Batra92] P. Batra and D. Cooke,
Hcompare: a hierarchical netlist comparison program,
Proceedings of DAC '92, p. 299-304.

[Pelz91] G. Pelz and U. Roettcher,
Circuit comparison by hierarchical pattern matching,
Proceedings of ICCAD 1991, pp. 290-293.

[Pelz94] G. Pelz and U. Roettcher,
Pattern matching and refinement hybird approach to circuit comparison,
IEEE Transactions on CAD, Vol. 13, no 2, Februari 1994, p. 264-276.

[ISCAS89] F. Brglez et.al.,
Combinational profiles of sequential benchmark circuits,
Proceedings of ISCAS 1989, p. 1929-1934.

[Jouppi87] N.P. Jouppi,
Derivation of signal flow direction in MOS VLSI,
IEEE Transactions on CAD, Vol. 6, no. 3, p. 480-490.

[Read77] R.C. Read and D.G. Corneil,
The graph isomorphism disease,
Journal of graph theory, Vol. 1 (1977) p.363-399.

[Reingol77] E.M. Reingold et. al.,
Combinatorial Algorithms, theory and practice,
1977, p. 319-322, Prentice Hall, Inc.

[Sedge88] R. Sedgewick,
Algorithms,
1988, p. 426, Addison-Wesley, Inc.

[Berzt73] A.T. Berztiss,
A backtrack procedure for isomorphism of directed graphs,
Journal of the ACM, Vol. 20, No. 3, july 1973, p.365-377.

[Purdom83] P.W. Purdom Jr.,
Search Rearrangement Backtracking and Polynomial Average Time,
Artificial Intelligence 21 (1983) p.117-133.

[Harar72.1] F. Harary,
Graph Theory,
1972, chapter 14, Addison-Wesley, Inc.

[Harar72.2] F. Harary,
Graph Theory,
1972, p. 16, Addison-Wesley, Inc.

[Stanat77] D.F. Stanat,
Discrete Mathematics in Computer Science,
1977, p. 317, Addison-Wesley, Inc.

[Koste89] A.P. Kostelijk,
Vera, a Rule-based Verification Assistant for VLSI Circuit Design,
Proc. of the VLSI 89 conference, p. 89-98, August 1989.

[Koste88] A.P. Kostelijk, G.G. Schrooten,
Vera, a Rule-based Verification Assistant for VLSI Circuit Design,
Proc. of ESPRIT Technical Week, Brussels 1988, p.263-268.

[Koste91] A.P. Kostelijk and B.J.S. De Loore
Automatic Verification of Library-based IC Designs,
IEEE Journal of Solid State Circuits, march 1991, Vol. 26, nr 3, p. 394-403.

[Koste92.1] A.P. Kostelijk, P.A. Kuppen, B.F. Lynch
Vera User manual,
Internal manual.

[Koste92.2] A.P. Kostelijk, P.A. Kuppen, B.F. Lynch
Vera Reference manual,
Internal manual.

[Koste92.3] A.P. Kostelijk, P.A. Kuppen, B.F. Lynch
Vera Rule manual,
Internal manual.

[Koste92.4] A.P. Kostelijk,
Method and device for tracking down a prespecified sub-circuit in an electrical circuit, method for constructing integrated circuit masks using the method,
Patent numbers: EP 05411170, KOKAI 93-233744

[Koste93] A.P. Kostelijk and A. van der Werf,
Functional verification for retiming and rebuffering optimization,
Proceedings of EDAC '93, p. 99-104.

[Kuppe89.1] P.A. Kuppen,
Functional Verification of Rom Layouts Using Vera,
Internal manual.

[Kuppe89.2] P.A. Kuppen,
Functional Verification of PLA Layouts Using Vera,
Internal manual.

[Deloor90] B.J.S. De Loore and A.P. Kostelijk
Automatic Verification of Library-based IC Designs,
Proc. of the CICC 90 conference, p. 30.6.1-30.6.5, May 1990.

[Deloor92] B.J.S. De Loore et. al.
The design of a competitive ASIC for the consumer market using the PIRAMID design system
Proceedings of IEEE ASIC 1992 conference, pp. 520-524.

[Wouds90] R. Woudsma et. al.
PIRAMID: an architecture-driven silicon compiler for complex DSP applications Proc. International Symposium on Circuits and Systems 1990, pp. 2596-2600.

[Rovers93] W.M.H.M. Rovers
Dracula LVS versus Vera, JAS-Stage at the IC-lab CE,
Philips' Nat. Lab. Technical Note Nr. NL-TN 268/93, p. 16.

[Rubic84] C. Lob,
RUBICC, A Rule-Based Expert system for VLSI Integrated Circuit Critique,
Memorandum no. UCB/ERL M84/80, University of California, Berkeley, September 1984.

[Spick88] Rick L. Spickelmier, A. Richard Newton,
Critic: A Knowledge-Based Program for Critiquing Circuit Designs,
Proc. of ICCD 1988, p.324-327.

[Nebel86] W. Nebel,
REX - Automatic Extraction of RT-Level Descriptions from Integrated Circuit Layout Data,
PhD Thesis 1986, University of Kaiserslautern

[Nebel87] W. Nebel and R.W. Hartenstein,
Functional design verification by register transfer net extraction,
COMPEURO 1987, p.254-257

[Papas88] Alexander C. Papaspyridis,
A Prolog based connectivity verification tool
IEEE DAC 1988, p. 523-537.

[Hirsch88] Mark Hirsch and Daniel Siewiorek,
Automatically extracting structure from a logical design,
IEEE ICCAD 1988, p. 456-460.

Appendix A

Summary

Verification of electronic designs by reconstruction of the hierarchy

This thesis describes an effective method to find errors in an electronic circuit design, prior to production. Complicated electronic circuitry, such as integrated circuits, are designed in a number of steps, to elaborate all aspects in a divide and conquer manner. A synthesis phase and a verification phase can be distinguished in each step. Synthesis is the activity to transform a specification into a design. Verification is the activity to check a design for possible errors. Complicated electronic circuitry may consist of millions parts. Therefore, verification is cumbersome and time-consuming. In the last decade many CAD-programs have been created that automate many of the tedious synthesis steps. The number of new verification programs has lagged behind, so most design effort is currently spent on the verification phases. The verification method described in this thesis is called hierarchy reconstruction.

Hierarchy reconstruction is a method to verify the structure of an electronic circuit. A circuit structure describes the network, consisting of basic components such as transistors, resistances and their interconnections. The method is based on pattern recognition. Based on a given sub-network with a given function, such as a memory-cell for example, a pattern recognizer is used to identify all sub-networks occuring in a design forming a memory-cell. As a next step, every sub-network is replaced by a single "function-component", such as a memory component. This step is called "abstraction". A combination of recognition followed by abstraction is called "re-

construction". By applying reconstruction for all intended sub-networks (functions), checks can be performed whether all basic components are used, and whether all connections within each function are correct. Hereafter, the recognizer can be used again to find and abstract new sub-networks, consisting of function-components, forming more complicated higher level functions. In this way a complete hierarchy of structures can be reconstructed, for which presence and connections of all functions are checked.

The pattern recognizer is the main tool to allow hierarchy reconstruction, and is therefore described in detail in this thesis. The hierarchy reconstruction of structures with repetitive patterns, such as "RAMs" (Random Access Memories) is described as well. When a reconstruction is not successfully completed, the structure of a design contains errors. An error can appear because of the following.

- Too many components are present in the design, or
- some functions are not implemented as intended,
  i.e, components are missing, connections are missing,
  unintended connections (short-circuits) are present.

An important advantage of the hierarchy reconstruction method is that correct parts of the design are reconstructed as intended, independent from errors at other places. By viewing parts that are reconstructed only partially, the cause of an error can quickly be traced.

The method has been implemented in a program called "Vera", an acronym of VERification Assistant. Vera is a general program for structure analysis and manipulation. It has been used for several years now by many designers within Philips. A representative verification example is elaborated, to show that very complex designs can be verified efficiently.

Appendix B

Nederlandse samenvatting

Verificatie van ontwerpen van electronische schakelingen door reconstructie van de hiërarchie

Dit proefschrift beschrijft een effectieve methode om een ontwerp van een electronische schakeling op fouten te controleren, voordat het ontwerp gebruikt wordt om de schakeling te fabriceren. Het controleren van een ontwerp op fouten wordt "verificatie" genoemd. Het proces om van een ontwerpdoel, de specificatie, tot een ontwerp te komen wordt "synthese" genoemd. Ingewikkelde schakelingen worden stapsgewijs ontworpen, waarbij in elke stap een deelaspect wordt uitgewerkt. Binnen één stap worden synthese en verificatie direct na elkaar toegepast, om na afloop zeker te zijn van het resultaat. Ingewikkelde schakelingen, zoals die zich bijvoorbeeld in een IC (een "chip") kunnen bevinden, zijn opgebouwd uit miljoenen onderdelen. Hiervoor is verificatie vanzelfsprekend een moeizame en tijdrovende taak. Doordat sinds enige jaren allerlei computer programma's gebruikt kunnen worden die de synthese vereenvoudigen, vergt nu verificatie de meeste inspanning voor de ontwerper. Voor verificatie zijn er tot dus ver relatief weinig bruikbare programma's ontwikkeld. De in dit proefschrift beschreven methode om een ontwerp van een electronische schakeling te verifiëren heet hiërarchie reconstructie.

Hiërarchie reconstructie is een methode waarmee de netwerkstructuur van een schakeling verifiëerd kan worden. Een netwerkstructuur beschrijft de opbouw van een schakeling, bestaande uit basiscomponenten zoals transistoren, weerstanden, etc., en hun onderlinge verbindingen. De methode maakt gebruik van een patroonherkenner. Uitgaande van een gegeven deel-

structuur met een bepaalde functie, zoals bijvoorbeeld een geheugenschakeling, kan de patroonherkenner in een ontwerp alle dergelijke deelstructuren identificeren. Elke op deze manier herkende deelstructuur kan vervangen worden door één "functie"-component, bijvoorbeeld een geheugencomponent. Dit wordt "abstractie" genoemd. De combinatie van herkenning gevolgd door abstractie noemen we "reconstructie". Door reconstructie toe te passen voor alle beoogde deelstructuren (functies), kan gekeken worden of alle basiscomponenten gebruikt worden en of de verbindingen binnen een functie goed zijn. Nadat alle basiscomponenten gereconstrueerd zijn tot functie-componenten, kan de patroonherkenner opnieuw gebruikt worden om een andere gegeven deelstructuur bestaande uit functie-componenten, die samen een ingewikkeldere functie vormen, te herkennen en te abstraheren. Zo kan een hele hiërarchie van structuren gereconstrueerd worden, waarvoor de aanwezigheid van en de verbindingen tussen alle functies gecontroleerd worden.

De patroonherkenner is de basis van de hiërarchie reconstructie methode, en wordt in dit proefschrift daarom in detail beschreven. Ook wordt hiërarchie reconstructie beschreven voor deelstructuren waarin repeterende patronen voorkomen, zoals bijvoorbeeld in zogenaamde "RAMs". Als in een ontwerp de reconstructie niet lukt, dan is de opbouw van het ontwerp onjuist. Hieraan kunnen de volgende oorzaken ten grondslag liggen:

- Er zitten overbodige componenten in het ontwerp, of
- sommige functies zijn niet geïmplementeerd zoals beoogd.

In dit laatste geval kunnen componenten missen, verbindingen missen, of er zijn niet-bedoelde verbindingen (kortsluitingen) aanwezig. Een belangrijk voordeel van de methode is dat de deelstructuren waar zich geen fouten voor doen, hiërarchisch gereconstructueerd kunnen worden onafhankelijk van eventuele fouten elders. Door die deelstructuren te bekijken die maar gedeeltelijk zijn gereconstrueerd, kunnen oorzaken van fouten snel opgespoord worden.

De methode is geïmplementeerd in het computer programma "Vera", wat een acroniem is voor VERificatie Assistent. Vera is een algemeen programma waarmee structuuranalyse (o.a. herkenning) en -manipulatie gedaan kan worden. Vera wordt al enige jaren met succes gebruikt door vele ontwerpers binnen Philips. Een representatief verificatie voorbeeld is uitgewerkt, om aan te tonen dat met deze methode zelfs de meest ingewikkelde ontwerpen effectief geverifiëerd kunnen worden.

Appendix C

Biography

Ton Kostelijk was born in 1961, in Grootschermer, the Netherlands. He received the MSc degree in physics, at the experimental solid state physics department of the Free University of Amsterdam, the Netherlands, in 1985, where he was involved in a project regarding electromigration in metalhydrides. In 1985 he joined the Philips Research Laboratories, Eindhoven, the Netherlands. Since then he is involved in CAD for digital and analogue circuit design, especially in the design verification part. Vera, an environment for structure verification and analysis was created, and also a lot of effort was spent on support and promotion activities. Since 1991, his focus is shifted towards behavioral verification, such as functional verification of retiming and formal verification.

Appendix D

List of symbols

Sets and multi-sets

$(x_1, x_2, \ldots)$	Ordered (multi-)set
$\{x_1, x_2, \ldots\}$	Unordered (multi-)set
$\mu_a(B)$	Multiplicity of element a in set B
2^B	Powerset over B
$\mathbb{N}^B$	Set of all multi-sets over multi-set B
B^+	Set of all non-empty sequences over B
$\|B\|$	Number of elements in (multi-)set B
$\cup$	Union set operator
$\cap$	Intersection set operator
$\setminus$	Set-minus set operator

Functions

$F : A \rightarrow B$	Function F with domain A and co-domain B
$F(A)$	Image of set A for function F
B^A	Set of all functions with domain A and co-domain B
$F\|_C$	Restriction of function F to set C (Def. 4.1)
$[a]_F$	Equivalence set of a for function F (Def. 4.2)
A/F	Quotient set of A with respect to function F (Def. 4.3)
$[True]_F$, $\mathcal{F}$	True-set of function F (Def. 4.5)
$A \bullet B$	Pair function of sets A, B (Def. 4.6)

Functions and sets of a circuit

τ	Non-empty set of types
Γ	Non-empty set of terminal classes
α	Set of attribute names
β	Set of attribute values
TTC	Type Terminal Classes function (Def. 4.7)
TA	Type Attributes function (Def. 4.8)
T	Type function (Def. 4.9)
$vert(e)$	Pair of vertices connected by edge e
G	Circuit (Def. 4.17)
V	Set of vertices
T	Type function (Def. 4.9)
C	Set of components (Def. 4.10)
N	Set of nets (Def. 4.11)
A	Attribute function (Def. 4.12)
E	Set of multi-edges (Def. 4.13)
TC	Terminal Class function (Def. 4.14)
TCS	Terminal Classes function (Def. 4.15)
$DEGREE$	Degree function (Def. 4.16)
$\|G\|$	Size of circuit G (Def. 4.17)
γ	Present classes function (Def. 4.24)
Adj	Adjacency function (Def. 4.25)

Sub-circuit recognition definition

$\mathbf{G}$	Main circuit (Def. 4.18)
$\mathbf{V}, \mathbf{T}, \mathbf{A}, \mathbf{E}, \mathbf{TC}$	Sets and functions of a main circuit (Def. 4.18)
$\mathbf{C}, \mathbf{N}, \mathbf{DEGREE}$	Sets and functions of a main circuit (Def. 4.18)
G	Circuit of a template circuit (Def. 4.19)
NE	Set of external nets of a template circuit (Def. 4.19)
V, T, A, E, TC	Sets and functions of a template circuit (Def. 4.19)
$C, N, NI, DEGREE$	Sets and functions of a template circuit (Def. 4.19)
w	Main circuit vertex
v	Template circuit vertex
$\mathbf{G}\|_{\mathcal{V}}$, $\mathcal{G}$	Sub-circuit (Def. 4.21)
$\mathcal{V}, \mathcal{T}, \mathcal{A}, \mathcal{E}, \mathcal{TC}$	Sets and functions of a sub-circuit (Def. 4.21)
$\mathcal{C}, \mathcal{N}, \mathcal{DEGREE}$	Sets and functions of a sub-circuit (Def. 4.21)
$\mathbf{S}$	Isomorphism predicate (Def. 4.20)
$\mathcal{S}$	Set of isomorphisms (Def. 4.20)
$\mathcal{M}$	Set of matches (Def. 4.23)

Backtracking

$\mathbf{Y}$	Search space (Def. 4.26)
Y_j	Coordinate of a search space
$\mathcal{Y}$, $\mathcal{Y}_j$	Candidate set (Def. 4.31)
$\mathbf{D}$	Search predicate (Def. 4.26)
$\mathbf{D}_j$	Partial search predicate (Def. 4.29)
$\mathcal{D}$	Solution set of search problem (Def. 4.26)
d_i	Demand function (Def. 4.28)
$\mathbf{d}$, $\mathbf{d}_k$	Demand set, partial demand set (Def. 4.28)
$\mathbf{p}$, $\mathbf{p}^{-1}$	Permutation, inverse permutation
$G_{\mathbf{p}}, V_{\mathbf{p}}, E_{\mathbf{p}}$	General search tree, its vertex set and edge set (Def. 4.27)
V_j	Vertex set at level j
$\|G_{\mathbf{p}}\|$	Traversal size of a search tree (Def. 4.30)
$\|G_{\mathbf{p}}\|_j$	Partial traversal size of a search tree (Def. 4.30)

Sub-circuit recognition algorithm

$\mathbf{V}^k$	Search space (Def. 4.32)
$\mathbf{D}$, $\mathbf{D}_j$	Search predicate, partial search predicate (Def. 4.33)
$\mathbf{s}$, $\mathbf{s}_j$	Search list, partial search list
$G_\mathbf{S}, V_\mathbf{S}, E_\mathbf{S}$	Search tree, its vertex set and edge set (Def. 4.35)
$\mathbf{V}^j$	Search tree vertex set at level j (Def. 4.35)
$\mathbf{w}$, $\mathbf{w}_j$	Search tree vertex
L_v	Local demand function (Def. 4.36)
F_j	Connectivity demand function (Def. 4.37)
d_i	Demand function (Def. 4.38)
G_j	Template sub-circuit
b_i	Template Neighbor Function (Def. 4.39)
P_j	Main circuit neighbor set (Def. 4.40)
$\mathcal{F}_{j+1}$	Tentative candidate set (Def. 4.41)
$\mathcal{Y}_j$	Candidate set (See Eq. 4.25 and Def. 4.42)
$\mathcal{U}_j$	Branching factor function (Def. 4.43)
U_1	Initial branching factor estimation function (Eq. 4.30, 4.32)
U_j	Branching factor estimation function (Def. 4.48)
$\mathcal{L}$	Initial candidates function (Def. 4.44)
B_j	Border Set (Def. 4.45)
$d_{(j+1,v)}$	Proto-demand function (Def. 4.46)
Par_j	Parallel function (Def. 4.47)

Appendix E

Layouts

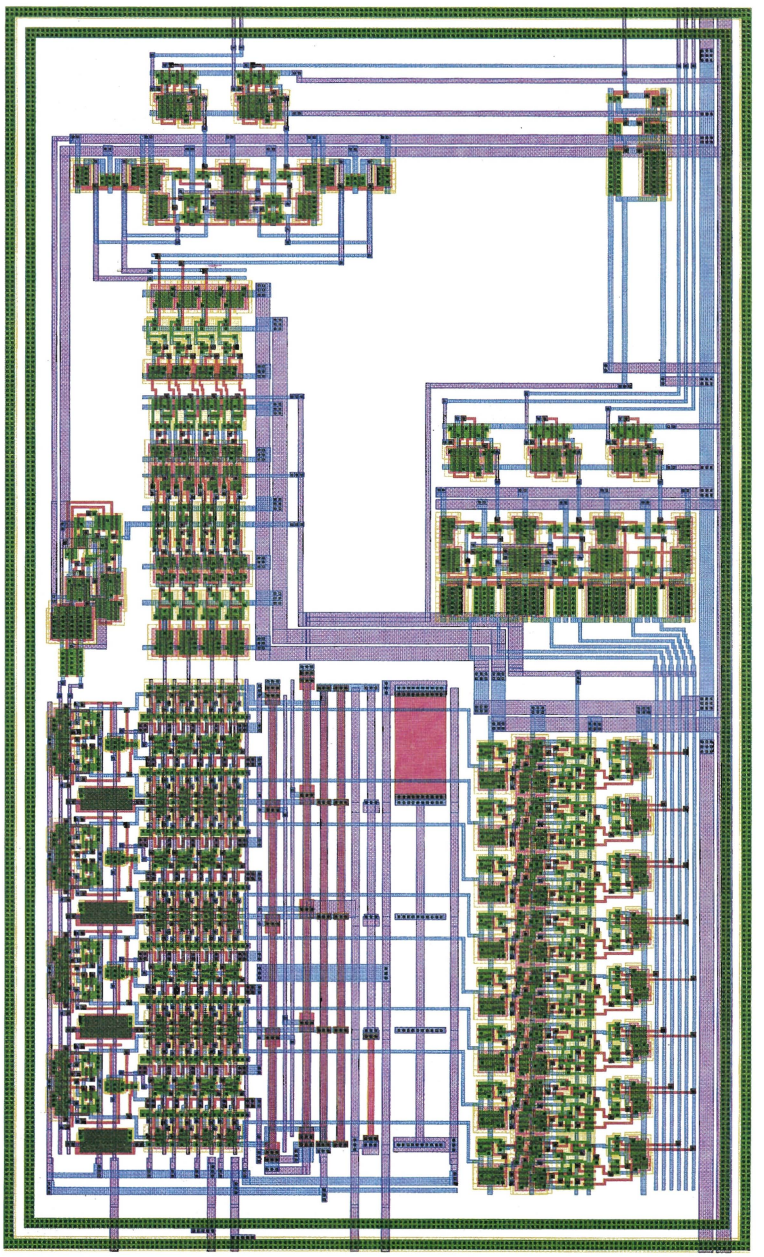


Figure E.1: *Example of a small layout in which two wires are exchanged. The error indication resulting from LVS is given in Figure E.1a.*

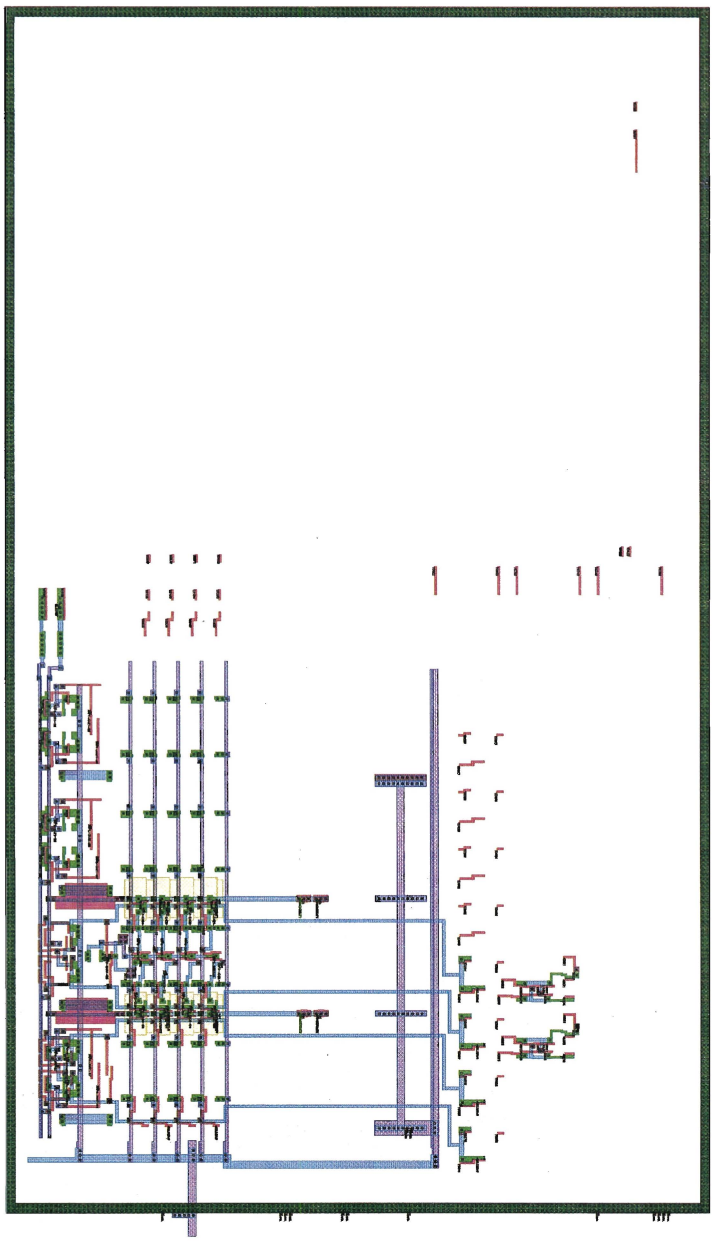


Figure E.1a: *The layout parts indicated in an LVS error report, as a result of the layout shown in Figure E.1. The error messages cover nearly half of the design.*

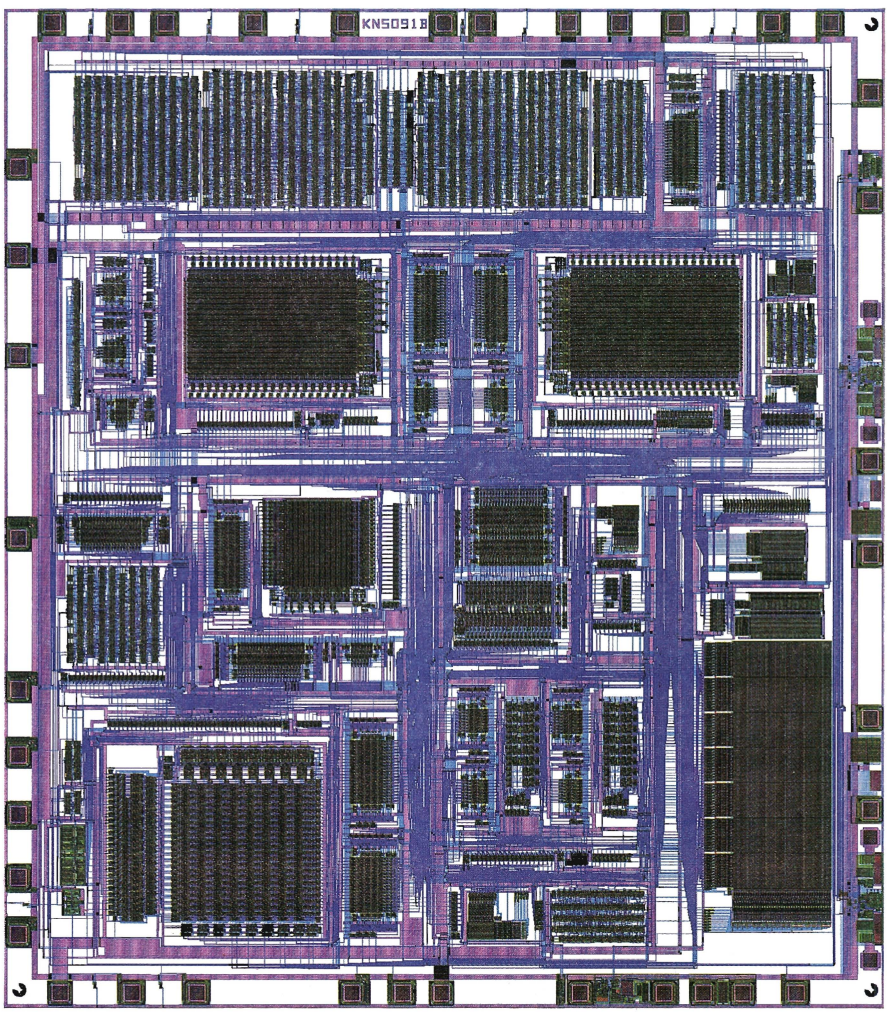


Figure E.2: *The layout of the TDA-1307.*

Stellingen

behorende bij het proefschrift

Verification of electronic designs by reconstruction of the hierarchy

door A.P. Kostelijk

1. In het IC ontwerp proces kost verificatie de meeste tijd en inspanning. Vreemd genoeg worden research gelden nog steeds voornamelijk aan synthese projecten besteed.

2. Het herkennen van structuren in een netwerk is een NP-volledig probleem; desalniettemin bestaan geschikte algorithmen voor toepassing bij IC ontwerp (dit proefschrift, hoofdstuk 4).

3. De waarde van research voor IC ontwerp programmatuur is van generlei waarde zonder beproeving in een industrieel ontwerp proces. Daarom zou het verplicht gesteld moeten worden dat onderzoekers op universiteiten hun programmatuur laten beoordelen door de industrie. Dit zou bovendien een geschikt middel zijn om het kaf van het koren te scheiden in de onafzienbare hoeveelheid publicaties.

4. Net zoals de constructeur van een auto niet de beste coureur is, is de constructeur van een programma niet de handigste gebruiker.

5. Een belangrijk maar minder bekend effect van een globale garbage-collector in een programma is dat thrashing (excessief "pagen") minder gauw optreedt.

6. Het schrijven van een goede gebruikershandleiding is meer werk dan het schrijven van een goed computerprogramma.

7. Een overeenkomst tussen discrete wiskunde en roddelbladen is dat beide zich bezig houden met relaties en, mogelijkerwijs vermeende eigenschappen daarvan.

8. De lagere school beoordelingen voor "gedrag" en "vlijt" zouden op de middelbare school ook gegeven moeten worden, maar dan zonder default waarde, omdat dit belangrijkere indicatoren zijn voor de toekomst dan de andere rapportcijfers.

9. Dat deregulering door de overheid een farce is,
blijkt o.a. uit de gang van zaken m.b.t. het kenteken Deel III.
De redenen om deze in te voeren (o.a. autodiefstal bemoeilijken)
blijken achteraf niet op te gaan, maar deze regeling daarom weer
afschaffen kan blijkbaar niet.

10. Naarmate een vakgebied exacter is hebben de meeste mensen
er minder interesse voor.

11. Een cultuur wordt sterker bepaald door de ontwikkeling van de
techniek dan door de ontwikkeling van de kunst.

12. Er is sprake van beroepsdeformatie,
als bij het lezen van een krant bij de woorden muis, file, bit,
in eerste instantie gedacht wordt aan klikken i.p.v. piepen,
gegevens i.p.v. auto's, bytes i.p.v. paarden.

13. De invoering van "klaag belasting" zou het geldtekort van
de overheidsbegroting snel doen veranderen in een overschot.
Bovendien zou hierna voor elke belasting ambtenaar recursie
een bekend en vaak toegepast begrip kunnen worden.