TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

# Netlist comparison with components with recursively interchangeable terminal groups

van Rootselaar, G.J.

*Award date:*
1995

Link to publication

**Disclaimer**
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit Eindhoven
Faculteit Elektrotechniek
Vakgroep Automatisch Systeemontwerpen (ES)

# Netlist comparison with components with recursively interchangeable terminal groups

G.J. van Rootselaar

Begeleiding:

Ir. G.L.J.M. Janssen (Technische Universiteit Eindhoven)
Dr. A.P Kostelijk (Philips Research Eindhoven)
Ir. W.M.H.M. Rovers (Philips Research Eindhoven)

Hoogleraar:
Prof. Dr. Ing. J.A.G. Jess

De faculteit der Elektrotechniek aanvaardt geen verantwoordelijkheid voor de inhoud van stage- en afstudeerverslagen

# summary

This thesis describes a method to compare descriptions of electronic designs at equal levels of abstraction in the structural domain. Nowadays the use of computer aided design tools in the design stage is commonplace. One of the functions of these tools is to transform high level structural descriptions into gate level descriptions. Furthermore, structures from different sources are combined. Since errors may be introduced in these processes, it must be verified whether the implementations are consistent with their high level specifications. To ease correction, a verification tool should also locate discrepancies. One method of performing a verification, is to make an abstraction of the implementation to the same level of detail as the specification, and compare it with its specification.

Successful methods to compare large gate level structural descriptions exist. They are based on the heuristic refinement algorithm for determining graph isomorphisms. The low level descriptions have the distinguishing feature that the pin permutabilities of the components can be represented by assigning a terminal class to each terminal. Terminals which belong to the same class may then be freely interchanged. Thus the permutation set is described by local properties of the terminals. This is a requirement for the refinement algorithm.

Higher level descriptions generally consist of components with more complex terminal interchangeability relations which cannot be represented by local properties of terminals. Instead, a permutation set is given for each component type. This renders the refinement algorithm unsuitable for direct comparison of high level descriptions.

The solution to this problem, which is described in detail in this report, lies in transforming the reference and test netlists into a form which can be described using terminal classes. Naturally, such a transformation must be invariant under graph isomorphism. The main idea is to (1) represent the permutation sets of the component types in the form of trees with terminal classes, and (2) to replace every component by a tree describing its terminal permutation set. Using this approach, netlists with components with recursively interchangeable terminal groups are transformed into forms which can effectively be compared using the refinement algorithm.

STRICT, a new comprehensive tool which performs the abovementioned transformation and isomorphism checks has been developed. The algorithm used for the isomorphism check is based on that of an earlier tool, named Gemini, which was originally developed at Carnegie Mellon University and improved at Philips research. The main improvements of the fault location part of STRICT with respect to Gemini, are a more precise refinement procedure and an improved handling of parameter values. Despite an inherent deficiency of the refinement algorithm, it performs well for practical circuits. When the discrepancy between the reference and test netlists is large, however, the number of reported discrepancies can become extremely large making fault location hard.

# Contents


**1 Introduction** **3**
- 1.1 Design of integrated circuits . . . 3
- 1.2 Verification of descriptions . . . 6
- 1.3 Netlist comparison . . . 6
- 1.4 Outline of this thesis . . . 11

**2 Transformation** **12**
- 2.0.1 General notions . . . 12
- 2.1 Representation of netlists . . . 13
- 2.2 Recursive interchangeability relations . . . 18
  - 2.2.1 Permutation trees . . . 19
  - 2.2.2 Mapping of a permutation tree onto a sub-netlist . . . 20
  - 2.2.3 Insertion mapping . . . 22

**3 Construction of an isomorphism mapping** **23**
- 3.1 The refinement algorithm . . . 23
  - 3.1.1 An informal discussion . . . 24
  - 3.1.2 A formal discussion . . . 29
- 3.2 Netlist comparison . . . 33
  - 3.2.1 Termination . . . 33
  - 3.2.2 Component types and terminal classes . . . 34
  - 3.2.3 Parameter values . . . 35
  - 3.2.4 Locating discrepancies . . . 37
  - 3.2.5 Functions and algorithms . . . 41

**4 Implementation** **56**
- 4.1 Use . . . 56
  - 4.1.1 Command line options . . . 57
  - 4.1.2 File formats . . . 58

**5 Results** **61**
- 5.1 Approach . . . 61
- 5.2 Missing components . . . 62
- 5.3 Interchanged component terminals . . . 67
- 5.4 High level designs . . . 71

**6 Conclusions** **72**
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

# Chapter 1

# Introduction

This chapter introduces and provides a motivation for the subject of this thesis. The subject is structure verification of electronic designs that are described at equal levels of abstraction. The chapter consists of four sections. The first part gives a brief overview of the integrated circuit design process. The second part considers verification. The third section focuses on netlist comparison and netlist comparison with components with recursively interchangeable component terminals in particular. It provides a motivation for the research presented in this report. The final section of this chapter gives an overview of the netlist comparison process.

## 1.1 Design of integrated circuits

The objective of integrated circuit (IC) manufacturing is to provide chips with a specified electrical functionality under given constraints. The trajectory from an initial concept to a final working integrated circuit encompasses four main stages: *design, fabrication, testing* and *packaging* [9]. We shall consider only the design stage. The aim of the design stage is to create a physical description of an integrated circuit, starting from an idea. It can be subdivided into three main tasks: *conceptualization* and *modeling*, *creation*[1], and *validation*.

The first part consists of creating a model of the idea, which describes the functionality of the circuit.

The goal of the creation task is to generate a detailed model of a circuit, such as a geometric layout, that can be used for fabricating the chip. This objective is achieved by iteratively performing transformations on the circuit. These transformations yield different intermediate descriptions in different domains and with different levels of detail. The three domains that can be distinguished are the *behavioral domain*, the *structural domain*, and the *physical domain*. We consider here three main levels of abstraction, namely: *architectural*, *logic*, and *circuit*. The domains and levels of abstraction can be conveniently represented by means of Gasjki and Kuhn's Y-chart as in figure (1.1). This figure also includes examples of the basic elements which constitute the various descriptions.

Descriptions in the behavioral domain specify the relations between the inputs and outputs of the circuit in functional or algorithmic form. Descriptions in the structural domain specify the circuit in terms of components and interconnections, and are generally referred to as netlists. Descriptions in

[1] The term *creation* is used here instead the usual term *synthesis*, to distinguish the task from another more specific kind of synthesis which is defined further on in the text.

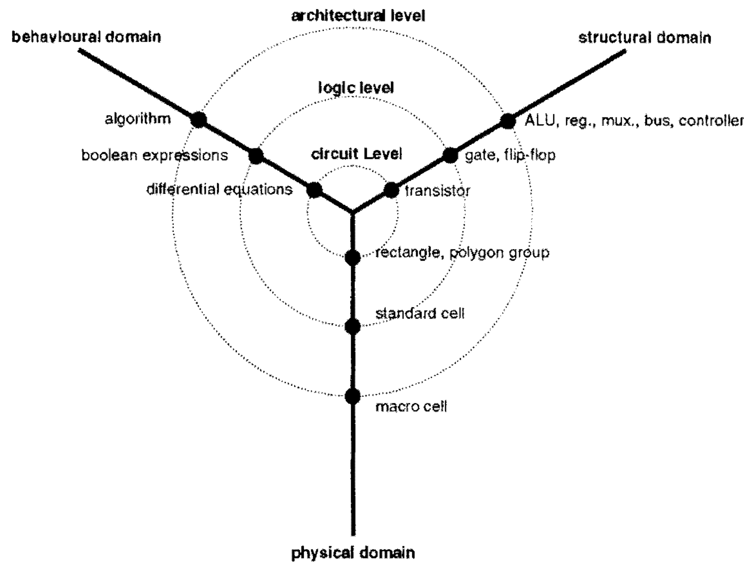



Figure 1.1: Gasjki and Kuhn's Y-chart. From [7, 9]

the physical domain have a geometrical character.
Various transitions between domains and levels of detail can be distinguished [7]. They are introduced here:

*refinement* Refinement adds more detail to a description (specification), yielding a new description in the same domain. The new description is referred to as the implementation of the specification.

*abstraction* Abstraction makes a description more abstract, yielding a new description in the same domain. It can be used for verification as will become apparent further on. Generally speaking, a detailed description can have multiple abstract descriptions.

*synthesis* Synthesis transforms a behavioral description (the specification) into a structural description at the same abstraction level. The new description is referred to as the implementation of the specification. A behavioral description can have more than one implementation in the structural domain.

*analysis* Analysis adds behavioral information to structural information yielding a description in the behavioral domain at the same abstraction level. It can be used for verification. A structural description implies its functionality.

*generation* Generation transforms a structural description (the specification) yielding a description in the physical domain at the same abstraction level. The new description is referred to as the implementation of the specification.

*extraction* Extraction transforms a description in the physical domain into a description in the structural domain at the same abstraction level. It plays an important role in validation. Generally speaking a layout implies a description in the structural domain at the lowest level (consisting of transistors and resistors).

*optimization* Optimization transforms a description (the specification) into a description in the same domain which has the same overall functionality but which yields a more optimal final layout. The new description is referred to as the implementation of the specification.

The above mentioned transitions are represented graphically on the Y-chart in figure (1.2). Note that refinement, synthesis, generation, and optimization yield implementations of specifications.

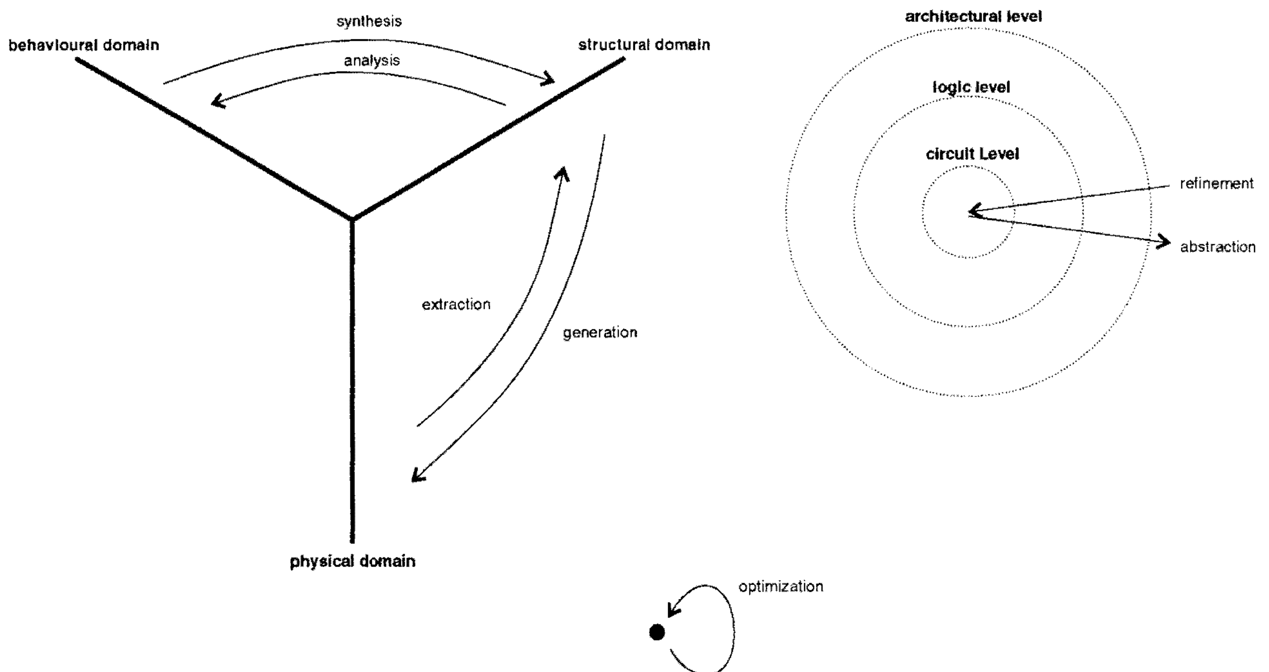



Figure 1.2: Transitions between different levels of abstraction and domains

Since circuits with transistor counts in excess of one million are not uncommon nowadays, the use of automated design systems is inevitable. These systems may introduce errors, yielding incorrect implementations with respect to their specifications. Furthermore, computer aided design (CAD) tools are limited to certain application classes thus forcing some degree of manual design. Therefore layout designs must be completely validated before proceeding to the expensive chip manufacturing.

The validation task must establish whether the implementations are correct with respect to their specifications. It can be performed by *simulation* and *verification* methods. Circuit simulation consists of analyzing circuit variables over an interval of time for specific sets of test vectors. That is, the *response* of an implementation to inputs is measured and compared with that of the specification. It will not be considered here. Verification is the process of comparing the entire *description* of the implementation with that of the specification.

## 1.2 Verification of descriptions

The ultimate goal of verification is to determine whether the description of the final chip layout is consistent with its initial functional specification. If discrepancies exist these should be indicated in such a way that they can easily be corrected. They can indicate errors made by the designers, or errors in the design software.
Again the total verification can be subdivided into different verification tasks. To verify whether implementations are consistent with their specifications an extraction, abstraction, or analysis of the implementation is first made, yielding a description in the same domain and at the same abstraction level as the specification. In a narrow sense the term verification applies to the comparison of descriptions within the same domain. Verification of descriptions in the behavioral domain is referred to as *functional verification*. Verification in the structural domain is referred to as *structure verification*.

Purely combinatorial circuits can be described in canonical form in the behavioral domain using BDD's [7], allowing functional verification. Unfortunately components such as multipliers yield BDD's that grow more than linear in the number of bits of the operands, making a purely functional verification of descriptions with such components hard for practical circuits with bus widths in excess of 14 bits. Due to the difficulties involved in the verification of sequential logic, hybrid forms of verification are also used, where circuit descriptions consisting of memory elements connected via canonical behavioral descriptions of combinatorics are verified. An example of such a tool is the Philips tool YATC [13].

Structure verification of descriptions at equal levels of abstraction is the subject of this thesis. Many verification tools have been developed to perform structure verification of descriptions at equal levels of abstraction. Even though these tools function well for descriptions with low levels of abstraction, such as the circuit level, making them suitable for layout vs. schematic comparison (LVS), they suffer from a deficiency which makes them less suitable for direct comparisons at the architectural and logic level. Why will become apparent in the next section. With the advent of structure verification tools that perform abstraction to the logic and architectural level [8, 11, 10] it becomes necessary to perform structure verification at higher levels of abstraction. Furthermore the heuristics used in the structure verification tools make them impractical for use with the mixed verification mentioned in the previous paragraph.

## 1.3 Netlist comparison

The term netlist comparison stems from the fact that the topology of a circuit can be represented in textual form by means of a netlist. An example of a structural representation of a circuit (a schematic) with a corresponding netlist is given in figure (1.3). In this example type definitions are not shown. The aim of netlist comparison is twofold: to determine whether two descriptions are structurally equivalent, and to locate the discrepancies.

Before considering netlist comparison, first the difference between the descriptions at the various levels of abstraction is discussed.

At the circuit level, the basic components of a description are transistors and resistors and their interconnections. Resistors are symmetrical components in the sense that their two contacts cannot be distinguished. In a similar sense, for digital circuits, transistors are often assumed to be symmetrical in the sense that the drain and source contacts cannot be distinguished. The drain and source terminals are in

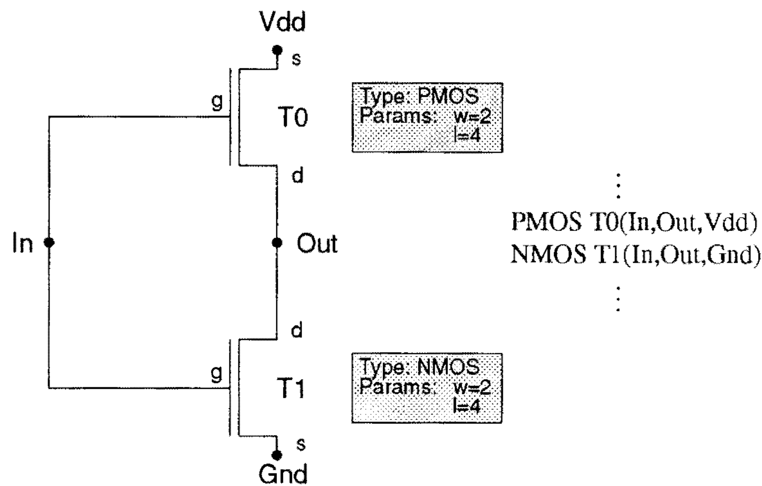



Figure 1.3: Example of a schematic with a corresponding netlist section

other words, permutable. The permutation set equals $\{\{(g,g),(d,d),(s,s)\},\{(g,g),(d,s),(s,d)\}\}$. The terminal permutation sets of these components can also be represented by simply associating (numerical) labels with the terminals in such a way that terminals which may be interchanged have the same label. The labels are referred to as *terminal classes*. The permutation set of the transistor type mentioned, for example, can be represented by associating a '0' with the gate terminal, a '1' with the drain terminal, and a '1' with the source terminal also.

At higher levels of abstraction, such as the logic and architectural level, the components may have pin permutabilities which cannot be represented using terminal classes. Consider for example a 'set-reset' (S-R) flip-flop. For this component the '$s$' and '$r$' terminals may be interchanged only if the '$q$' and '$\overline{q}$' terminals are also interchanged. The permutation set, in other words, equals $\{\{(s,s),(r,r),(q,q),(\overline{q},\overline{q})\},\{(s,r),(r,s),(q,\overline{q}),(\overline{q},q)\}\}$. Another example of a component with a groupwise interchangeability of the terminals is the And Or Invert (AOI) gate.

To determine whether two netlists are isomorphic, one tries to construct an isomorphism mapping of the reference netlist onto the netlist which is to be validated (the *test* netlist). An isomorphism mapping is a $1-1$ mapping which preserves the adjacency of the nets and the components. That is, a component and a net in the test netlist are adjacent if and only if the corresponding components and nets in the reference netlist are adjacent. Furthermore it is required that the types and parameter values of the components mapped onto each other match. Also for each component the mapping of the terminal names in the test netlist onto the terminal names in the reference netlist must be a member of the permutation set for that component. Note that a mapping of the terminals is implied by a mapping of the components and the nets. In the case where permutation sets can be described by assigning terminal classes to the terminals the last requirement is equivalent with the requirement of equal terminal classes for terminals that are mapped onto each other.

An example of an isomorphism mapping is given in figure (1.4).

Existing algorithms for proving netlist isomorphism are based either on depth-first search, or on refinement [6, 5, 4, 3].

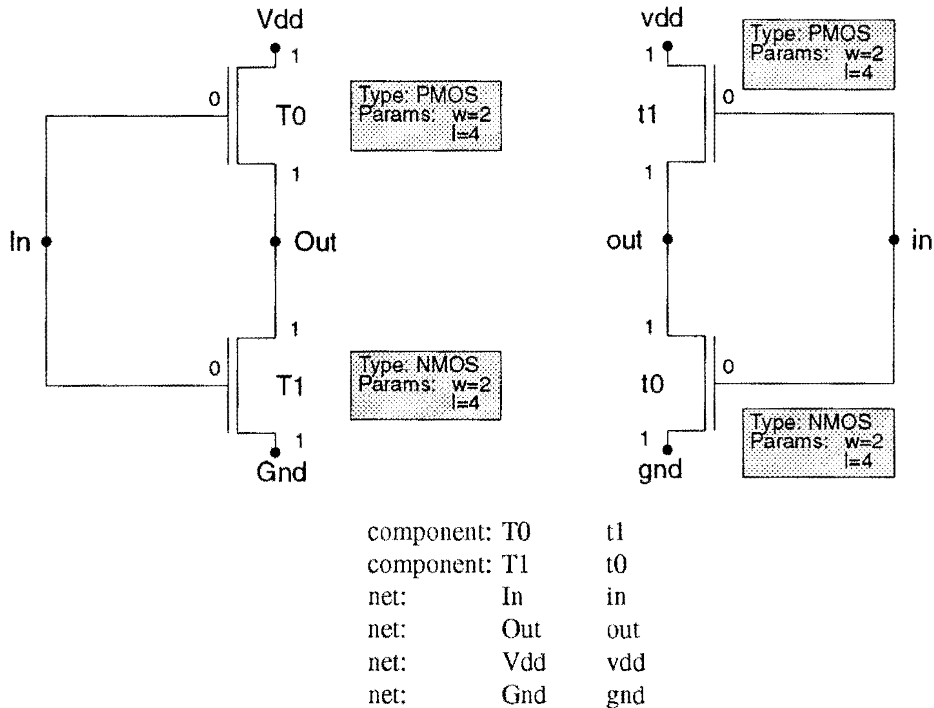



Figure 1.4: Two isomorphic schematics with mapping

### Depth-first search

This paragraph describes briefly the depth-first search algorithm for determining netlist isomorphisms as in [8]. The algorithm first determines 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, 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, terminal permutability, and 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, however, this approach leads to unacceptable run times. Therefore, most modern algorithms for proving netlist isomorphism are based on refinement. The netlist comparison tool described in this thesis also uses refinement.

### Refinement

The principle of refinement [1] is described here briefly in order to show where the problems lie in comparing high level structures. It will be explained extensively in chapter 3.
The components and nets in the two netlists are iteratively partitioned into sets of elements having

equal signatures. The initial partitioning is based on initial *local* properties of the components and nets, such as the types of the components and the number of connections to the nets. In every iteration that follows the initialization, the signature of every element is reassigned a value by combining the current signature, the current signatures of the *direct* neighbors, and the properties of the terminals connecting them, using a signaturing function. When both netlists are partitioned into singleton sets, the pairs of elements having equal signatures form an isomorphism mapping.

This method works for structural descriptions at the circuit level where the permutability of the component terminals is represented by associating terminal classes with the terminals, since here the permutation set is defined by the local properties of the terminals. The approach cannot be used to find an isomorphism mapping for netlists with components with arbitrary terminal permutation sets however. Since the permutation set of the components specify relations between the terminals, the permutation set is not a local property of the component or its terminals. It can therefore not be used in conjunction with the refinement algorithm.

### Permutation trees

A solution to the abovementioned problem has been found. It will be fully explained in chapter two. Here an informal introduction is given.

The aim is to map the netlists onto netlists with interchangeability relations which can be described with terminal classes (such as the circuit level description that has been mentioned) so that they can be compared using the refinement algorithm. The basic idea is to represent the permutation sets by means of structures in the form of trees. These trees are inserted for every instantiation of the components for which they describe the terminal permutation sets. The permutation sets are represented by the trees in such a way that the isomorphism sets of the trees onto themselves (automorphism sets) define the valid terminal permutations. Because a netlist isomorphism implies isomorphism mappings of its sub-graphs, a netlist isomorphism mapping then implies valid terminal permutations.

To describe a permutation set of a component using a tree, each terminal of a given component type is associated with a leaf of a tree. Then a set of valid bijective mappings of the tree leaves onto themselves is defined using the tree structure. The valid mappings have the following properties: Any two vertices (including the leaves) that share the same parent can be mapped onto each other if and only if the labels of the edges connecting them to their parent are equal. Furthermore the mapping must preserve adjacency of all the vertices.

As an example consider the permutation set of the S-R flip-flop:
$\{\{(s,s),(r,r),(q,q),(\overline{q},\overline{q})\},\{(s,r),(r,s),(q,\overline{q}),(\overline{q},q)\}\}$. It can be represented by the permutation tree shown in figure (1.5.a). Part (b) of the same figure shows graphically one of the valid mappings.
Thus we have described the permutation of the component terminals by means of a tree with labeled edges. Such a tree is associated with each component type. By inserting such a tree for each component instantiation in the netlist, a netlist is obtained for which isomorphism can be checked using the refinement algorithm since the permutation sets are completely described by terminal classes. An example of such an insertion is shown in figure (1.6). Here an S-R flip-flop is replaced by its permutation tree. Several details have been omitted here. These include the fact that trees cannot be simply inserted into the netlist. The component type and parameter data, for example, disappear.

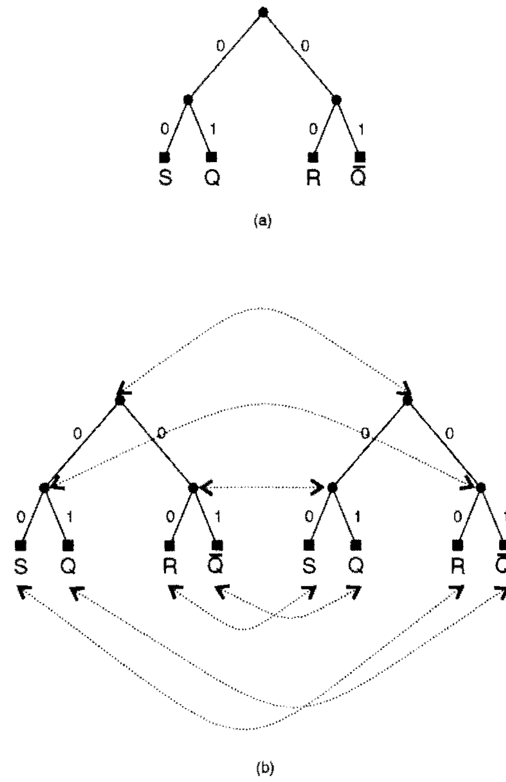



Figure 1.5: Permutation tree and a valid mapping

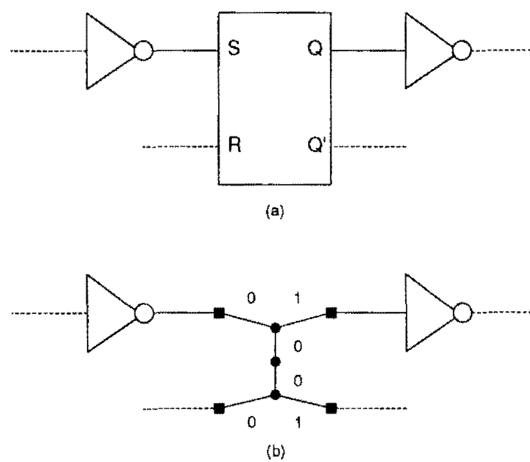



Figure 1.6: Insertion of a permutation tree into a netlist

## 1.4 Outline of this thesis

The process of verifying netlists at high levels of abstraction can be subdivided into two main tasks. The first task involves transforming the netlists into forms which can compared using the refinement algorithm. It is described in chapter two. The second task, which is described in chapter three, involves finding an isomorphism mapping of the reference netlist onto the test netlist and reporting the discrepancies in case the two netlists are not isomorphic. Many of the heuristics deployed have been inspired by those used in another netlist comparison tool named Gemini [6, 5, 4, 3]. Chapter four describes an implementation called STRICT (which is an acronym for STRucture Iso Check Tool). The results are discussed in chapter five. Finally conclusions and recommendations are given in chapter six.

# Chapter 2

# Transformation

This chapter describes a formal representation of netlists with components with arbitrary terminal permutation sets and a mapping of these netlists onto netlists with terminal classes.

### 2.0.1 General notions

- A *set* is a finite collection of elements in which each element occurs once. It is denoted by $S = \{s_0, s_1, \ldots, s_{|S|-1}\}$ where $s_i$ $(i = 0, 1, \ldots, i_{|S|-1})$ are the elements in the set. $|S|$ denotes the number of elements in the set, i.e. the cardinality. An empty set is denoted by $\emptyset$. Note that the definition of a set given here, corresponds to the traditional notion of a *finite* set. This has been done since we will be dealing only with finite sets.
- An *ordered set* is a finite, ordered collection of elements in which each element only occurs once. It is denoted by $O = (o_0, o_1, \ldots, o_{|O|-1})$ where $o_i$ $(i = 0, 1, \ldots, o_{|O|-1})$ are the elements of the ordered set. $|O|$ denotes the number of elements in the set, i.e. the cardinality. An empty ordered set is denoted by $\emptyset$. The $i$-th element (counting from zero inclusive onwards) of an ordered set $O$ is sometimes denoted by $[O]_i$. It is, when possible, denoted simply by $o_i$. The definition given here, corresponds to the traditional notion of a *finite* ordered set.
- A *multi-set* is a finite collection of elements in which each element may have multiple occurences. It is denoted by $M = \underline{\{}m_0, m_1, \ldots, m_{|M|-1}\underline{\}}$ where $m_i$ $(i = 0, 1, \ldots, i_{|M|-1})$ are the elements of the multi set. $|M|$ denotes the number of elements in the multi set, i.e. the cardinality. An empty multi set is denoted by $\emptyset$. The definition of a multi set given here, corresponds to the traditional notion of a *finite* multi set. When discussing a multi-set $M$, it is sometimes referred to simply as a *set* if it is clear from the context that we are dealing with a multi-set. The binary operator $\underline{\cup}$ is used to denote the union of mult-sets. Thus $M_0 \underline{\cup} M_1$ is the union of the multi sets $M_0$ and $M_1$. The multiplicity, i.e. the number of occurrences, of an element $m$ in a multi-set $M$ is denoted by $\mu_m(M)$.
- Elements of type $String$ are ordered multi-sets of alphanumeric symbols (characters).
- The type **bool** denotes the set $\{\mathbf{true}, \mathbf{false}\}$

  A box has been drawn around this example to distinguish it from real definitions. Note that a type definition starts with the specification of the types of the entities of which the type is made up (here $x$ is a natural number and $y$ is complex) . After this the domains can possibly be restricted. All the conditions specified (by item) must be satisfied.

## 2.1 Representation of netlists

This section starts with a formal representation of netlists with components with arbitrary interchangeability relations on their terminals. After this an isomorphism relation will be defined on these netlists.

A circuit[1] consists of a set of interconnected nets and components. In this section a circuit will be mapped onto a formal structure referred to as a $Netlist'$. In order to do this the elements which constitute the netlist need to be represented. It is necessary to represent the components, the nets and the interconnections. But it is also necessary to represent additional information associated with the components. For each component type, the parameters and their respective tolerances need to be represented. Furthermore a terminal permutation set needs to be represented for each component type.

Before proceeding, the meaning of each of the terms used will be explained with an example. Figure (2.1) depicts a CMOS inverter. It consists of four *nets* ($In$, $Out$, $Vdd$ and $Gnd$) and two *components*: *T0* and *T1*. Each of these components has terminals which are labelled g,d, and s. These terminals form the connections between the components and the nets and will be represented by edges.

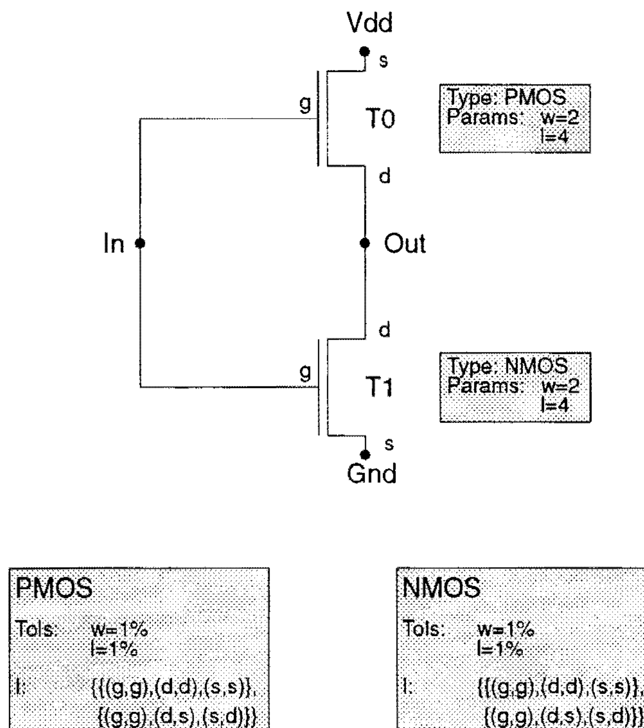



Figure 2.1: Example of a circuit. A CMOS inverter

The shaded boxes on the right-hand side of the components contain information which is part of the

[1] The term *circuit* does not neccessarily refer to a description at the circuit level in the Y-chart of chapter one. It is used here, as is the term *netlist*, to denote a description in the structural domain.

component instantiations. They give information regarding the *types* and the parameter values of the components. *T1* is of type $PMOS$. It has values of 2 and 4 for its $w$ and $l$ parameters respectively. The shaded boxes at the bottom of figure (2.1) describe the component *types*. The type $PMOS$ has two parameters: $w$ and $l$. They both have a tolerance of 1%. Finally each component type has a permutation set $I$ on its terminals associated with it. These sets are given at the bottom of the type descriptions. For both the $PMOS$ and the $NMOS$ the $d$ and the $s$ terminals may be mutually interchanged whereas the $g$ terminal may not be swapped with any other terminal.

**Definition 2.1.1** (*Permutation set*)
A permutation set on a set of terminal names $S$, is a set of mappings $I = \{i_0, i_1, i_2, \ldots, i_{|I|-1}\}$. Where:

- $i_0, i_1, i_2, \ldots, i_{|I|-1}$ are $1-1$ mappings of the terminal names associated with the component type onto the terminal names associated with the component type.

□
The componenttypes $PMOS$ and $NMOS$ in figure (2.1), for example, both have a permutation set $I = \{\{(\mathtt{g},\mathtt{g}),(\mathtt{d},\mathtt{d}),(\mathtt{s},\mathtt{s})\},\{(\mathtt{g},\mathtt{g}),(\mathtt{d},\mathtt{s}),(\mathtt{s},\mathtt{d})\}\}$.

We now proceed with the formal representation of component types. A component type has a name, terminal names, parameter names, a function associating a tolerance with every parameter, a set of valid terminal name permutations, and an optional sub-netlist associated with it. The meaning of this optional sub-netlist will become apparent in a later section (2.2). In the previous example the component types did not have a sub-netlist associated with it.

**Definition 2.1.2** (*ComponentType*)
A $ComponentType$ is a 6-tuple: $(name, terminalNames, I, parameterNames, tol, netlist)$.
Where:

- $name$ is the name of the ComponentType. It is of type $String$
- $terminalNames$ is a set of terminal names. It is of type set of $String$
- $I$ represents the interchangeability of the component terminals. It is a permutation set.
- $parameterNames$ is an ordered set of parameter names. It is of type odered set of $String$
- $tol$ is a function which associates a tolerance with every parameter. It is a function: $tol : String \mapsto I\!R$
- $netlist$ is a sub-netlist associated with the component type. It is of type $Netlist$, which is defined further on (definition 2.2.1).

□
As an example, consider the component type of $T0$ in figure (2.1). It is represented as follows: $PMOS = (\mathtt{PMOS}, \{\mathtt{g},\mathtt{d},\mathtt{s}\}, \{\{(\mathtt{g},\mathtt{g}),(\mathtt{d},\mathtt{d}),(\mathtt{s},\mathtt{s})\},\{(\mathtt{g},\mathtt{g}),(\mathtt{d},\mathtt{s}),(\mathtt{s},\mathtt{d})\}\}, \{\mathtt{w},\mathtt{l}\}, \{(\mathtt{w},0.01), (\mathtt{l},0.01)\}, \emptyset)$

Next we discuss the representation of components. A component has a name, a type, and parameter values.

**Definition 2.1.3** (*Component*)
A $Component$ is a 1-tuple: $(name, type, parameter)$, where:

- $name$ is the name of the component. It is of type $String$.
- $type$ is the type of the component. It is of type $ComponentType$.
- $parameter$ is a function which associates a value with every parameter of the component. It is a function: $parameter : String \mapsto I\!R$. We will also use the notation $parameter_i$ to refer to the value of the $i$'th parameter (counting from zero). Thus for some component, $c$, $parameter_i(c) \equiv parameter([parameterNames(type(c))]_i)$

□
The component $T0$ in figure (2.1) is represented as follows: $T0 = (\mathtt{T0}, PMOS, \{(\mathtt{w}, 2), (\mathtt{l}, 4)\})$

A net only has a name:

**Definition 2.1.4** (*Net*)
A $Net$ is a 1-tuple: $(name)$, where

- $name$ is the name of the net. It is of type $String$.

□
The connections between the components and the nets, i.e. the terminals of the components, are represented using edges:

**Definition 2.1.5** (*Edge*)
An $Edge$ is a pair $\{c, n\}$ , where:

- $c$ is the component with which the edge is incident. It is of type $Component$.
- $n$ is the net with which the edge is incident. It is of type $Net$.

□
A netlist is represented by a bipartite graph $G$ of net vertices and component vertices. With each edge we associate the name of the component terminal to which it is connected.

**Definition 2.1.6** (*Netlist'*)
A $Netlist'$ is derived from a bipartite graph. It is a 5-tuple $G = (C, N, E, name)$, where:

- $C$ is a set of components.
- $N$ is a set of nets.
- $E$ is set of edges connecting the components and the nets.
- $name$ is a function which associates a name with every edge:
$name : E \longrightarrow String$

□
Thus we have defined a complete representation of netlists. Figure (2.2) shows the complete formal representation of the circuit depicted in figure (2.1).
Next, we shall define an isomorphism relation on netlists. Isomorphic netlists must have equal structures. Since netlists have been represented as bipartite graphs, a netlist isomorphism should clearly

be a graph isomorphism. Additionally we require that the types of the components mapped onto each other correspond and that the connections to the terminals obey the interchangeability which is defined for each component type.

**Definition 2.1.7** (*Netlist isomorphism*)
Let $G$ and $H$ be netlists as in definition (2.1.6). A *netlist isomorphism*, $\Theta'(G, H)$, of $G$ onto $H$, is a pair $(f, g)$, where:

1. $f$ is a $1-1$ mapping of $C(G)$ onto $C(H)$
2. $g$ is a $1-1$ mapping of $N(G)$ onto $N(H)$
3. $(\forall c, n : c \in C(G) \wedge n \in N(G) \wedge \{c, n\} \in E(G) : \{f(c), g(n)\} \in E(H))$
4. $(\forall c : c \in C(G) : type(c) = type(f(c))$
5. $M(c) \equiv (\exists i\text{: } i \in I(type(c)) :$
   $(\forall n : \{c, n\} \in E(G) : i(name(\{c, n\})) = name(\{f(c), g(n)\}))$
   $)$

   $(\forall c : c \in C(G) : M(c))$

□

The first three items specify the 'normal' graph isomorphism for bipartite graphs. The fourth item specifies that the component types of components mapped onto each other should match. Finally, the last item specifies that the connections to the terminals of the components should obey the interchangeability rules of the component type. That is, for each component the mapping of the terminal names in the test netlist onto the terminal names in the reference netlist must be a member of the permutation set for that component. The mapping of the terminal names is implied by the mapping of the terminals (edges in the graph).

Finally we define an isomorphism which takes the parameter values of the components into account. Such an isomorphism is referred to as a *parameterically correct netlist isomorphism*. It is a simply *netlist isomorphism* with the additional property that the parameter values of the components of the test netlist must lie within the tolerance margins of the components of the reference netlist.

**Definition 2.1.8** (*Parameterically correct netlist isomorphism*)
Let $G$ and $H$ be netlists as in definition (2.1.6). A parameterically correct netlist isomorphism, $\Gamma'(G, H)$, of $G$ onto $H$, is a pair $(f, g)$ where:

- $\Gamma'(G, H)$ is a netlist isomorphism as in definition (2.1.7)
- $(\forall c : c \in C(G) :(\forall p\text{: } p \in parameterNames(type(c)) :$
  $|parameter(p(c)) - parameter(p(f(c))| \leq tol(p)))$
  $)$
  $)$

Note that for two given netlists, $G$ and $H$, the existence of a parameterically correct netlist isomorphism of $G$ onto $H$ does not imply the existence of a parameterically correct netlist isomorphism of $H$ onto $G$ since the tolerances defined for the component types may be different in the two netlists.
□

Permutation sets:
$I_0 = \{\{(\mathtt{g},\mathtt{g}),(\mathtt{d},\mathtt{d}),(\mathtt{s},\mathtt{s})\},\{(\mathtt{g},\mathtt{g}),(\mathtt{d},\mathtt{s}),(\mathtt{s},\mathtt{d})\}\}$
$I_1 = \{\{(\mathtt{g},\mathtt{g}),(\mathtt{d},\mathtt{d}),(\mathtt{s},\mathtt{s})\},\{(\mathtt{g},\mathtt{g}),(\mathtt{d},\mathtt{s}),(\mathtt{s},\mathtt{d})\}\}$
Component types:
$PMOS = (\mathtt{PMOS},\{\mathtt{g},\mathtt{d},\mathtt{s}\},I_0,\{\mathtt{w},\mathtt{l}\},\{(\mathtt{w},0.01),(\mathtt{l},0.01)\},\emptyset)$
$NMOS = (\mathtt{NMOS},\{\mathtt{g},\mathtt{d},\mathtt{s}\},I_1,\{\mathtt{w},\mathtt{l}\},\{(\mathtt{w},0.01),(\mathtt{l},0.01)\},\emptyset)$
Nets:
$Gnd = (\mathtt{Gnd})$
$Vdd = (\mathtt{Vdd})$
$In = (\mathtt{In})$
$Out = (\mathtt{Out})$
$N = \{Gnd, Vdd, In, Out\}$
Components:
$T0 = (\mathtt{T0}, PMOS, \{(\mathtt{w},2),(\mathtt{l},4)\})$
$T1 = (\mathtt{T1}, NMOS, \{(\mathtt{w},2),(\mathtt{l},4)\})$
$C = \{T0, T1\}$
Interconnections:
$e_0 = \{In, T0\}$
$e_1 = \{Vdd, T0\}$
$e_2 = \{Out, T0\}$
$e_3 = \{In, T1\}$
$e_4 = \{Gnd, T1\}$
$e_5 = \{Out, T1\}$
Terminal names associated with interconnections:
$name = \{(e_0,\mathtt{g}),(e_1,\mathtt{s}),(e_2,\mathtt{d}),(e_3,\mathtt{g}),(e_4,\mathtt{s}),(e_5,\mathtt{d})\}$
$E = \{e_0,e_1,e_2,e_3,e_4,e_5\}$
The circuit represented as a $Netlist'$:
$G = (C, N, E, name)$

Figure 2.2: Formal representation of circuit in figure 2.1 (The CMOS inverter).

## 2.2 Recursive interchangeability relations

It is not possible to construct an isomorphism mapping between two netlists with components with arbitrary terminal permutation sets, such as those described in section 2.1, using the refinement algorithm. This section describes a transformation of these netlists onto netlists for which an isomorphism mapping *can* be constructed using the refinement algorithm. The transformation and the isomorphism condition in the co-domain are defined in such a way that two netlists in the original domain are isomorphic if and only if their images in the co-domain are isomorphic. The transformation can only be used for netlists with components with permutation sets that can be represented by means of the permutation tree which is defined further on.

The aim is to transform the netlists into netlists with components with simple terminal pin permutation sets, such as the netlists at the circuit level mentioned in the introduction. Before discussing the transformation steps, the destination netlist is described first.
The resulting netlist consists of components with permutation sets which can be represented by assigning *terminal classes* to the terminals. These terminal classes are assigned in such a manner that terminals having equal terminal classes may always be interchanged. In the formal representation of the netlists, edges are used to represent the terminals of the components. Thus associating a terminal class with each terminal means associating a terminal class with each edge of the netlist. Therefore definition (2.1.6) of $Netlist'$ must be extended with a terminal class function:

**Definition 2.2.1** (*Netlist*)
A $Netlist$ is a 4-tuple $G = (C, N, E, tc)$, where:

- $C$,$N$, and $E$ have the same meaning as in definition 2.1.6
- $tc$ is a function which associates a terminal class with every edge:
  $tc : E \mapsto \mathbb{N}$

□
For these netlists the terminal permutation sets are described completely by the terminal classes, so that the netlist isomorphism condition can be written in a simpler form. Because for each component terminals having the same terminal class may be interchanged, condition 5 of the $Netlist'$ isomorphism condition in definition (2.1.7), in the domain of $Netlist$ becomes:

$$(\forall c, n : c \in C(G) \wedge n \in N(G) \wedge \{c, n\} \in E(G) : tc(\{c, n\}) = tc(\{f(c), g(n)\})$$

Hence, the definition of a $Netlist$ isomorphism becomes:

**Definition 2.2.2** (*Netlist isomorphism*)
Let $G$ and $H$ be netlists as in definition (2.2.1). A *netlist isomorphism*, $\Theta(G, H)$, of $G$ onto $H$, is a pair $(f, g)$, where:

1. $f$ is a $1-1$ mapping of $C(G)$ onto $C(H)$
2. $g$ is a $1-1$ mapping of $N(G)$ onto $N(H)$
3. $(\forall c, n : c \in C(G) \wedge n \in N(G) \wedge \{c, n\} \in E(G) : \{f(c), g(n)\} \in E(H))$
4. $(\forall c : c \in C(G) : type(c) = type(f(c))$

5. $(\forall c, n : \{c, n\} \in E(G) : tc(\{c, n\}) = tc(\{f(c), g(n)\})$

□

The transformation from the original netlist to a netlist of type *Netlist* is performed in three steps. The first step involves describing the permutation set of the terminals of the component types in the form of trees. The second step consists of transforming these trees into a form suitable for insertion into the netlists, in doing so a sub-netlist (of type *Netlist*) is associated with every component type in the domain of *Netlist'*. The third step involves the actual transformation of the netlists, it is done by inserting the sub-netlists into the netlist. These three steps are described in the following sections.

### 2.2.1 Permutation trees

In general, the interchangeability of a set of terminals is represented by associating a name with each terminal, and writing a permutation set $I$, which represents the set of valid mappings, in the names of these terminals. Given a set of terminals names, $S$, and a permutation set $I$ as in definition (2.1.1), a $1-1$ mapping of the terminals onto the terminals, $y$ is a valid mapping if, and only if: $y \in I$.

To represent the subset of recursive interchangeability relations, the terminal names are mapped onto the leaves of a permutation tree by a $1-1$ mapping, $\gamma$. By definition a permutation of the terminal names in $S$ is a valid permutation if and only if a mapping of the corresponding leaves is a valid mapping. The structure of the tree then defines valid permutations of the terminals. This is so, because the structure of the tree determines the set of valid $1-1$ mapping of its leaves (and thereby the corresponding terminal names) onto its leaves in a sense to be defined.
First, the permutation tree is introduced.

**Definition 2.2.3** (*Permutation tree*)
An permutation tree for a given component with a set of terminals $S$, is a tree $T(I) = (V = L \cup N, E, tc')$, where:

1. $T$ is a tree
2. $L$ is a set of leaf vertices
3. $N$ is a set of vertices.
4. $E$ is a set of directed edges from the parents to their children.
5. $tc'$ is a function which associates a terminal class with every edge:
$$tc' : E \mapsto I\!N. \quad (2.2.1)$$
6. sub-trees of a vertex that are connected to it with edges with equal terminal classes, are isomorphic.

□

An example of a permutation tree with a mapping, $\gamma$, of the terminal names onto the leaves is shown in figure (2.3). The mapping $\gamma$ is shown in this figure by simply writing the terminal names under the leaves onto which they are mapped.
A valid $1-1$ mapping of the vertices of a permutation tree (which naturally includes its leaves) onto the vertices of the same permutation tree, is referred to as a permutation tree automorphism. It is defined in the following:

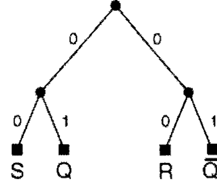



Figure 2.3: A permutation tree with mapped terminal names

**Definition 2.2.4** (*Permutation tree automorphism*)
For a given permutation tree, $T$, a permutation tree automorphism, $z$, is a mapping, where:

1. $z : V(T) \mapsto V(T)$
2. $z$ is a $1 - 1$ mapping.
3. Any two vertices that share the same parent can be mapped onto each other if and only if the terminal classes of the edges connecting them to their parent are equal:
$$(\forall p : p \in N : \quad (\forall c : (p, c) \in E : tc'((z(p), z(c))) = tc'((p, c))) \quad ) \qquad (2.2.2)$$
4. If two vertices are mapped onto each other, their respective children are also mapped onto each other. In other words, adjacency is preserved:
$$(\forall p : p \in N : \quad (\forall n : (p, c) \in E : (z(p), z(c)) \in E) \quad ) \qquad (2.2.3)$$

□

An example of a permutation tree automorphism of the tree of figure (2.3) is shown in figure (2.4). We see that sets of terminal names that are unconditionally permutable are connected to a common parent with edges with equal terminal classes, while terminal names which are fixed relative to each other are connected to a common parent but with different terminal classes. This holds recursively.

### 2.2.2 Mapping of a permutation tree onto a sub-netlist

The aim is to replace every component by its respective permutation tree, so that an isomorphism in the domain of *Netlist* is equivalent with an isomorphism in the domain of *Netlist'*. It is not possible to insert the trees which have been described directly because it does not consist of components and nets.

Here a mapping of isomorphism trees onto netlists is described. This means that the permutation sets of the component types can be described by associating a *netlist* with every type. A provision for this has been made in definition (2.1.2) of the component type.
The mapping is constructed in such a way that if, and only if, a given mapping of the vertices of the permutation tree onto the vertices of the permutation tree is an automorphism mapping, the mapping

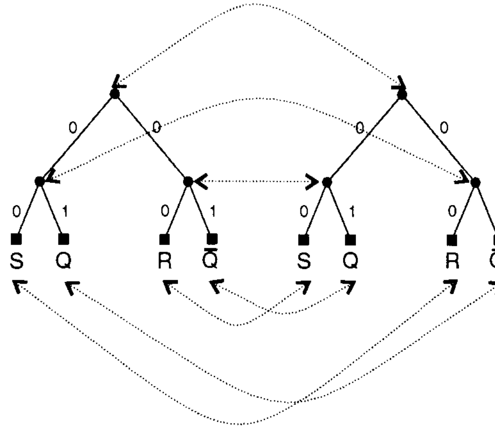



Figure 2.4: Example of a permutation tree automorphism

of the corresponding components and nets in the netlist is a netlist isomorphism.

**Definition 2.2.5** (*Mapping of a permutation tree onto a netlist*)
Let $T$ be a permutation tree as in definition (2.2.3). Let $G^*$ be a netlist as in definition (2.2.1). A mapping of a permutation tree onto a netlist is a 4-tuple $M(T) = (v_c, v_n, e, t)$, where:

- $v_c$ is a $1-1$ mapping of the vertices of the tree onto the components of the netlist: $v_c : V(T) \mapsto C(G^*)$, where the type of the component is set to $\emptyset$.
- $v_n$ is a $1-1$ mapping of the edges of the tree onto the nets of the netlist: $v_n : E(T) \mapsto N(G^*)$
- $e$ is a mapping of the edges of the tree onto unordered pairs of edges of the netlist: $e : E(T) \mapsto \mathbf{setOf}(E(G^*))$. Let $x \in V(T), y \in V(T), (x, y) \in E(T)$ $e((x,y)) = \{\{v_c(x), v_n((x,y))\}, \{v_c(y), v_n((x,y))\}\}$
- $t$ is mapping of the terminal class function, $tc'$, of the tree onto the terminal class function, $tc$, of the netlist. It is constructed such that the terminal classes associated with the edges of the tree are equal to the terminal classes associated with their 'edge' images in the netlist:
$(\forall : x \in E(T) : (\forall y : y \in e(x) : tc(y) = tc'(x)))$ ( 2.2.4)

□

An example of a mapping of a permutation tree onto a netlist is shown in figure (2.5). Here the mapping of one vertex and one edge is shown. The mapping of the vertex onto a component, $v_c$, is represented by the dotted line. The mapping, $v_n$, of the edge onto a net is shown by the dashed/dotted line. The mapping, $e$, of the edge onto edges is indicated by the dashed lines.

Using the functions $T$ and $M$ as defined in the previous, the association between a component type, $tp$, and and its sub-netlist $netlist(tp)$ may now formally be written as: $netlist(tp) = M(T(I(tp)))$

( 2.2.5)

⋮

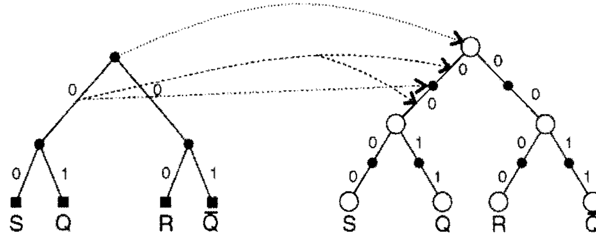



Figure 2.5: Mapping of the permutation tree of the S-R flip-flop onto a netlist

### 2.2.3 Insertion mapping

The actual transformation from a netlist, $netlist'$, of type $Netlist'$ onto a netlist, $netlist$, of type $Netlist$ is done by mapping every component $c$ of $netlist'$ onto a netlist $\alpha(c) \subseteq netlist$. The mapping is constructed such that two netlists of type $Netlist'$ are isomorphic in their domain if and only if their images are isomorphic in the domain of $Netlist$.

**Definition 2.2.6** (*Mapping of a Netlist' onto a Netlist*)
Let $G$ be a netlist as in definition (2.1.6), and $H$ a netlist as in definition (2.2.1). A Mapping of a $Netlist'$ onto a $Netlist$ which preserves isomorphism is a triple $(\alpha, \beta, \delta)$, where:

1. $\alpha$ maps each component $c$ onto a netlist that corresponds to its type. The components of this netlist have the same type and parameter values as the original component $c$.
   $\alpha : C(G) \mapsto Netlist$
   Let $netlist(type(c)) = (C, N, E, tc)$
   $\alpha = ((\cup c' : c' \in C : (name(c'), type(c), parameter(c))\ ), N, E, tc)$

2. $\beta$ is a mapping of the nets of $Netlist'$ onto the nets of $Netlist$:
   $\beta : N(G) \mapsto N(H)$
   $\beta(n) = n$

3. $\delta$ is a mapping of the edges of $Netlist'$ onto corresponding edges of $Netlist$ in such a way that each edge representing a terminal, connected to some net in $Netlist'$ is mapped onto an edge which connects the component representing this terminal in $Netlist$ to the same net.
   $\delta : E(G) \mapsto E(H)$
   $\delta(\{c, n\}) = (v_c(\gamma(name\{c, n\})\ ), n)$

□

# Chapter 3

# Construction of an isomorphism mapping

The two netlists which need to be compared with each other can be represented as labeled bipartite graphs as was discussed in chapter two. To determine whether the two netlists are equivalent, it is attempted to construct an isomorphism mapping from the reference graph onto the test graph. If such a mapping can be constructed, the two netlists are isomorphic. If the two netlists are not isomorphic, we want an as accurate as possible indication of the discrepancies.

The construction of an isomorphism mapping is an open problem, i.e. it is not known whether it is in $\mathcal{P}$, or whether it is $\mathcal{NP}$-complete [2]. However, from a practical point of view the refinement algorithm [1] is known to perform well for the purpose of netlist comparison [6, 5, 4, 3, 14].

This refinement algorithm for determining graph isomorphism mappings is discussed in the following sections. For illustrative purposes the algorithm is presented in its basic form suitable for unlabelled graphs first.

Apart from the fact that this algorithm does not take component types, terminal classes, and parameter values into account it has a number of shortcomings which make it unsuitable for netlist comparison. These shortcomings include the fact that discrepancies between the reference and test graphs are not located. Furthermore the algorithm can terminate prematurely. That is, for two given isomorphic graphs, it might construct an incomplete mapping. The algorithm and the associated problems will be discussed in section (3.1). This section forms the basis for the discussion of a more complete algorithm in section (3.2).

## 3.1 The refinement algorithm

This section discusses an algorithm for determining graph isomorphism mappings, or graph isomorphisms. The algorithm is analogous to the one presented in [1], and is referred to as the *iterative partitioning, iterative refinement* or simply *refinement* algorithm.

First the algorithm is discussed informally with the aid of an example. After this, the algorithm is treated formally. The algorithm is presented in such a manner that it can easily be adopted for the purpose of netlist comparison.

### 3.1.1 An informal discussion

The aim is to find a 1-1 mapping of the vertices of the graph in the left-hand side of figure (3.1.a) (named for convenience the *reference* graph) onto the vertices of the graph in the right-hand side of the same figure (named for convenience the *test* graph).

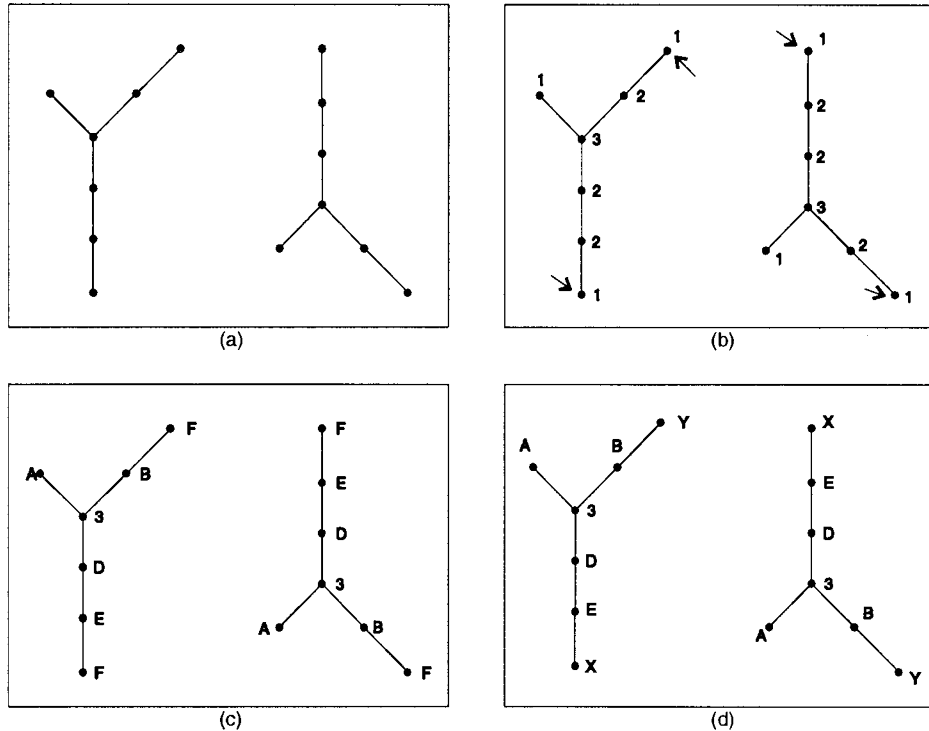



Figure 3.1: Stages in the refinement process

The basis of this algorithm is that of iteratively partitioning the sets of vertices of the graphs. Once the partitions contain pairs of vertices, one from the reference and one from the test graph, these vertices can be mapped onto each other. If at some stage there are cells that contain unequal number of vertices from the two graphs, we know that the two graphs are not isomorphic and the algorithm terminates.

First an initial partitioning is made which is based on the degrees of the vertices. The initial partitioning of the example graphs in figure (3.1.a) is shown in figure (3.1.b). The labels shown in the figure are referred to as signatures. Vertices with equal signatures are members of the same cell. Thus all the vertices labeled '1' in figure (3.1.b) belong to the same cell, with signature '1'. In this example we see that the cell with signature '3' contains two vertices; one from the reference graph and one from the test graph. These vertices can therefore be mapped onto each other.

After the initial partitioning step, the sets of vertices are iteratively partitioned and possibly mapped until all the vertices in the reference graph have been mapped onto vertices in the test graph. As a partitioning criterion the signature of each vertex and its direct neighbors is used. In one partitioning step all vertices are considered. Vertices which have the same signature and the same multi-set of signatures of neighboring vertices are placed together in a new cell. Each new cell which contains

more than one vertex from each graph is assigned a new unique signature which is to be used in the next partitioning step.
As an example consider the partitioning step made from figure (3.1.b) to (3.1.c). The vertices in figure (3.1.b) with an arrow pointing to it, have a signature '1' and a multi-set of neighbor signatures equal to $\underline{\{2\}}$. These vertices together with the other vertices which have a signature of '1' and a multi-set of neighbor signatures equal to $\underline{\{2\}}$ are put together in a new cell as depicted in figure (3.1.c). They are assigned a new unique signature 'F'. The same procedure has been applied to the other vertices.
All the vertices, except those labeled 'F' can be mapped. One more partitioning step yields the partitioning of figure (3.1.d).

### Static partitioning

In the previous example, the graphs could be partitioned into pairs of single vertices, making it possible to construct an isomorphism mapping. Consider, however, the graphs of figure (3.2). The sets of vertices in these graphs cannot be partitioned any further using the refinement algorithm. Hence we are left with a cell with two vertices from each graph (those labeled 'A'). Such a partitioning will be referred to as a *static* partitioning. It will *always* occur when comparing two graphs which have more than one automorphism mapping.

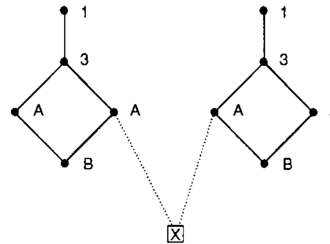
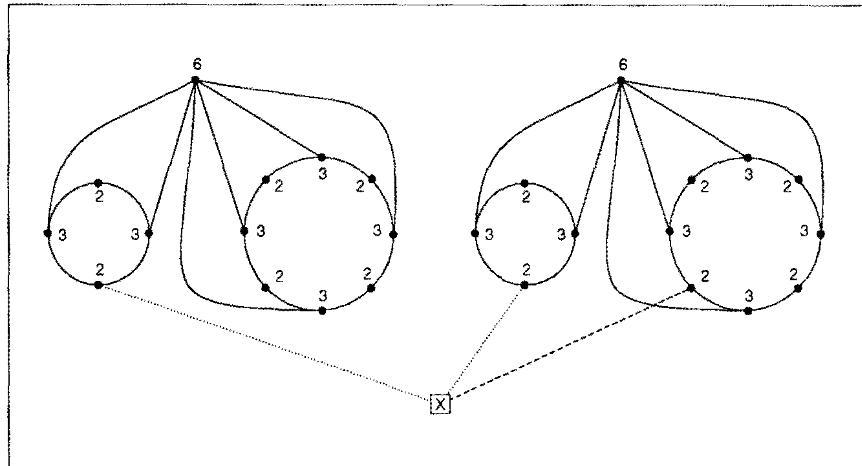



Figure 3.2: A 'static' partitioning

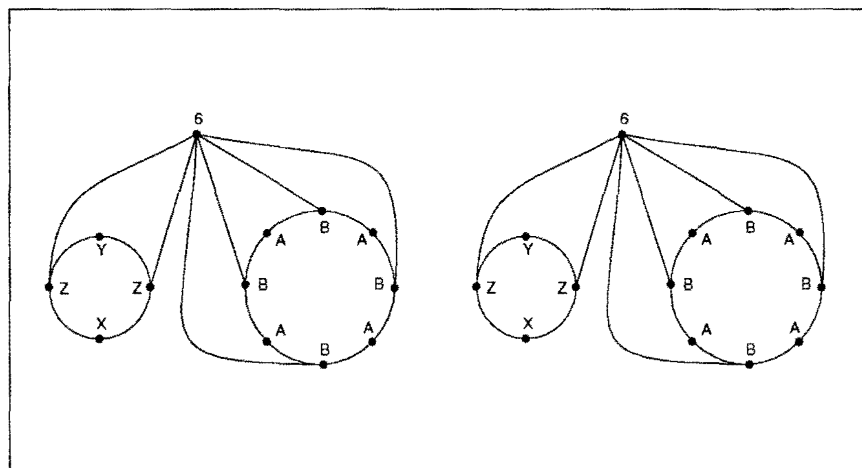
It has been conjured [1] that when constructing an isomorphism mapping between two isomorphic graphs, $G$ and $H$, if the refinement leads to a static partitioning, one may restart the partitioning process by assigning a unique signature, $s$, to two vertices from a single cell: one from graph $G$ and one from graph $H$.
In our example this corresponds to assigning a signature, say 'X', to one of the vertices with a signature '2' in the left hand graph, and to a vertex with a signature '2' in the right hand graph. This indeed leads to an isomorphism mapping.

It can easily be shown, with the aid of a contradiction, that the conjecture does not hold in the general case however. One such contradiction is illustrated by means of the graphs in figure (3.3). In figure (3.3.a), again we have a static partitioning. Assigning a new signature 'X' to the vertices pointed to by the dotted lines, and restarting the refinement process leads to the partitioning shown in figure (3.3.b). We see that the vertices with labels 'Y' can be mapped onto each other and that another static partitioning is reached (again requiring a 'forced' unique signature). Nothing has gone wrong here.
If, however, the new signature 'X' is assigned to the vertex pointed to by the dashed line in the

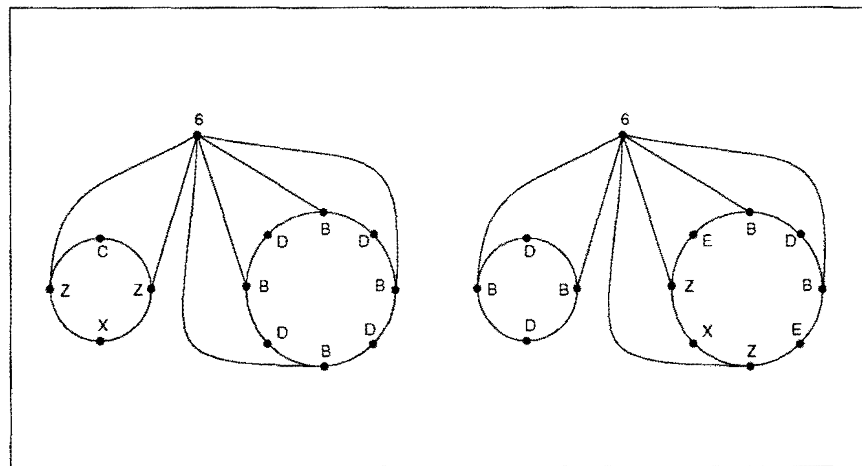
right hand graph, and the vertex pointed to by the dotted line in the left hand graph of figure (3.3.a), clearly a mistake is made because the mapping of these 'X' vertices onto each other is *not* part of an isomorphism mapping. This becomes apparent after two more partitioning steps, resulting in the partitioning shown in figure (3.3.c). Because at least one cell contains unequal numbers of vertices from the reference and the test graph (the cell with signature 'C' for example), the algorithm will terminate and erroneously conclude that the two graphs are not isomorphic.




(a)




(b)




(c)

Figure 3.3: A 'static' partitioning with an ambiguity

**Discrepancies**

In the previous examples it was attempted to construct isomorphism mappings between isomorphic graphs. Here we shall see the effect of the refinement algorithm when comparing two non-isomorphic graphs. Consider the graphs of figure (3.4.a). The vertices of these graphs have been assigned their initial signatures. Since the initial partitioning contains a cell with unequal number of vertices from the two graphs (the cell with vertices with signature '1' for example), the refinement algorithm terminates immediately, concluding that the two graphs are not isomorphic.

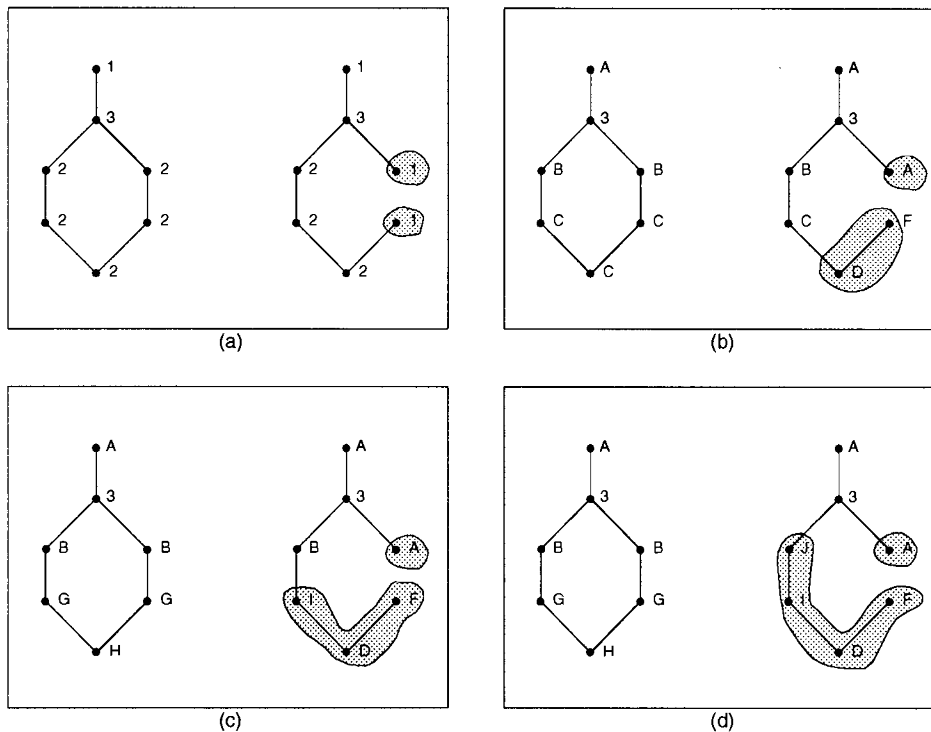



Figure 3.4: Stages in the refinement process for two non-isomorphic graphs

The aim of netlist comparison, however, is to locate the discrepancies. This means that as many as possible vertices must be mapped onto each other in such a way that the mapping resembles an isomorphism mapping 'as closely as possible'. We shall examine here what happens if the refinement process is continued. First of all, the vertices with the signature '3' will be mapped onto each other. The following partitions are shown in figures (3.4.b-d). We see that in figure (a) two vertices have been marked with a blob ink. These two vertices have different signatures from the vertices onto which they should be mapped (a '1' instead of a '2'). Because they have different signatures, however, they will never be mapped. All the other vertices in figure (a) are still potential candidates. One more partitioning step gives the partitioning shown in figure (b). The bottom blob of ink in this figure has become larger. The reason for this is that another vertex in the right hand graph (the one with signature 'D') has been assigned a signature which is different from its corresponding vertex in the left graph. Thus we see that the signature discrepancies *propagate* through the graph. The final

partitioning in figure (d) has been corrupted to the stage that no more vertices can be mapped. Note that the top blob of ink does not expand, because the error propagation is limited by the vertex (with signature '3' ) that has been mapped.
The error propagation which was illustrated here with the aid of an example, happens in general for two non-isomorphic graphs. Because in each refinement stage information is transferred from vertices to their neighbors. It can be concluded that merely continuing the refinement process is not a useful method for locating discrepancies.

### 3.1.2 A formal discussion

Before proceeding with a discussion of the actual functions, definitions of a cell and a vertex partitioning are given. These definitions are suited to the algorithm used for testing graph isomorphism and deviate somewhat from the traditional definitions of partitionings and cells.

**Definition 3.1.1** (*Cell*)
Let $G$ and $H$ be graphs, and let $signature$ be a mapping which associates a signature with the vertices of these graphs: $signature : V(G) \mapsto Signature$. A $Cell(G, H)$ is an ordered pair $(x, y)$, where:

- $x$ is a set of vertices in $G$. $x \subseteq V(G)$
- $y$ is a set of vertices in $H$. $y \subseteq V(H)$
- All the vertices in a cell have the same signature. $(\forall a, b : (a, b) \in (x \cup y, x \cup y) : signature(a) = signature(b))$

The sets of vertices $x$ and $y$ are referred to as cell-halves. □

**Definition 3.1.2** (*Vertex partitioning*)
Let $G$ and $H$ be graphs. A vertex partitioning of these graphs, $\Pi(G, H)$ is a set of $Cell$, $\{c_0, c_2, \ldots, c_{|\Pi|-1}\}$,where for all instances of $\Pi$, $\pi$:

- $(\cup c : c \in \pi : [c]_0) = V(G)$
- $(\cup c : c \in \pi : [c]_1) = V(H)$
- $(\forall c, c' : (c, c') \in (\pi, \pi) : signature(x : x \in c) \neq signature(x' : x' \in c') \Leftrightarrow c \neq c')$

□
Since for a given partitioning, $\pi$, each cell has its own, unique, signature it is possible to identify the cells of a partitioning by means of their signature. Thus a function is introduced which associates a cell with a signature for a given partitioning:
$cell : \Pi \times Signature \mapsto Cell$ ( 3.1.1)
$cell(\pi, s) = c : c \in \pi \wedge signature(c) = s$ ( 3.1.2)
For some vertex, $x$, we sometimes use $cell(\pi, x)$ as shorthand for $cell(\pi, signature(x))$.

To create an isomorphism mapping of two graphs onto each other, first an initial partitioning is created with a function:
$partition_0 : Graph \times Graph \mapsto \Pi$ ( 3.1.3)
The actual construction of a (partial) mapping from a given partition is performed by a function:

$map : \Pi(Graph, Graph) \mapsto V(Graph) \times V(Graph)$ (3.1.4)

The iterative partitioning is performed by repeated application of a function:

$partition : Graph \times Graph \times \Pi \mapsto \Pi$ (3.1.5)

These three functions will be discussed in the following. After this a simple loop is presented which tries to construct an isomorphism mapping between two given graphs.

### Initial vertex partitioning

The initial vertex partitioning is the union of the cells over all vertices of the graphs $G$ and $H$, where each cell has vertices with the same initial signature. The cells consist of two cell-halves which contain vertices from the graphs $G$ and $H$ respectively.

$$partition_0(\pi(G,H)) = (\cup x : x \in (V(G) \cup V(H)) : (initialCellHalf(G,x), initialCellHalf(H,x)) ) \quad (3.1.6)$$

Initial cell half, creates the cell part to which a given vertex $x$ in graph $G$ belongs.

$initialCellHalf(G,x) = (\cup y : y \in V(G) : initialSig(x) = initialSig(y))$ (3.1.7)

The initial signature is simply the degree of the vertex.

$initialSig(x) = degree(x)$ (3.1.8)

The following algorithm constructs an initial partitioning as in definition (3.1.6) for two given graphs.

```
|[var G, H : Graph|                                        ( 3.1.9)
{G and H are graphs.}

0.   π = ∅;
1.   for x : x ∈ V(G) → s := degree(x);
2.                      signature(x) := s;
3.                      [cell(π, s)]0 := [cell(π, s)]0 ∪ {x}
4.   rof ;
5.   for x : x ∈ V(H) → s := degree(x);
6.                      signature(x) := s;
7.                      [cell(π, s)]1 := [cell(π, s)]1 ∪ {x}
8.   rof

{π = partition0(G, H)}
|]
```

In line (0) of program (3.1.9) the partitioning $\pi$ is initialized. In lines (1-4) a loop is made over all the vertices in graph $G$. For each vertex $x$ in graph $G$, the appropriate signature is set (the initial signature which equals the degree of the vertices) and the vertex is added to the appropriate cell half. In lines (5-8) the same is done for the vertices in graph $H$.

### Construction of a partial mapping

To construct a partial mapping from a given partitioning $\pi$, the vertex pairs from the cells containing two vertices (one in each cell-half) are mapped onto each other.

$map(\pi) = (\cup c : c \in \pi \wedge |[c]_0| = 1 \wedge |[c]_1| = 1 : (x : x \in [c]_0, x : x \in [c]_1))$ (3.1.10)

Next, an algorithm is presented. Besides constructing a partial mapping this algorithm also detects non-isomorphism by checking whether for each cell, the cell-halves contain equal numbers of vertices. Furthermore vertices that have been mapped are removed from the sets of vertices that are

refined, by placing them in sets of so-called *unique* vertices for each graph, $U(G)$ and $U(H)$. This is done because there is no point in assigning new signatures to these vertices. It has the additional advantage of limiting the error propagation, as was mentioned on page (29).

$$[|\mathbf{var}\ \pi : \Pi, f : V(G) \mapsto V(H), U(G), U(H) : V(G), notIsomorphic : \mathbf{bool}| \quad (3.1.11)$$

```
0.   notIsomorphic := false ;
1.   for x : x ∈ π → if |[x]_0| ≠ |[x]_1| → notIsomorphic := true
2.                   [] |[x]_0| = |[x]_1| →if |[x]_0| = 1 → f(x' : x' ∈ [x]_0) := y' : y' ∈ [x]_1;
3.                                          U(G) := U(G) ∪ x' : x' ∈ [x]_0;
4.                                          U(H) := U(H) ∪ x' : x' ∈ [x]_1;
5.                                         [] |[x]_0| ≠ 1 → skip
6.                                         fi
7.                   fi
8.   rof
```

$\{(f = map(\pi) \vee notIsomorphic)\}$
|]

In lines (1-8) of program (3.1.11) we loop over all the cells, $x$, of the partitioning. For each cell it is checked whether the number of vertices in the two halves (corresponding to the vertices from $G$ and $H$) are equal. If this is not true, it is noted by setting $notIsomorphic$. If both the cell halves contain one vertex, they are mapped onto each other in line (2). Furthermore these vertices are added to the sets of unique vertices, $U(G)$ and $U(H)$. The vertices in these sets will be excluded from the refinement process as will be described in the next section. After the procedure has completed, either a partial mapping has been constructed, or the flag $notIsomorphic$ has been set.

### Further vertex partitioning

A new vertex partitioning is the union of the cells over all vertices of the graphs $G$ and $H$, where each cell has either vertices with the same new signature or vertices with the same old signature. Vertices which are not unique must have the same new signature. The cells consist of two cell-halves which contain vertices from the graphs $G$ and $H$ respectively.

$$\begin{aligned} partition(G, H, \pi) = &(\cup x : x \in (V(G) \cup V(H)) \wedge x \notin (U(G) \cup U(H)) : \\ &\quad (newCellHalf(G, \pi, x), newCellHalf(H, \pi, x)) \\ &)\cup \\ &(\cup x : x \in (V(G) \cup V(H)) \wedge x \in (U(G) \cup U(H)) : \\ &\quad cell(\pi, x) \\ &) \end{aligned} \quad (3.1.12)$$

New cell half, creates the cell part to which a given vertex $x$ in graph $G$ belongs.

$$newCellHalf : Graph \times \Pi \times V(Graph) \mapsto \Pi \quad (3.1.13)$$

$$newCellHalf(G, \pi, x) = (\cup y : y \in V(G) : newSig(G, \pi, x) = newSig(G, \pi, y)) \quad (3.1.14)$$

The new signature is a pair consisting of the current cell, and the multi-set of cells of the neighboring vertices.

$$newSig(G, \pi, x) = ((cell(\pi, x), (\underline{\cup} y : \{x, y\} \in E(G) : cell(\pi, y))) \quad (3.1.15)$$

The following algorithm constructs a partitioning as defined by (3.1.12). It is similar to program (3.1.9).

$$|[\mathbf{var}\ G, H : Graph, \pi : \Pi| \quad (3.1.16)$$

$\{G$ and $H$ are graphs and $\pi$ is a vertex partitioning on these graphs $\}$

```
0.  π' := π;
1.  for x : x ∈ V(G) \ U(G) → s := newSig(G, π, x);
2.                             signature(x) := s;
3.                             [cell(π, s)]_0 := [cell(π, s)]_0 ∪ {x}
4.  rof ;
5.  for x : x ∈ V(H) \ U(H) → s := newSig(H, π, x);
6.                             signature(x) := s;
7.                             [cell(π, s)]_1 := [cell(π, s)]_1 ∪ {x}
8.  rof
```

$\{\pi = partition(G, H, \pi')\}$
$]|$

The code of program (3.1.16) is almost identical to that of the program which calculates the initial partitioning (3.1.9). The only difference is that a different signaturing function is used.

### A loop which tries to construct an isomorphism mapping

The following loop, uses the procedures $partition_0$, $map$, and $partition$ to construct an isomorphism mapping. It does not force any vertices once a 'static' partitioning is reached. Rather, it terminates. The variables $G, H, \pi, f$, and $notIsomorphic$ which were, strictly speaking, local variables in the algorithms presented are assumed to be global variables here for sake of simplicity.

$$[[\mathbf{var}\ G, H : Graph, U(G) : V(G), U(H) : V(H), \pi, \pi^* : \Pi, f : V(G) \mapsto V(H), \quad (3.1.17)$$
$$notIsomorphic : \mathbf{bool}|$$

```
0.  f := ∅
1.  notIsomorphic := false
2.  U(G) := ∅
3.  U(H) := ∅
4.  π := ∅
5.  partition_0
6.  map
7.  π* := ∅
8.  do π ≠ π* ∧ ¬notIsomorphic →
9.              π* := π
10.             partition
11.             map
12. od
```

$\{notIsomorphic \vee (U(G) = V(G) \wedge$ 'f is an isomorphism mapping'$) \vee$
$(U(G) \neq V(G))\}$
$]|$

In lines (0-4) of program (3.1.17) initialization takes place. In line (5) an initial partitioning is created which is used by the function $map$ in the next line. The loop of lines (8-12) terminates once it is noticed that a *static* partitioning is reached or when it is noticed that the graphs $G$ and $H$ are not isomorphic. Within the loop the partitioning is partitioned and it is attempted to map new vertices. Note that static partitionings are detected here simply by using a dummy partitioning $\pi^*$.

## 3.2 Netlist comparison

This section describes a netlist comparison program. In the previous section the refinement algorithm for the determination of graph isomorphisms was discussed. As mentioned, this algorithm in its pure form is not suitable for netlist comparison. In order to overcome some of the deficiencies of the refinement algorithm heuristics are used. Some others can be solved in a deterministic fashion. A large number of the heuristics are based on those used in the Philips version of Gemini which was developed by D.W. Harberts [6, 5]. This program is an improved version of the original program Gemini [4], which was developed at Carnegie Mellon University. When referring to Gemini, in the following, the version by D.W Harberts is meant.
The problems associated with the 'bare' refinement algorithm are summarized here:

- It can terminate without constructing a complete isomorphism mapping even though the graphs which are compared with each other are isomorphic. This occurs when the graphs have more than one automorphism mapping or when ambiguous static partitionings are reached.
- It does not take component types and terminal classes into account.
- It does not take parameter values into account.
- It does not locate discrepancies.

The general idea behind the solutions to these problems will be discussed in the following four sections. The actual algorithms and functions will be discussed in section 3.2.5.

### 3.2.1 Termination

The fact that the refinement algorithm can terminate without constructing a complete isomorphism mapping is a fundamental deficiency. In section (3.1.1) two cases with static partitionings were distinguished; a case where any assignment of a pair of unique signatures to vertices in a static cell would yield a correct isomorphism mapping (figure 3.2), and a case where an arbitrary assignment could lead to a false negative. This last type of partitioning will be referred to as an *ambiguous* partitioning.

Backtracking immediately comes to mind as a solution. The procedure would be the following: create a search tree. Assign a unique signature to one vertex in a static cell in the reference graph, and let the vertices of the cell in the test graph be the children of the root vertex of the search tree. Assign the same unique signature to one of these children and proceed with refinement. The next time a forced choice has to be made, repeat the same procedure, this time adding the vertices as children of the last vertex on the path of the search tree. Once a partitioning is reached with cells with unequal numbers of vertices from the two graphs, instead of concluding that the two graphs are not isomorphic, start the refinement procedure over again from the initial partitioning this time taking an alternate path in the search tree that has been created. The problem with this approach is that it is exponential in the number of times a static partitioning is reached. Even if the number of ambiguous static partitionings that is reached is very small, say one, but they are followed by a large number of static partitionings run times become prohibitive.

Another solution would be to perform a depth-first search such as mentioned in the introduction, to map the vertices of the static partitioning. After two vertices of a static cell (one from each graph) have been mapped, the refinement algorithm can be used again. The problem with this approach

is that the 'depth' of the search may become large because the symmetrical sub-graphs may have arbitrary sizes, again yielding prohibitive run times. Furthermore this approach has the disadvantage of performing unnecessary searches in case the static partitionings are not ambiguous.

Still another, albeit heuristic, approach is the use of vertex invariants other than the degree. Various different approaches are discussed in references [12] and [2].

The tool Gemini does not use any of the approaches mentioned in this section. Rather, it forces unique signatures in the manner described in section (3.1.1). Thus false negatives are inherent. Gemini also checks in a *heuristic* fashion whether a static partitioning has been reached. It does this by allowing a maximal number of refinement steps without mapping any vertices. Nevertheless it is known to perform well for practical circuits.

This means that circuits which cause ambiguous static partitionings are uncommon. This in turn seems to imply that for a given circuit the number of ambiguous partitionings is small because a circuit can be regarded as a concatenation of sub-circuits. One might therefore argue that the backtracking approach mentioned at the beginning of this section is feasible. The problem with this approach, however, lies in the fact that while *ambiguous* static partitionings are uncommon, static partitionings are common in typical circuits with, for instance, parallel paths. If two circuits which are to be compared with each other contain discrepancies differences in cell sizes will always occur, even if they are not caused by a wrong forced assignment of unique signatures. The backtracking procedure will keep traversing the search tree without result.
We shall, therefore, use an approach very similar to Gemini. That is we use the method which was conjured to be correct and force unique signatures as described on page (25) in section (3.1.1). This is only done, however, once it has been established (in a *deterministic* manner) that the partitioning has not changed after a refinement step. Strictly speaking this is not the precise procedure that will be followed; further on another modification will be discussed concerning parameter values (section 3.2.3).

### 3.2.2 Component types and terminal classes

Component types and terminal classes can easily be incorporated into the refinement process by making use of them in the signaturing function as was mentioned in the introduction.

The component types only have to be used when constructing the initial partitioning, i.e. when assigning the initial signatures to the *components*. After this the signatures of the components contain the information regarding the component types. This means that the signaturing function for the components is different from that of the nets. We shall, therefore, use a separate partitioning for the components and the nets. A partitioning of components, $\Pi_C$, is a partitioning $\Pi$ with the additional property that its vertices are components. Similarly we have a partitioning of nets $\Pi_N \subseteq \Pi$. The signaturing function which assigns an initial signature, as in definition (3.1.8), to a component becomes:

$$initialSig(c) = type(c) \qquad (3.2.1)$$

The terminal classes which are assigned to the edges need to be taken into account in every partitioning step since the edges themselves are not partitioned, but rather the vertices which they connect. Each time a new signature is assigned to a vertex (using the cells to which the neighbors belong), the terminal class of the edge connecting it to its respective neighbors must also be incorporated. Therefore a new signature is a pair consisting of the current cell, and the multi-set of pairs of terminal

classes and the cells of neighboring vertices. In other words the signaturing function which assigns a new signature, as in definition (3.1.15), to the vertices becomes:

$$newSig(G, \pi, x) = (cell(\pi, x), (\biguplus y : \{x, y\} \in E(G) : (tc\{x, y\}, cell(\pi, y)) ) ) \quad (3.2.2)$$

This is, however, not the signaturing function as it will be used in the algorithm. Further on another modification is discussed which is related to error propagation.

### 3.2.3 Parameter values

The primary task of netlist comparison tool is to verify whether the structure of netlists is equivalent. Additionally it should be verified whether the parameter values of the components that are mapped onto each other are equivalent.
An obvious procedure would be to find an isomorphism first (based purely on the topology), and to check the parameter values afterwards. This can lead to problems however. Consider the simple circuit consisting of two parallel resistors, depicted in figure (3.5). If parameter values are not taken into account during the construction of the isomorphism mapping, the top resistor in the reference netlist might be mapped onto the bottom resistor in the netlist and vice-versa. This would lead to an error report when checking the parameter values. Clearly another approach is needed.

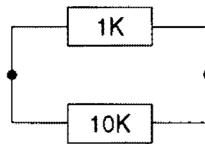



Figure 3.5: Two parallel resistors with different parameter values

Using the parameter values in the signaturing function in the mapping stage, however, might cause components not to be mapped because of differences in parameter values. This would prohibit a mapping of other components and nets which are correct, and therefore seems to be an unviable approach. Furthermore it is impossible to use the parameter values in the signaturing function directly because they have tolerances.

An alternative approach is necessary. We use the refinement algorithm without taking parameter values into account until a static partitioning is reached, as in the example of the two parallel resistors. At this stage the parameter values come into play; instead of assigning unique signatures as mentioned in section (3.2.1), it is attempted to split a cell of the partitioning using the parameter values. To do this two ordered sets, $O_0$ and $O_1$, with the vertices of the respective cell halves of a static cell, $C$, are created. Naturally $|O_0| = |O_1|$. The vertices in these sets are sorted according to one of their parameter values. Say the parameter with index $i$ (which is initially set to zero). Then the cells are split into sub-cells using markers or, more precisely, an ordered set of markers $M$.
The positions of these markers are determined by the values of parameter $i$ of the components in *one* of the sets, namely $O_0$: the set containing vertices from the reference netlist. $m_0$ points to the first element of $O_0$. $m_j$ is positioned such that all the components which fall left to it have a

parameter value which falls within the tolerance region of the component pointed to by of $m_{j-1}$. The component on the marker $m_j$, and all those to the right of it have a parameter with index $i$ which does not fall within this tolerance region. Using these markers a sub-partitioning, $\pi'_C$ is created which is the union of the sub-cells delimited by the markers. The original cell is removed from the partitioning $\pi_C$ and the new sub-partitioning is added. The previous can be formalized as follows:

For an ordered set of components $O$, an ordered set of markers $M$, a marker index $j$, and a marker parameter index $i$, the condition which must be satisfied can be written as:

$$
\begin{aligned}
&splitP(O, M, j, i) \equiv \qquad (3.2.3)\\
&m_0 = 0 \wedge (\forall c_k : k \in I\!N \wedge k < m_j \wedge c_k \in O : \\
&\qquad parameter_i(c_k) \leq parameter_i(c_{m_{j-1}}) + tol(type(c_k)) \\
&\quad ) \wedge \\
&\quad (\forall c_k : k \in I\!N \wedge k \geq m_j \wedge c_k \in O : \\
&\qquad parameter_i(c_k) > parameter_i(c_{m_{j-1}}) + tol(type(c_k)) \\
&\quad )
\end{aligned}
$$

This must hold for all marker indices so that the valid set of markers for some given set of components $O_0$, ordered on parameter with index $i$, equals:

$$M(O, i) := x : (\forall j : j \in \{1, 2, \ldots, |x|\} : splitP(O, x, j, i)) \qquad (3.2.4)$$

For some cell, $C$, and a set of markers,$M(O, i)$, the new sub-partitioning $\pi'_C$ is constructed using:

$$
\begin{aligned}
&\pi'_C(O, i) = (\cup j : j \in \{1, 2, \ldots, |M(O, i)|\} : \qquad (3.2.5)\\
&\qquad ( (\cup k : m_{j-1} \leq k < m_j : [O_0]_k) \,,\, (\cup k : m_{j-1} \leq k < m_j : [O_1]_k) ) \\
&\quad ) \cup \\
&\qquad ( (\cup k : m_j \leq k < |O_0| : [O_0]_k) \,,\, (\cup k : m_j \leq k < |O_0| : [O_1]_k) )
\end{aligned}
$$

The old cell, $C$, is removed from the partitioning $\pi_C$ and replaced by the newly created sub-partitioning:

$$\pi_C := (\pi_C \setminus C) \cup \pi'_C(O, i) \qquad (3.2.6)$$

As an example of the previous consider a cell $C = (\{a, b, c, d, e\}, \{u, v, w, x, y\})$, with the following functions which assign values to parameter zero of the components of the respective netlists:

$parameter_0 = \{(a, 1.1), (b, 1.0), (c, 5.0), (d, 9.3), (e, 9.1)\}$, and

$parameter_0 = \{(u, 8.9), (v, 5.3), (w, 8.9), (x, 0.9), (y, 3.0)\}$

Furthermore we define a tolerance of 10% for parameter zero.

The ordered sets, $O_0$ and $O_1$ will then equal:

$O_0 = (b, a, c, e, d)$

$O_1 = (x, y, v, u, w)$

The valid set of markers is then given by:

$M = \{0, 2, 3\}$

The resulting sub-partitioning is then given by:

$\pi'_C = \{(\{b, a\}, \{x, y\}), (\{c\}, \{v\}), (\{e, d\}, \{u, w\})\}$

Note that ultimately an error message should be given since the value of parameter zero of component $y$ (which equals 3.0), does not fall within the 10% tolerance range of parameter zero of either $a$ or $b$. □

Once a static partitioning is reached the method that has just been described is applied to all static cells, and for all parameter indices. The method of dealing with parameter values described here is different from that of Gemini [6].

### 3.2.4 Locating discrepancies

Besides verifying whether two netlist descriptions are equivalent, locating discrepancies between the reference and test netlists is an important part of the netlist comparison task. The refinement algorithm that has been discussed terminates as soon as the existence of a discrepancy is detected. It was explained on page (28) that merely continuing the refinement process is not a good method to map more vertices because discrepancies tend to propagate. A *heuristic* approach to solving this and other problems will be described. The basic idea is that of placing stringent demands on vertices which are mapped. Instead of immediately mapping vertices that are members of a cell containing one vertex from each graph, other properties are checked first. If no vertices can be matched, the requirements are gradually loosened until a pair of vertices is matched. Once this happens the requirements for mapping are made strict again. Four heuristics are used. They are those of preferably mapping neighbors of vertices that are already mapped, the limitation of error propagation by using the concept of 'suspect' and 'bad' vertices, using only unique neighbors in the signaturing function at a late stage, and trying an as best as possible match after all else fails. The details of changing the requirements for mapping, are described in section (3.2.5).

**Neighbors of mapped vertices**

In practical cases where netlists are compared the user usually gives a set of 'user-bindings'. This is a partial mapping of the vertices of the reference netlist onto the test netlist. The user might, for example, map the terminals of the circuit (such as the inputs, outputs, and power supply). This set of mappings should naturally be a subset of the final isomorphism mapping that is constructed by the tool. The partial mapping given by the user serves two purposes; it is an additional constraint which should be verified, and it can aid the construction of a mapping in case the two netlists are not isomorphic.

The refinement process leads to partitionings with cells with one vertex from each graph. This normally implies that the vertices can be mapped onto each other. If, however, the netlists are not isomorphic, such a conclusion might be premature. The singleton cells might be caused because the partitioning is corrupted by discrepancies. Furthermore they may cause a mapping which later causes discrepancies which contradict with the intuitive interpretation of the user. As a simple example consider the graphs of figure (3.6). Here the vertex marked 'X' has been mapped by the user. The initial partitioning is shown here with the degrees as signatures of the vertices. It can easily be seen that one more partitioning step will render the vertices to the left of the vertices with a signature '1' unique. Thus they will be mapped onto each other (as shown by the dotted line). Also the vertices to the right of the vertices with a signature 'X' will be mapped. This means that in the end the middle vertex in the top graph is left unmapped.

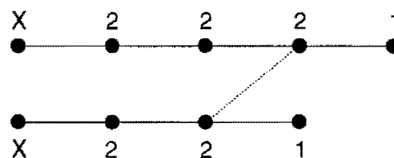



Figure 3.6: Mapping that leads to interpretation difficulties

Instead, the user might expect the rightmost vertex in the top graph to remain unmapped. Since it is

an 'extra vertex'. We see that there are different ways of interpreting the discrepancy. In order to be consistent with the intuitive notion of an extra vertex, an alternative approach is necessary. We therefore postpone the mapping of vertices which belong to singleton cells. It is first tried iteratively to map neighbors of vertices that are already mapped. Only when it becomes impossible to map neighbors of unique vertices, separate mappings are allowed. Using this approach, the portion of the netlist that is mapped expands around sections which are reliable.

### Suspect and bad vertices

After cells containing cell halves with unequal sizes are detected, the mapping process must continue, yet in such a manner that error propagation is limited. In order to do this we make use of the notions of *suspect* and *bad* vertices. These concepts are introduced here.

Vertices of a netlist, $G$, are qualified as *suspect* if they are the member of a cell half which differs in size from the corresponding cell half of the vertices of $H$, with the constraint that the corresponding cell half of $H$ is not an empty set. We shall call a cell with suspect vertices a suspect cell. Vertices of a netlist, $G$, are qualified as *bad* if they are the member of a cell half which has a corresponding cell half of the vertices of $H$ which is empty. A cell with bad vertices will be referred to as a bad cell.

As an example consider the following cell: $C = (\{a\}, \{x, y\})$. The vertices $a$, $x$, and $y$ are qualified as suspect. Consider the cell $C = (\emptyset, \{b, c\})$. The vertices $b$ and $c$ are qualified as bad.

The function which constructs the partial mapping, $map$, is altered such that after each partitioning step, the suspect and bad cells are detected. Their members are placed in the sets of suspect and bad vertices for each netlist. These are referred to as $S$ and $B$ respectively. As will be shown in the following paragraphs, the members of these sets are treated in a special manner.

It is important to note that the qualifiers say something about the vertices at a certain instance during the dynamic refinement process. If a vertex is qualified as suspect or bad at some instance, this does not mean that it will not be mapped eventually. Why will become apparent further on. Eventually, after no more progress can be made with the refinement process (i.e. a static partitioning is reached), the vertices in these sets will be assigned their *initial* signature so that the refinement starts over again locally. When this happens many of the vertices that became suspect or bad initially because of discrepancies at some distance, $d$, will not become suspect or bad again because they become unique within fewer than $d$ refinement steps. This can happen because reliable vertices in their vicinity might have been refined in a proper manner.

First we shall consider bad vertices. The propagation of errors can be restricted by not including bad neighbors in the signaturing function. That is when constructing signatures for the vertices, the cells of the bad neighbors are not included in the multi-set. Using a bad neighbor when constructing the signature for some vertex, $x$, in some cell half, $C_0$, would also cause $x$ to become bad because none of the vertices in $C_1$ have a neighbor with the same signature. Therefore the following function

is used for assigning new signatures to vertices. It is a modified version of the function defined by equation (3.2.2):

$$newSig(G, \pi, x) = (cell(\pi, x), (\biguplus y : \{x, y\} \in E(G) \wedge y \notin B(G) : (tc\{x, y\}, cell(\pi, y)) ) ) \quad (3.2.7)$$

In order to keep the mapping reliable, vertices that have only one non bad neighbor are only allowed to match if this neighbor is unique. Finally, the bad vertices are excluded from the refinement process since there is no point in assigning new signatures to them. How this will be done will become apparent in section (3.2.5).

The suspect vertices are also excluded from the refinement process, although not immediately. The reason that the suspect vertices should not be refined any further is that they are not reliable. It is not desired to map them accidentally as might happen if continued refinement would lead to singleton sub-cells. The problem with the concept of suspect vertices is that a suspect cell may also contain vertices that are at a large distance from any discrepancies. Consider as an example the graphs of figure (3.7). Here components are represented by open circles, with their types represented by the letters A,B, and C. After the initial partitioning step is made, all the nets have become suspect. Even those in the bottom hand corner of the graphs. If the suspect cells are not refined at all, none of the nets can be matched. This is a common problem since in practical circuits large numbers of nets have the same degree. To overcome this problem, the suspect cells are allowed a *small* number of refinement steps (one or two). This localizes the suspect vertices, yet prevents unwanted matches. Allowing one refinement step solves the problem in this example; the nets at the bottom right hand side of the graphs (with an initial signature of '2') *will* then be mapped.

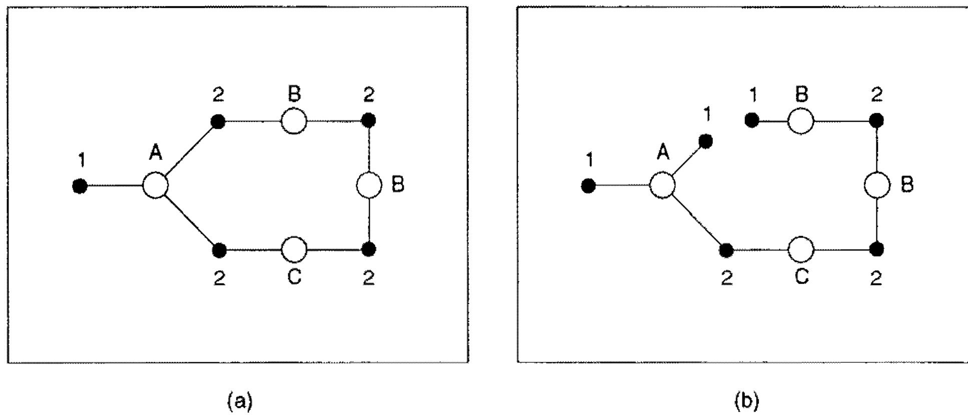



Figure 3.7: Two graphs with a discrepancy

As will be shown in section (3.2.5) the number of times that suspect vertices are allowed to take place in the refinement process is initially set to zero. Once the refinement process cannot proceed, more refinement steps are allowed for the suspect vertices. The maximum number of steps allowed can be given by the user, with a default value of one.

### Disregard vertex degree

Imagine that a component is missing from the test circuit, or that it has been connected to the wrong nets. This would mean that the net could *never* be matched (because its initial signature does not match that of the corresponding net in the reference netlist) thereby possibly preventing neighboring components from matching. If, however, the net has some neighboring components that *are* matched the problem can be solved. The solution lies in assigning alternative initial signatures to the unmatched nets that are based on the unique neighboring components instead of the degree of the net itself. This is done only when no more components or nets can be matched.
Thus we have an alternative signaturing function:

$$altInitialSig(G, \pi_C, n) = (\cup c : (n,c) \in E(G) \wedge c \in U(G) : (tc((n,c)), cell(\pi_C, c))\ ) \qquad (3.2.8)$$

Gemini follows the same procedure.

### Try best match

Here we introduce the last method used to map vertices in the case of non-isomorphic graphs. It is a very rough approach that is used only when all else fails. It maps vertices that are inherently different, because they have different properties or different neighbors, onto each other. An example of this are nets with differing degrees that could not be mapped with the method described in the previous paragraph. The reason that it is attempted to map these vertices, which might very well be the cause of discrepancies, anyway is that it can make it easier for the user to find discrepancies.

The approach that is used to map remaining vertices is the following: Find a vertex, $x$, in netlist $G$ that has the largest number of unique neighbors. At least one unique neighbor is required. Otherwise the netlist comparison terminates. Next, construct the multiset of images, $M$, in $H$ of the unique neighbors of $x$. From this set $M$ select the vertex, $y$, that has the largest number of occurences (multiplicity) in $M$. Finally, map $x$ onto $y$.
This can be formalized as follows:

For a given graph $G$, and sets of suspect and bad vertices, $S(G)$ and $B(H)$, the vertex, $x$ is given by:

$$\begin{aligned} x := x : x \in (S(G) \cup B(G)) \wedge (\forall x' : x' \in (S(G) \cup B(G)) :& \\ &|(\cup z : \{x,z\} \in E(G) : z \in U(G))| \geq \\ &|(\cup z : \{x',z\} \in E(G) : z \in U(G))| \\ )\wedge& \\ (\exists z : \{x,z\} \in E(G) : z \in U(G)\ )& \end{aligned} \qquad (3.2.9)$$

The multiset of images for two given graphs $G$ and $H$, and a vertex $x$ is given by:

$$\begin{aligned} &M(G,H,x) := \\ &(\underline{\cup} v, v' : v \in U(G) \wedge \{x,v\} \in E(G) \wedge v' \in (S(H) \cup B(H)) \wedge (v', f(v)) \in E(H) : v') \end{aligned} \qquad (3.2.10)$$

Finally the vertex in graph $H$ that is selected for mapping is given by:

$$y := y : y \in M \wedge (\forall y' : y' \in M : \mu_y(M) \geq \mu_{y'}(M)\ ) \qquad (3.2.11)$$

If the set $M$ is empty, the requirement for the number of unique neighbors is lowered down to a minimum of one. If it becomes impossible to create such a set, the netlist comparison process terminates.

### 3.2.5 Functions and algorithms

This section describes how the data structures, functions, and concepts that have been explained so far are applied to form a complete netlist comparison program. The transformation described in chapter two is not considered here since an algorithm for the construction of such a mapping is fairly trivial and more of an implementation issue. Thus, the program described here compares two netlists of type *Netlist*. The algorithm tries to create an as accurate isomorphism mapping as possible. After such a mapping has been constructed it is verified whether the mapping is indeed an isomorphism mapping. At this stage discrepancies are reported. We shall use a top-down approach when discussing the 'low level' functions. The remainder of the code will be discussed bottom-up. The 'low level' functions perform the mapping for some given set of requirements. At a higher level these requirements are altered.

As mentioned in section (3.2.4) the vertices of the graphs can be classified as *suspect*, *bad* and *unique*. All the remaining vertices will be referred to as *pending*. We can formalize this as follows:

**Definition 3.2.1** (*Augmented netlist for comparison*)
An augmented netlist for comparison, $Netlist^+$ is a netlist, $G = (C, N, name, tc, Nx, P, S, B, U)$, where:

- $C$ is a set of components as in definitions (2.2.1) and (2.1.6).
- $N$ is a set of nets as in definitions (2.2.1) and (2.1.6).
- $tc$ is a terminal class function as in definition (2.2.1).
- $Nx$ is a set of 'next evaluate' vertices.
- $P$ is a set of *pending* vertices.
- $S$ is a set of *suspect* vertices. See section (3.2.4).
- $B$ is a set of *bad* vertices. See section (3.2.4).
- $U$ is a set of *unique* vertices. See section (3.2.4).
- $C \cup N = P \cup S \cup B \cup U$
- $P \cap (S \cup B \cup U) = \emptyset$.

For convenience we shall use $V$ to denote the union of $C$ and $N$. □
All of the items of the definition given here have been introduced so far, except for the set of 'next evaluate' vertices. This set denotes the vertices that are used in the next refinement step. Initially all vertices are a member of this set. As mentioned, however, *bad* cells are not refined and *suspect* cells are only refined a limited a number of times. Thus the members of these sets must be removed form the set $Nx$ at some stage. Furthermore *static* cells can, under certain circumstances, be removed from the set of vertices which are refined. This improves the efficiency of the program considerably. The function *update* updates the set $Nx$ and is described further on. In the following, for the sake of readability, we shall refer to a $Netlist^+$ simply as a *Netlist*.

The requirements, or conditions, which are necessary (but not sufficient) for the matching of vertices onto each other are given by a global variable *matchChar* of type *MatchChar*. The state of this

variable also determines the number of times suspect cells are refined and whether it is *attempted* to split static component partitionings using the parameter values of their components.

**Definition 3.2.2** (*Match characteristics*)
The match characteristics are denoted by a triple:
$MatchChar = (suspectTime, splitMode, uniqueRequired)$, where:

- $suspectTime$ is a counter which indicates how many passes a suspect vertex remains in the $Nx$ set before it is removed from the $Nx$ set. It is initially set to zero and gradually increased to a maximum value set by the user:
  $suspectTime : \mathbb{N}$
- $splitMode$ A flag which indicates whether it will be attempted to split static component cells using the values of the member components. It is initially false. $splitMode$ : **bool**
- $uniqueRequired$ is a flag which indicates whether a vertex needs unique neighbors in order to be matched:
  $uniqeRequired$ : **bool**

□

### Notation

In the previous, the two graphs or netlists which where compared with each other were denoted by $G$ and $H$. Here we shall use a different notation. The reference and test graphs are referred to as $G_0$ and $G_1$ respectively. This allows us to perform an inversion on the graph index (the inverse of 0 is defined to be 1 here).

For some set of vertices $S$, $S_C$ denotes the subset of component vertices and $S_N$ denotes the subset of net vertices. The inverse of the type component is defined to be the type net, i.e. $S_{\overline{C}} = S_N$.

$Nx$ is used to denote the union of $Nx_0$ and $Nx_1$.

We shall use $f$ to denote the mapping of the vertices (both the nets and the components).

### Global variables

The set of variables given here partially describe that state of the netlist comparison process. They are made global since they say something about both the reference and test graphs. Furthermore they are used at all levels. An alternative approach would be to pass them as arguments for a large number of function calls thereby sacrificing readability:

1. $f : V(G_0) \mapsto V(G_1)$. The 'isomorphism' function which is constructed.
2. $\pi_C : \Pi_C$ The partitioning of the sets of components. See section (3.2.2).
3. $\pi^*_C : \Pi_C$ A variable to check whether the component partitioning has changed.
4. $\pi_N : \Pi_N$ The partitioning of the sets of the nets. See section (3.2.2).
5. $\pi^*_N : \Pi_N$ A variable to check whether the net partitioning has changed.

6. $splitEffect$ : **bool** A flag which indicates whether it may be possible to split cells of a static partitioning by using the parameter values of the components. Once it is set to false it will not be attempted to split the cells of a static partitioning on the values of the parameters of its components.

7. $passType$ : $\{net, component\}$ Since we are dealing with bi-partite graphs, the nets and components can be refined in an alternating fashion.

8. $matchChar$ : $MatchChar$ The characteristics used in the low level refinement as in definition (3.2.2).

9. $userSupectTime$ : $\mathbb{N}$ The maximum number of times that suspect vertices may be refined in the program. $suspectTime(matchChar)$ is never larger than this value.

10. $mode$ : $\{deterministic, heuristic\}$ If the mode is set to *deterministic*, the program will not force vertices to match in the manner described in section (3.2.1). This will in most cases only yield a partial partitioning. For certain applications, however, this is desired in order to be *sure* that no invalid mappings are made. An example of such an application is the mixed verification of structure and functionality used to verify sequential logic as mentioned in the introduction.

### Partitioning and mapping

This section considers the partitioning and mapping functions. They operate under a given set of matching requirements (determined by $matchChar$). The state of this variable is changed by the procedures which are described in the next section. The procedures described here are similar to those described in section 3.1.2.

The partitioning and mapping functions that have been considered so far took into account all the vertices in the reference and test graphs, except for those which were unique. Here a slightly different approach is used. The functions which are defined here only modify the signatures of the vertices that are members of the set of 'next evaluate' vertices, $Nx$. This allows us to stop refining suspect partitionings, for example.

The function $partition_0$, partitions the vertices in $Nx$ on their initial properties. Vertices which are not in $Nx$ are left in their respective cells. Using this approach the vertices of suspect and bad cells can be assigned to their initial partitioning at some stage.

$$\begin{aligned}
partition_0(G_0, G_1, \pi) = &(\cup x : x \in (Nx(G_0) \cup Nx(G_1))) : \\
&\quad (initialCellHalf(G_0, x), intialCellHalf(G_1, x)) \\
&)\cup \\
&(\cup x : x \in (V(G_0) \cup V(G_1)) \wedge x \notin (Nx(G_0) \cup Nx(G_1))) : \\
&\quad cell(\pi, x) \\
&)
\end{aligned} \qquad (3.2.12)$$

$$initialCellHalf(G, x) = (\cup y : y \in Nx(G) : initialSig(x) = initialSig(y)) \qquad (3.2.13)$$

The function $partition_t$ partitions all the vertices of $Nx$ of type $t$, i.e. vertices of $Nx_t$ using a 'new signature' function. The vertices which are not in $Nx$ are left in their respective cells.

$$partition_t(G_0, G_1, \pi) = (\cup x : x \in (Nx_t(G_0) \cup Nx_t(G_1)) : (newCellHalf(G_0, \pi, x), newCellHalf(G_1, \pi, x)) ) \cup (\cup x : x \in (t(G_0) \cup t(G_1)) \wedge x \notin (Nx(G_0) \cup Nx(G_1)) : cell_t(\pi, x) ) \quad (3.2.14)$$

$$newCellHalf(G, \pi, x) = (\cup y : y \in Nx(G) : newSig(G, \pi, x) = newSig(G, \pi, y)) \quad (3.2.15)$$

The function $altPartition_N$ partitions the vertices on an alternative initial signature (disregarding the vertex degree and looking at unique neighbor components), as discussed on page (40).

$$altPartition_{0N}(G_0, G_1, \pi) = (\cup x : x \in (Nx_N(G_0) \cup Nx_N(G_1)) : (altCellHalf(G_0, \pi_C, x), altCellHalf(G_1, \pi_C, x)) ) \cup (\cup x : x \in (N(G_0) \cup N(G_1)) \wedge x \notin (Nx(G_0) \cup Nx(G_1)) : cell(\pi_N, x) ) \quad (3.2.16)$$

$$altCellHalf(G, \pi_C, x) = (\cup y : y \in Nx_N(G) : altInitialSig(G, \pi_C, x) = altInitialSig(G, \pi_C, y)) \quad (3.2.17)$$

The procedure *refine* forms the main loop of the matching process. It is similar to program (3.1.17). The main difference is that the nets and components are refined in an alternating fashion.

**proc** *refine*=(|[ ( 3.2.18)

```
0.  π*_N := ∅;
1.  π*_C := ∅;
2.  do ((passType = net) ∧ (π_C ≠ π*_C))∨
3.     ((passType = component) ∧ (π_N ≠ π*_N)) →
4.          if passType = net → π*_N := π_N;
5.                              π_N := partition_N(G_0, G_1, π);
6.                              map_N;
7.                              passType := component
8.          [] passType = component → π*_C := π_C;
9.                              π_C := partition_C(G_0, G_1, π);
10.                             map_C;
11.                             passType := net
12.         fi ;
13. od

]|
)
```

The function $map_t$ has a structure similar to that of program (3.1.11). There are a few major differences though. Instead of iterating over cells, we iterate over vertices. This has the same effect, because for each vertex, its cell is taken. The reason that we do this is because we want to iterate only over the vertices of $Nx$.

Instead of immediately mapping vertices in line (3), the procedure *checkAndMatch* is called. This is done to prevent the vertices which do not satisfy the requirements as specified by *matchChar* from

matching.

Note that vertices are qualified as either *pending*, *suspect*, or *bad*. Vertices which become unique are qualified as such by the procedure *checkAndMatch*.

Two loops are made to prevent from mapping vertices twice. Finally, map makes a call to $update_t$ which updates the set $Nx$, by performing tasks such as the removal of bad vertices form $Nx$. The function returns **true** iff vertices have been mapped.

**bool func** $map_t$ =(|[**var** $x : V, U_t^* :$ **setOf**$(V)$| ( 3.2.19)

0. $U_t^* := U_t(G_0)$
1. **for** $x : x \in Nx_t(G_0) \rightarrow$
2. **if** $(|[cell(\pi,x)]_0| = |[cell(\pi,x)]_1|) \wedge (|[cell(\pi,x)]_0| = 1) \rightarrow$
3. $checkAndMatch(G_0, G_1, x, y : y \in [cell(\pi,x)]_{\bar{i}})$
4. $[]\ (|[cell(\pi,x)]_0| = |[cell(\pi,x)]_1|) \wedge (|[cell(\pi,x)]_0| \neq 1) \rightarrow$
5. $P(G_0) := P(G_0) \cup \{x\}$
6. $[]\ (|[cell(\pi,x)]_0| \neq |[cell(\pi,x)]_1|) \wedge ([cell(\pi,x)]_1 \neq \emptyset) \rightarrow$
7. $S(G_0) := S(G_0) \cup \{x\}$
8. $[]\ (|[cell(\pi,x)]_0| \neq |[cell(\pi,x)]_1|) \wedge ([cell(\pi,x)]_1 = \emptyset) \rightarrow$
9. $B(G_0) := B(G_0) \cup \{x\}$
10. **rof** ;
11. **for** $x : x \in Nx_t(G_1) \rightarrow$
12. **if** $(|[cell(\pi,x)]_1| = |[cell(\pi,x)]_0|) \wedge (|[cell(\pi,x)]_1| = 1) \rightarrow$
13. **skip**
14. $[]\ (|[cell(\pi,x)]_1| = |[cell(\pi,x)]_0|) \wedge (|[cell(\pi,x)]_1| \neq 1) \rightarrow$
15. $P(G_1) := P(G_1) \cup \{x\}$
16. $[]\ (|[cell(\pi,x)]_1| \neq |[cell(\pi,x)]_0|) \wedge ([cell(\pi,x)]_0 \neq \emptyset) \rightarrow$
17. $S(G_1) := S(G_1) \cup \{x\}$
18. $[]\ (|[cell(\pi,x)]_1| \neq |[cell(\pi,x)]_0|) \wedge ([cell(\pi,x)]_0 = \emptyset) \rightarrow$
19. $B(G_1) := B(G_1) \cup \{x\}$
20. **rof** ;
21. $update_t(G_0)$;
22. $update_t(G_1)$;
23. **return** $U_t^* \neq U_t(G_0)$

|]
)

The procedure $checkAndMatch$ maps the vertices passed as arguments if they satisfy the requirements as specified by $matchChar$. Furthermore it makes vertices with a bad neighborhood (see definition (3.2.21) ) bad. This prevents unreliable mappings. Note that if the procedure succeeds in mapping vertices, the match characteristics are reset to the most stringent value. The higher level functions described in the next section make the requirements less strict if it becomes impossible to map vertices.

In order to define a bad neighborhood, we use the notion of a neighbor set. The neighbor set of a vertex $x$, is simply the set of vertices adjacent to $x$:

$neighborSet(x) = (\cup y : \{x,y\} \in (E(G_0) \cup E(G_1)\ ) : y)$ ( 3.2.20)

A neighborhood of a vertex $x$, is considered bad if $x$ has neighbors, and none of them are unique and $x$ does not have at least two non bad neighbor vertices:

$$badNeighborhood(x) \equiv (\exists y : y \in neighborSet(x) : y \in (B(G_0) \cup B(G_1)) ) \wedge \neg(\exists y : y \in neighborSet(x) : y \in (U(G_0) \cup U(G_1)) ) \wedge |(\cup y : y \in neighborSet(x) \wedge y \notin (B(G_0) \cup B(G_1)) : y)| < 2 \quad (3.2.21)$$

**proc** $checkAndMatch$=($\downarrow x, \downarrow y : V$| (3.2.22)

|[

0. **if** $badNeigborhood(x) \vee badNeighborhood(y) \rightarrow$
1. $B(G_0) := B(G_0) \cup x;$
2. $B(G_1) := B(G_1) \cup y$
3. [] $\neg(badNeigborhood(x) \vee badNeighborhood(y)) \rightarrow$
4. **if** $\neg uniqueRequired(matchChar) \vee hasUniqueNeighbor(x) \rightarrow$
5. $f(x) := y;$
6. $U(G_0) := U(G_0) \cup \{x\};$
7. $U(G_1) := U(G_1) \cup \{y\};$
8. $matchChar := (0, \mathbf{false}, \mathbf{true})$
9. [] $uniqueRequired(matchChar) \wedge \neg hasUniqueNeighbor(x) \rightarrow$
10. **skip**
11. **fi**
12. **fi**

|]

)

The procedure $update_t$ updates the set of 'next evaluate vertices', $Nx$. As mentioned only vertices which are a member of $Nx$ are refined. Bad vertices are removed immediately from $Nx$. Suspect vertices are removed if the number of times that they have been refined, and stayed suspect (denoted by $time(x)$) becomes larger than, or equal to a maximum specified by $matchChar$.

If none of the cells of the neighbors of the vertices of a cell, $C$, are partitioned during a refinement step, the cell $C$ will not be refined in a next refinement step because the cells of the neighboring vertices are used to partition $C$. Since we refine in an alternating fashion (nets/components), this implies that the vertices which are member of a cell, $C$, which has been partitioned during the last refinement step can be removed from $Nx$. If, however, a cell *is* partitioned, all the cells of neighboring vertices must be inserted in $Nx$ again for refinement. Removing *temporarily* static cell from $Nx$ has the advantage of reducing run times considerably. A similar procedure is applied in [3].

It is important to set $\pi^*$ equal to $\pi$ before it is attempted to change a partitioning in order to recognize changes with the function $update_t$. In the actual implementation, a different mechanism is used to monitor cell changes.

**proc** $update_t$=( $\downarrow G : Netlist$| (3.2.23)

|[

0. **for** $x : x \in Nx_t(G) \rightarrow$
1. **if** $x \in P(G) \rightarrow$
2. **if** $cell(\pi_t, x) \neq cell(\pi_t^*, x) \rightarrow$
3. $Nx_{\bar{t}} := Nx_{\bar{t}} \cup (\cup x' : (\{x, x'\} \in E(G)) \wedge x' \in P(G) : x')$
4. [] $cell(\pi_t, x) = cell(\pi_t^*, x) \rightarrow$
5. $Nx_t(G) := Nx_t(G) \setminus \{x\}$

```
6.              fi
7.          [] x ∈ S(G) →
8.              if time(x) ≥ suspectTime(matchChar) →
9.                          Nx_t(G) := Nx_t(G) \ {x}
10.             [] time(x) < suspectTime(matchChar) →
11.                         skip
12.         [] x ∈ B(G) →
13.             Nx_t(G) := Nx_t(G) \ {x}
14.         [] x ∈ U(G) →
15.             Nx_t(G) := Nx_t(G) \ {x};
16.             Nx_t̄ := Nx_t̄ ∪ (∪x' : ({x, x'} ∈ E(G)) ∧ (x' ∈ P(G)) : x')
17.         fi
18. rof
|]
)
```

### Variation of match characteristics

The previous section described the refinement and mapping functions. The mapping functions were used to construct a mapping for given requirements. Furthermore suspect and bad cells were removed form the set $Nx$. This section describes the way $matchChar$ is altered, and the way suspect and bad vertices are handled.

A set of procedures, each of which varies a particular attribute of $matchChar$, is introduced bottom-up. These functions all have the prefix '*var*'. Furthermore, the auxillary functions and procedures $splitOnParameter$, $forceMatch$, and $matchWithBestNeighbors$ are introduced.

We use the predicate $allAreMatched$ to determine whether the mapping process is complete. $allAreMatched$ is true iff all the vertices of one of the netlists are matched.

$$allAreMatched \equiv U(G_0) = V(G_0) \vee U(G_1) = V(G_1) \qquad (3.2.24)$$

The procedure $varSuspect$ varies the number of times that suspect vertices may be refined before they are removed from the set $Nx$. First a call to *refine* is made. Once it becomes impossible for *refine* to map anymore vertices, it returns. Suspect and bad vertices are re-assigned their initial signatures and inserted in the set $Nx$ in lines (7-14). If $splitMode(matchChar)$ is true and if splitting has any effect, the cells are split on the parameter values of the member components. $splitMode(matchChar)$ is only set to **true** once it becomes impossible to map any more vertices without doing so.

$$\textbf{proc}\ varSuspect\text{=}(|[\textbf{var}\ temp_0, temp_1 : \textbf{setOf}(V), loop : \textbf{bool}| \qquad (3.2.25)$$

```
0.  loop := true ;
1.  suspectTime(matchChar) := 0;
2.  do loop →
3.          refine;
4.          if allAreMatched → loop := false
5.          [] ¬allAreMatched(G_0, G_1) →
6.                          π* := π;
7.                          temp_0 := Nx_0;
8.                          temp_1 := Nx_1;
```

```
9.                      Nx_0 := S(G_0) ∪ B(G_0);
10.                     Nx_1 := S(G_1) ∪ B(G_0);
11.                     S(G_0) := ∅; B(G_0) := ∅; S(G_1) := ∅; B(G_1) := ∅;
12.                     π := partition_0(G_0, G_1, π);
13.                     Nx_0 := Nx_0 ∪ temp_0;
14.                     Nx_1 := Nx_1 ∪ temp_1;
15.                     suspectTime(matchChar) = suspectTime(matchChar) + 1;
16.                     if suspectTime(matchChar) > userSuspectTime → loop := false
17.                     [] suspectTime(matchChar) ≤ userSuspectTime →
18.                                 map_C;
19.                                 map_N;
20.                                 if splitMode(matchChar) ∧ splitEffect →
21.                                           Nx(G_0) := Nx(G_0) ∪ P(G_0);
22.                                           Nx(G_1) := Nx(G_1) ∪ P(G_1);
23.                                           splitEffect = splitOnParameter()
24.                                 [] ¬splitMode(matchChar) ∨ ¬splitEffect → skip
25.                                 fi
26.                     fi
27.         fi
28. od
|]
)
```

The procedure *varUnique* first tries to use *varSuspect* to map only vertices neighboring unique vertices. Once no more vertices can be mapped, the requirement of unique neighbors is dropped.

**proc** *varUnique*=(|[**var** *loop* : **bool**| (3.2.26)

```
0.  loop := true ;
1.  uniqueRequired(matchChar) := true
2.  do loop→
3.          varSuspect;
4.          if allAreMatched∨¬uniqueRequired(matchChar) →
5.                          loop := false
6.          [] ¬(allAreMatched ∨ ¬uniqueRequired(matchChar)) →
7.                          uniqueRequired(matchChar) := false
8.          fi
9.  od
|]
)
```

The procedure *varSplit* first tries to map vertices using *varUnique* without using parameter values to split cells. Once a static partitioning is reached, the pending vertex are inserted in $Nx$ , and it is attempted to partition static cells using *splitOnParameter*.

**proc** *varSplit*=(|[*loop* : **bool**| (3.2.27)

```
0.  loop := true ;
1.  splitMode(matchChar) := false ;
2.  do loop∧¬allAreMatched →
```

3. $varUnique$;
4. **if** $splitMode(matchChar) \rightarrow loop :=$ **false**
5. $[]\ \neg splitMode(matchChar) \rightarrow$
6. $splitMode(matchChar) :=$ **true** ;
7. **if** $splitEffect \rightarrow$
8. $Nx(G_0) := Nx(G_0) \cup P(G_0)$;
9. $Nx(G_1) := Nx(G_1) \cup P(G_1)$;
10. $splitEffect := splitOnParameter(G_0, G_1, \pi_c)$;
11. **if** $\neg splitEffect \rightarrow loop :=$ **false** ;
12. $[]\ splitEffect \rightarrow$ **skip**
13. **fi**
14. $[]\ \neg splitEffect \rightarrow loop :=$ **false**
15. **fi**
16. **fi**
17. **od**

|]
)

The auxillary function $splitOnParameter$ is used to partition static cells on parameter values of the components as described on page (35). The sets $Done_0$ and $Done_1$ are used to keep track of the cells that have been considered in the **for** loop. The function returns true iff a partitioning has taken place.

**bool func** $splitOnParameter$ =(|[$Done_0, Done_1$ : **setOf**$(C)$, $partitioned$ : **bool**| (3.2.28)

0. $partitioned :=$ **false** ;
1. $Done_0 := \emptyset$;
2. $Done_1 := \emptyset$;
3. **for** $i : i \in \{0, 1, \ldots,$ 'The maximum parameter index over all components'$\} \rightarrow$
4. $\pi^* := \pi$;
5. **for** $c : c \in Nx_C(G_0) \rightarrow$
6. **if** $|[cell(\pi, c)]_0| = |[cell(\pi, c)]_1| \wedge |[cell(\pi, c)]_0| > 1 \wedge |parameters(type(c))| > i \rightarrow$
7. $O := cell(\pi, c)$;
8. $Done_0 := Done_0 \cup [O]_0$;
9. $Done_1 := Done_1 \cup [O]_1$;
10. $Nx(G_0) := Nx(G_0) \setminus [O]_0$;
11. $Nx(G_1) := Nx(G_1) \setminus [O]_1$;
12. $\pi_C := (\pi_C \setminus O) \cup \pi'_C(O, i)$ { assignment (3.2.6) }
13. $[]\ \neg(\ |[cell(\pi, c)]_0| = |[cell(\pi, c)]_1| \wedge |[cell(\pi, c)]_0| > 1 \wedge |parameters(type(c))| > i\ ) \rightarrow$
14. **skip**
15. **fi**
16. **rof** ;
17. $Nx(G_0) := Nx(G_0) \cup Done_0$;
18. $Nx(G_1) := Nx(G_1) \cup Done_1$;
19. **if** $\pi^* \neq \pi \rightarrow$
20. $partitioned :=$ **true** ;
21. $map_C$;
22. $passType := net$
23. $[]\ \pi^* = \pi \rightarrow$ **skip**
24. **fi**

```
25. rof ;
26. return partitioned

|]
)
```

The procedure *varForce* tries to construct an isomorphism mapping by calling *varSplit*. If this is impossible, and the program is not running in *deterministic* mode, vertices are forced to match using *forceMatch*. It is first attempted to force neighbors of unique vertices. If no static cells have neighbors that are unique, it is attempted to force a match without the requirement of unique neighbors.

**proc** *varForce*=(|[**var** *loop*, *success* : **bool**| (3.2.29)

```
0.  loop := true ;
1.  do loop ∧ ¬allAreMatched →
2.          varSplit;
3.          if mode =¬deterministic →
4.                  uniqueRequired(matchChar) := true ;
5.                  Nx(G0) := Nx(G0) ∪ P(G0);
6.                  Nx(G1) := Nx(G1) ∪ P(G1);
7.                  success := forceMatch;
8.                  if ¬success→
9.                          uniqueRequired(matchChar) := false ;
10.                         loop := forceMatch
11.                 [] sucess → skip
12.         [] mode = deterministic → loop := false
13.         fi
14. od

|]
)
```

The function *forceMatch* uses two predicates to select vertices for mapping:
$PPairU_t(x, y)$ is true iff $x$ and $y$ can be mapped onto each other. It requires that (1) $x$ and $y$ are members of a cell with halves of equal size and (2) both $x$ and $y$ have unique neighbors. The fact that $x$ and $y$ have unique neighbors imples that neither $x$ nor $y$ has a bad neighborhood.

$$PPairU_t(x, y) \equiv (\exists c : c \in \pi_t \wedge |[c]_0| = |[c]_1| : x \in [c]_0 \wedge y \in [c]_1) \wedge (\exists x' : x' \in neighborSet(x) : x' \in U(G_0)) \wedge (\exists y' : y' \in neighborSet(y) : y' \in U(G_1)\,) \quad (3.2.30)$$

$PPair_t(x, y)$ is similair to $PPairU_t(x, y)$. Instead of requiring unique neighbors, a less strict demand is placed: Neither of $x$ nor $y$ may have a bad neighborhood.

$$PPair_t(x, y) \equiv (\exists c : c \in \pi_t \wedge |[c]_0| = |[c]_1| : x \in [c]_0 \wedge y \in [c]_1) \wedge \neg badNeighborhood(x) \wedge \neg badNeighborhood(y) \quad (3.2.31)$$

The function *forceMatch* tries two force two vertices for a given *matchChar*. It attempts to map vertices form a cell with a minimal size. The function returns true iff vertices have been mapped.

**bool func** *forceMatch* =(|[**var** $x, y : V, c : Cell, U^* :$ **setOf**$(V)$| (3.2.32)

0. $U^* := U(G_0)$;
1. $\pi^* := \pi$;
2. **if** $uniqueRequired(matchChar) \rightarrow$
3. $(x,y) := (x,y) : PPairU_N(x,y) \wedge (\forall x',y' : PPairU_N(x',y') : cell(\pi,x') \geq cell(\pi,x))$
4. $[]\ \neg uniqueRequired(matchChar) \rightarrow$
5. $(x,y) := (x,y) : PPair_N(x,y) \wedge (\forall x',y' : PPair_N(x',y') : cell(\pi,x') \geq cell(\pi,x))$
6. **fi**
7. **if** $(x,y) \neq \emptyset \rightarrow$
8. $c := cell(\pi,x)$;
9. $checkAndMatch(x,y)$;
10. $\{x \in U(G_0) \wedge y \in U(G_1)\}$
11. $\pi_N := (\pi_N \setminus c) \cup ([c]_0 \setminus x, [c]_1 \setminus y) \cup (\{x\},\{y\})$;
12. $Nx(G_0) := Nx(G_0) \cup [c]_0$;
13. $Nx(G_1) := Nx(G_1) \cup [c]_1$;
14. $update_N$;
15. $passType := component$
16. $[]\ (x,y) = \emptyset \rightarrow$
17. **if** $uniqueRequired(matchChar) \rightarrow$
18. $(x,y) := (x,y) : PPairU_C(x,y) \wedge (\forall x',y' : PPairU_C(x',y') : cell(\pi,x') \geq cell(\pi,x))$
19. $[]\ \neg uniqueRequired(matchChar) \rightarrow$
20. $(x,y) := (x,y) : PPair_C(x,y) \wedge (\forall x',y' : PPair_C(x',y') : cell(\pi,x') \geq cell(\pi,x))$
21. **fi**
22. **if** $(x,y) \neq \emptyset \rightarrow$
23. $c := cell(\pi,x)$;
24. $checkAndMatch(x,y)$;
25. $\{x \in U(G_0) \wedge y \in U(G_1)\}$;
26. $\pi_C := (\pi_C \setminus c) \cup ([c]_0 \setminus x, [c]_1 \setminus y) \cup (\{x\},\{y\})$;
27. $Nx(G_0) := Nx(G_0) \cup [c]_0$;
28. $Nx(G_1) := Nx(G_1) \cup [c]_1$;
29. $update_C$;
30. $passType := net$
31. $[]\ (x,y) = \emptyset \rightarrow$ **skip**
32. **fi**
33. **fi** ;
34. **return** $U^* \neq U(G_0)$

|]
)
The procedure *varInitial* maps vertices using *varForce*. If no more vertices can be matched, it is attempted to match more vertices by assigning new initial signatures to the nets as desribed on page (40).

**proc** *varInitial*=(|[**var** *loop*, *success*, *stopNextTime* : **bool**| (3.2.33)

0. *loop*, *success* : **bool**
1. *loop* := **true** ;
2. *stopNextTime* := **false**
3. **do** *loop*$\rightarrow$
4. *varForce*;
5. **if** $\neg allAreMatched \rightarrow$

```
6.                  π* := π;
7.                  π_N := altPartition_0N(G_0, G_1, π);
8.                  passType := component;
9.                  success := map_N;
10.                 if ¬success→
11.                         sucess := forceMatch;
12.                         if ¬sucess→
13.                                 if stopNextTime → loop := false
14.                                 [] ¬stopNextTime → stopNextTime := true
15.                                 fi
16.                         [] sucess → skip
17.                         fi
18.                 [] sucess → skip
19.                 fi
20.         [] allAreMatched → loop := false
21.         fi
22. od
|]
)
```

**Try best match**

When all else fails, the method desribed on page (40) is used to map remaining vertices.

proc *varBest*=(|[**var** *loop* : **bool**| ( 3.2.34)

```
0.  loop := true ;
1.  do loop∧¬allAreMatched →
2.          varInitial;
3.          suspectTime(matchChar) := 0;
4.          π*_C := π_C;
5.          π_C := partition_C(G_0, G_1, π);
6.          map_C;
7.          π*_N := π_N;
8.          π_N := partition_N(G_0, G_1, π);
9.          map_N;
10.         loop := matchWithBestNeighbors;
11. od
|]
)
```

**bool func** *matchWithBestNeighbors* =(|[**var** $x, y : V, iV$ : **setOf**($V$), $U^*$ : **setOf**($V$), *loop* : **bool**| ( 3.2.35)

```
0.  π* := π;
1.  U* := U(G_0);
2.  iV := ∅; { Impossible vertices }
3.  loop := true ;
```

4. **for** *loop* $\rightarrow$
5. $\quad x := x : x \in (S_N(G_0) \cup B_N(G_0)) \setminus iV \wedge (\forall x' : x' \in (S_N(G_0) \cup B_N(G_0)) \setminus iV :$
6. $\quad\quad |(\cup z : \{x, z\} \in E(G_0) : z \in U(G))| \geq$
7. $\quad\quad |(\cup z : \{x', z\} \in E(G_0) : z \in U(G))|$
8. $\quad )\wedge$
9. $\quad (\exists z : \{x, z\} \in E(G_0) : z \in U(G)\ )$ { see definition (3.2.9) }
10. $\quad$ **if** $x = \emptyset \rightarrow$
11. $\quad\quad$ *loop* $:=$ *false*
12. $\quad [\!]\ x \neq \emptyset \rightarrow$
13. $\quad\quad M(G, H, x) :=$
14. $\quad\quad (\underline{\cup} v, v' : v \in U(G) \wedge \{x, v\} \in E(G) \wedge v' \in (S(H) \cup B(H)) \wedge (v', f(v)) \in E(H) : v');$
    { see definition (3.2.10) }
15. $\quad\quad$ **if** $M = \emptyset \rightarrow$
16. $\quad\quad\quad impossibleV := impossibleV \cup \{x\}$
17. $\quad\quad [\!]\ M \neq \emptyset \rightarrow$
18. $\quad\quad\quad y := y : y \in M \wedge (\forall y' : y' \in M : \mu_y(M) \geq \mu_{y'}(M)\ );$ { see definition (3.2.11) }
19. $\quad\quad\quad [cell(\pi, x)]_0 := [cell(\pi, x)]_0 \setminus \{x\};$
20. $\quad\quad\quad [cell(\pi, y)]_1 := [cell(\pi, y)]_0 \setminus \{y\};$
21. $\quad\quad\quad \pi_N := \pi_N \cup (\{x\}, \{y\});$
22. $\quad\quad\quad Nx(G_0) := Nx(G_0) \cup \{x\};$
23. $\quad\quad\quad Nx(G_1) := Nx(G_1) \cup \{y\};$
24. $\quad\quad\quad map_N;$
25. $\quad\quad\quad$ *passType* $:=$ *component*;
26. $\quad\quad\quad$ *loop* $:=$ *false*
27. $\quad\quad$ **fi**
28. $\quad$ **fi**
29. **rof** ;
30. **if** $U^* \neq U(G_0) \rightarrow$
31. $\quad$ **skip**
32. $[\!]\ U^* = U(G_0) \rightarrow$
33. $\quad$ *loop* $:=$ **true** ;
34. $\quad$ **for** *loop* $\rightarrow$
35. $\quad\quad x := x : x \in (S_C(G_0) \cup B_C(G_0)) \setminus iV \wedge (\forall x' : x' \in (S_C(G_0) \cup B_C(G_0)) \setminus iV :$
36. $\quad\quad\quad |(\cup z : \{x, z\} \in E(G_0) : z \in U(G))| \geq$
37. $\quad\quad\quad |(\cup z : \{x', z\} \in E(G_0) : z \in U(G))|$
38. $\quad\quad\quad )\wedge$
39. $\quad\quad\quad (\exists z : \{x, z\} \in E(G_0) : z \in U(G)\ )$ { see definition (3.2.9) }
40. $\quad\quad$ **if** $x = \emptyset \rightarrow$
41. $\quad\quad\quad$ *loop* $:=$ *false*
42. $\quad\quad [\!]\ x \neq \emptyset \rightarrow$
43. $\quad\quad\quad M(G, H, x) :=$
44. $(\underline{\cup} v, v' : v \in U(G) \wedge \{x, v\} \in E(G) \wedge v' \in (S(H) \cup B(H)) \wedge (v', f(v)) \in E(H) \wedge type(x) = type(v') : v');$
    { see definition (3.2.10) }
45. $\quad\quad\quad$ **if** $M = \emptyset \rightarrow$
46. $\quad\quad\quad\quad impossibleV := impossibleV \cup \{x\}$
47. $\quad\quad\quad [\!]\ M \neq \emptyset \rightarrow$
48. $\quad\quad\quad\quad y := y : y \in M \wedge (\forall y' : y' \in M : \mu_y(M) \geq \mu_{y'}(M)\ );$ { see definition (3.2.11) }
49. $\quad\quad\quad\quad [cell(\pi, x)]_0 := [cell(\pi, x)]_0 \setminus \{x\};$

```
50.                     [cell(π, y)]₁ := [cell(π, y)]₀ \ {y};
51.                     π_C := π_C ∪ ({x}, {y});
52.                     Nx(G₀) := Nx(G₀) ∪ {x};
53.                     Nx(G₁) := Nx(G₁) ∪ {y};
54.                     map_C;
55.                     passType := net;
56.                     loop := false
57.             fi
58.        fi
59.    rof
60. fi ;
61. return U* ≠ U

]|
)
```

**Error reporting**

After a mapping, $f$, has been constructed it must be verified whether this mapping is indeed a parameterically correct isomorphism mapping. The function *report* does this by simply iterating over all components and nets, and checking whether $f$ preserves adjaceny and whether the parameter values of the components match. Discrepancies are reported. The next chapter considers the error report in more detail.

**The netlist comparison program**

The netlist comparison program reads the graphs and the (partial) mapping specified by the user. The user also specifies the mode and the maximum number of times that suspect vertices are refined. The vertices that have been mapped by the user are assigned unique cells in lines (5-6). The other vertices are assigned initial signatures in line (17). The procedure *varBest* performs the actual mapping. Finally, an error report is generated by the function *report*.

```
|[

0.  userSuspectTime := ...; { defaulting to 1 }
1.  mode := ...; { deterministic or heuristic }
2.  G₀ := ...;
3.  G₁ := ...;
4.  f := ...;
5.  π_C := (∪p : p ∈ f ∧ [p]₀ ∈ C(G₀) : ([p]₀, [p]₁) );
6.  π_N := (∪p : p ∈ f ∧ [p]₀ ∈ N(G₀) : ([p]₀, [p]₁) );
7.  U(G₀) := (∪p : p ∈ f : [p]₀);
8.  U(G₁) := (∪p : p ∈ f : [p]₁);
9.  S(G₀) := ∅;
10. S(G₁) := ∅;
11. B(G₀) := ∅;
12. B(G₁) := ∅;
13. Nx(G₀) := V(G₀) \ U(G₀);
```

14. $Nx(G_1) := V(G_1) \setminus U(G_1)$;
15. $matchChar := (0, \textbf{false}, \textbf{true})$;
16. $splitEffect := \textbf{true}$;
17. $\pi := partition_0$;
18. $\pi^* := \emptyset$;
19. $map_N$;
20. $map_C$;
21. $passType := net$;
22. $varBest$;
23. $report$;
24.

|]

# Chapter 4

# Implementation

This chapter describes an implementation of the netlist comparison program described in the previous chapters. It is called STRICT, which is an acronym for STRucture Iso Check Tool. The program is written in C++ and runs on HP-UX 9.0.

## 4.1 Use

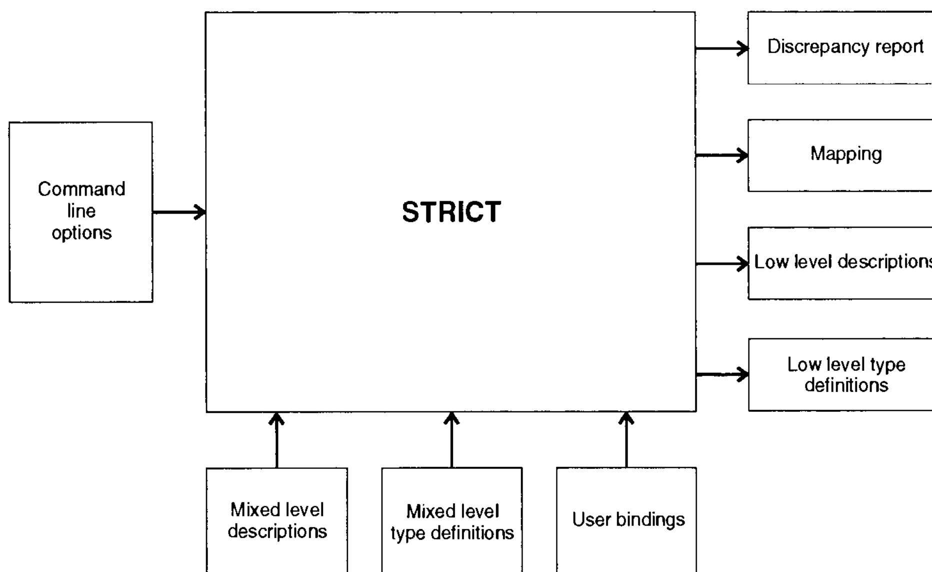



Figure 4.1: STRICT and input/output

Figure (4.1) shows a schematic view of STRICT's architecture. On the left hand side we see the command line options that are given by the user. These are discussed in section (4.1.1). The bottom of the figure shows the three possible input files. The term 'mixed level' is used here to indicate that the input files may contain descriptions and type definitions at arbitrary abstraction levels. That is,

the permutation sets of the component pins may be described by both permutation trees, and terminal classes. The right hand side of the figure shows the four possible output files.
STRICT can perform two functions:

- It can transform a mixed level description into a low-level description (*Netlist′* to *Netlist*). The low level description can then by used as input to a netlist comparison program such as Gemini.
- It can perform a complete netlist comparison.

### 4.1.1 Command line options

Table (4.1) explains the command line options. The '-' command line options should be passed on the command line *before* the other arguments. The other arguments are filenames. The first filename encountered on the command line is the reference netlist description. The second filename is the test netlist description. If a transformation is made (using the `-g` and `-y` options), only one netlist filename may be specified.
The two major modes of running STRICT are the following:

1. `STRICT -Rtypes.td reference.nl test.nl > discrepancies`
2. `STRICT -Rtypes.td -greference.out -ytypes.out reference.nl`

| Option | Arguments | Default | Meaning |
|---|---|---|---|
| `-R` | filename | | input type definitions |
| `-g` | filename | | output descriptions (netlist file) |
| `-y` | filename | | output type definitions |
| `-e` | number | 100 | maximum number of *file* errors before terminating |
| `-w` | number | 100 | maximum number of *file* warnings before terminating |
| `-D` | filename | | output mapping |
| `-E` | filename | | input mapping |
| `-d` | | not set | STRICT runs in *deterministic* mode if this flag is set (if there are no discrepancies) |
| `-t` | number on [0,65536] | 1 | Influences mapping process if there are discrepancies. The higher this number, the slower the program. Higher values can give more accurate results. |

Table 4.1: Command line options

### 4.1.2 File formats

This section discusses the various file formats in an informal manner. A precise grammar definition is not given here.
Files of any type may contain comments. These may either by single line comments which are started with the following 2-sequence of characters: `c:` (The character 'c' immediately followed by a colon ), or multi-line comments which are delimited by the following 2-sequences of characters: '`/:`' and '`:/`'. Multi line comments may be nested. In the following, 'white-space' refers to spaces and 'tabs'. Floating point values can be either (signed) simple decimal numbers such as '`0`', and '`-1.123`', or they can consist of a decimal mantissa followed by the symbol '`e`' or '`E`' followed by an integer exponent. An example of such a representation is '`-1.123e-1`'. Floating point and other values may not contain white-spaces.

#### Mixed level type definitions

The mixed level type definitions file consists of a non empty set of type definitions. For each type at least the terminals and their permutability must be specified. The permutability can be represented in two ways. Optionally one may specify parameters. If parameters are given, tolerances should be specified.
A definition starts with the keyword '`define:`'. It is followed by a white-space, a unique component type name, an open brace ('`(`'), a list of terminal names separated by commas, a close brace, a white-space and an optional list of parameter names separated by commas. The line must terminate with a semicolon.
On the next non empty line, the terminal permutability must be specified. The line must start with the keyword '`pins:`'. It is followed by a white-space. After this there are two possibilities:

1. A list of terminal classes is given. The terminal classes must be natural numbers. They are separated by commas. The line ends with a semicolon.

2. A permutation tree is given. Terminals or sets of terminals which are fixed relative to each other are enclosed by `(f ... )`. Terminals or sets of terminals which can be freely permuted are enclosed by `(p ...)`. The actual names of the terminals are used.

If a component type has parameters, parameter tolerances must be specified. A tolerance declaration starts with the keyword '`tolerance:`'. It is followed by a white-space and a list of parameter tolerances separated by commas. The order of the parameter tolerances is the same as the order of the parameter names in the declaration.
An example of type definition file is given in figure (4.2).

```
define: pmos(g,d,s) w,l;
pins: 0,1,1;
tolerances: 0.01,1e-2;  c:

define: SR(s,r,q,q_not);
pins: (p (f s,q),(f r,q_not ) );
```

Figure 4.2: Example of a type definition file

### Mixed level descriptions

The high level circuit description is 'component oriented'. It consists of a list of component instantiations where for each component the connected nets and the parameter values are given. Thus the nets are implicitly defined. Each line will be referred to as an entry. An entry may have zero or one component instantiation. A component instantiation must be followed by a semicolon: ';'. A component instantiation consists of the type name of the component, followed by a white-space, a unique component name, an open brace ('('), a list of net names separated by commas (','), a close brace (')'), a white-space, and an list of floating point values representing the parameter values, separated by commas.
An example of a mixed level description is given in figure (4.3)

```
SR ff1(a,b,c,d):
SR ff2(a,x,y,z);
pmos t1(a,e,f);
```

Figure 4.3: Example of a mixed level description

### User bindings

A user bindings file consists of a series of lines starting with either the keyword '`component:`', or the keyword '`net:`'. A line which starts with the keyword '`net:`' is used to define a net binding. The keyword is followed by a white-space, a net name in the reference netlist, a white space, and a net name in the test netlist. The same holds for components.

### Discrepancy report

The discrepancy report is written to `stdout`. The following errors are reported:

- Unmapped components and nets. The neighbors, which *have* been mapped, of the unmapped components and nets are reported to ease the location of discrepancies. If these neighboring vertices are not mapped properly, they are qualified as unreliable and are reported as such.
- Terminal class mismatches.
- Wrong surroundings of nets, caused either by improper mappings of adjacent components, or by mismatches of terminal classes. A list of causes is given.
- Parameter value mismatches.

The discrepancy reports are clustered. Connected nets and components which are erroneous are printed in groups.

### Mapping

The mapping file has the same format as the user bindings file. It contains comments which indicate the (cells of) the components and nets that have been *forced* to match.

### Low level descriptions

The low level description file has the same syntax as the mixed level description file. It contains only components with pin permutabilities which are described by pin classes. The low level file generated can be essentially different from the input 'mixed level' description !
If one wants to compare two mixed level netlists with Gemini, these can both be transformed using STRICT. The transformations have the property that if, and only if, two mixed level netlists are equivalent, their corresponding STRICT output is also equivalent.

### Low level type definitions

The low level type definitions file contains type definitions which use only terminal classes to describe the pin permutabilities of the component types. See the previous paragraph.

# Chapter 5

# Results

This chapter discusses the performance of the netlist comparison program STRICT. In order to test the program, pairs of reference and test netlists with varying sizes are verified. Furthermore, discrepancies are introduced. The accuracy of the discrepancy reports generated by STRICT and Gemini are compared to each other. First the test approach is discussed, after this the effects of removing components and interchanging nets will be considered. Finally a comment will be made on the comparison of high level designs.

## 5.1 Approach

The aim of the program is to either certify that two netlists are equivalent or to locate the discrepancies between them. There are a number of measures of the quality of the program. These are the following:

- The accuracy of the discrepancy report.
- The time taken to compare two netlists as some function of the netlists.
- The amount of memory used as some function of the netlists.

The last two points are purely quantitative. In order to quantify the accuracy of the discrepancy report, we count the number of extra unmatched components and nets for some given discrepancy. Four pairs of netlists are used to test the program. Each pair consists of a *reference* and a *test* netlist. The *test* netlists have been extracted from layouts. The following two sections consider two causes of discrepancies; missing components in the test netlist and interchanged component terminals in the test netlist. An example of a component with interchanged terminals is a transistor where the net which is supposed to be connected to the gate is connected to the drain, and vice versa.

When presenting the effects of missing components, the actual number of missing components is shown in a column. Also the number of components which are not matched is shown. As discussed in the previous chapters, discrepancies tend distort the construction of a partial isomorphism mapping. As a result of this, the number of unmatched components can be larger than the number of missing components. Thus, some components from the reference netlist are not matched while their proper images are not missing from the *test* netlist. In turn, components which should *not* be mapped onto each other are mapped onto each other (but a warning is given for this).

In this test case, we know which components are missing from the test netlists since they have been removed (randomly) on purpose. Thus it is possible to judge which unmapped components are 'correct'. To do this, first an isomorphism mapping is constructed between consistent reference and test netlists. After this a set of muted test netlists is constructed, with missing components. These test netlists are compared with the reference netlist. The number of components from the reference netlist that are reported as unmatched are counted for each comparison. But the unmatched components from the reference netlist are also checked against the components in the isomorphism mapping that correspond to the components missing from the muted netlists. Using this method it can be asserted that components which are reported as unmatched really have missing image components. The converse is not true since multiple valid mappings may exist between netlists. In order to judge reports of interchanged nets a similar approach is used.

## 5.2 Missing components

This section discusses the performance when reporting missing components. Flat netlists are used so that the STRICT error report statistics can be compared to the statistics of Gemini. Three designs are considered.

### Design 0

The first design to be considered consists of 14318 components. Table (5.1) consists of one column showing the number of components truly missing from the test netlists, and three sets of columns with information regarding the error reports generated by STRICT, STRICT with the value of *user-SuspectTime* set to 2, and Gemini respectively. Each of these sets consists of a column (labeled *a*) with the number of components reported as missing, and a column (labeled *b*) with the number of reported components that are *certainly* missing from the test netlist. As mentioned before, a number in this column which is lower than the number of reported components does *not* have to mean that the reported components are not the right ones. Rather, they are the numbers of reported components that are *certainly* reported properly.

We see that up to 4 missing components are reported with 100% guaranteed accuracy, by both STRICT and Gemini. For 5 missing components, Gemini generates a huge list of 41 unmapped components. STRICT on the other hand only reports 5 missing components (out of which 2 guaranteed correct). At a first glance, STRICT seems to perform far better than Gemini. These results are somewhat deceptive however.

| # missing | STRICT | | -t2 | | Gemini | |
|---|---|---|---|---|---|---|
| | a | b | a | b | a | b |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 1 | 2 | 1 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 5 | 5 | 2 | 5 | 2 | 41 | 4 |
| 6 | 6 | 2 | 6 | 2 | 30 | 4 |
| 7 | 7 | 2 | 7 | 3 | 33 | 5 |
| 8 | 8 | 1 | 8 | 2 | 50 | 5 |
| 9 | 9 | 2 | 9 | 2 | 35 | 4 |
| 10 | 10 | 4 | 10 | 4 | 32 | 5 |
| 15 | 15 | 7 | 15 | 7 | 67 | 8 |
| 20 | 20 | 8 | 21 | 11 | 59 | 14 |
| 25 | 25 | 7 | 25 | 9 | 19 | 7 |

Table 5.1: Effect of missing components on error report. Set 1.

| # missing | STRICT | | -t2 | | Gemini | |
|---|---|---|---|---|---|---|
| | a | b | a | b | a | b |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 3 | 4 | 4 |
| 5 | 5 | 4 | 5 | 3 | 5 | 5 |
| 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 7 | 7 | 4 | 7 | 4 | 7 | 7 |
| 8 | 8 | 4 | 8 | 3 | 8 | 8 |
| 9 | 9 | 5 | 9 | 4 | 9 | 9 |
| 10 | 10 | 5 | 10 | 6 | 11 | 10 |
| 15 | 15 | 9 | 15 | 5 | 15 | 14 |
| 20 | 20 | 6 | 20 | 2 | 59 | 17 |
| 25 | 25 | 5 | 25 | 6 | 34 | 23 |

Table 5.2: Effect of missing components on error report. Set 2.

| # missing | STRICT | -t2 | Gemini |
|---|---|---|---|
| | (min.) | (min.) | (s) |
| 1 | 2:02.5 | 2:01.4 | 6.3 |
| 2 | 2:19.4 | 2:19.8 | 6.3 |
| 3 | 2:03.1 | 2:44.5 | 6.2 |
| 4 | 2:06.6 | 2:13.3 | 6.6 |
| 5 | 1:57.2 | 2:11.7 | 7.0 |
| 6 | 2:43.5 | 3:39.2 | 6.9 |
| 7 | 2:06.4 | 2:02.4 | 7.1 |
| 8 | 2:20.7 | 2:29.2 | 7.0 |
| 9 | 2:06.0 | 2:46.8 | 7.1 |
| 10 | 3:05.7 | 3:05.3 | 7.2 |
| 15 | 5:34.0 | 2:48.8 | 9.0 |
| 20 | 3:53.0 | 3:52.2 | 9.2 |
| 25 | 3:13.5 | 6:34.2 | 8.9 |

Table 5.3: Time taken for comparisons

Table (5.2) shows the effect of removing a different random set of components. We see that up to 9 components are reported with 100 % accuracy by Gemini. Only when 20 components are removed, STRICT seems to perform better.
Before considering the effects of removing components on the error reports for other designs, we look at the run times and memory use. Table (5.3) shows the run times for the different numbers of components (set 1). We see that Gemini has runtimes in the order of seconds (!) whereas STRICT has run times in the order of minutes. The measured memory consumed by STRICT is about 1.3K byte per component for all designs considered (regardless of the value of *suspectTime*). Gemini uses about 500 bytes per component.

| # missing | STRICT | | -t2 | | Gemini | |
|---|---|---|---|---|---|---|
| | a | b | a | b | a | b |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 5 | 5 | 0 | 5 | 3 | 5 | 5 |
| 6 | 6 | 3 | 6 | 2 | 6 | 6 |
| 7 | 7 | 0 | 7 | 0 | 11 | 7 |
| 8 | 8 | 1 | 8 | 2 | 12 | 8 |
| 9 | 9 | 1 | 9 | 1 | 13 | 9 |
| 10 | 10 | 0 | 10 | 0 | 14 | 10 |
| 15 | 15 | 1 | 15 | 0 | 15 | 15 |
| 20 | 20 | 0 | 20 | 1 | 20 | 20 |
| 25 | 25 | 0 | 25 | 1 | 22 | 7 |

Table 5.4: Effect of missing components on error report.

### Design 1

Table (5.4) shows the error report statistics for design 1 which consists of 4727 components. These results illustrate the difficulty of measuring the performance. When 1 component is missing, 1 component is reported but as we see it is not the expected component. This is caused by the fact that the netlist has more than one isomorphism. Obviously the error report for 1 missing component must be correct.

Note that Gemini performs well, even for 20 and 25 missing components.

### Design 2

Table (5.5) shows the statistics for a design consisting of 10782 components. Unfortunately the guaranteed accuracy for Gemini is low.

| # missing | STRICT | | -t2 | | Gemini | |
|---|---|---|---|---|---|---|
| | a | b | a | b | a | b |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 2 | 2 | 2 | 2 | 2 | 2 | 0 |
| 3 | 3 | 3 | 3 | 3 | 3 | 1 |
| 4 | 4 | 4 | 4 | 4 | 4 | 2 |
| 5 | 5 | 5 | 5 | 5 | 5 | 2 |
| 6 | 6 | 6 | 6 | 6 | 6 | 2 |
| 7 | 7 | 7 | 7 | 7 | 7 | 3 |
| 8 | 8 | 8 | 8 | 8 | 8 | 3 |
| 9 | 10 | 8 | 9 | 9 | 9 | 3 |
| 10 | 10 | 10 | 10 | 10 | 10 | 3 |
| 15 | 15 | 14 | 15 | 15 | 15 | 3 |
| 20 | 21 | 18 | 21 | 19 | 20 | 4 |
| 25 | 26 | 24 | 27 | 24 | 25 | 5 |

Table 5.5: Effect of missing components on error report.

## 5.3 Interchanged component terminals

In this section we shall study the effects of interchanging component terminals. The same components are affected as in the previous measurements. Only here, instead of removing the components from the test netlist, their terminal connections are interchanged (rotated). In the ideal case, the netlist comparison programs would indicate exactly which nets have been affected, and the precise causes (connections to a wrong terminal class).

**Design 0**

Table (5.6) consists of one column showing the number of components affected and three sets of columns with statistics for programs and their modes. The column sets for STRICT and STRICT with *suspectTime* set to 2 each consist of 6 columns. They are labeled *a* through *f*. For Gemini only four columns are given. Column *a* shows the number of nets that are unmatched. Column *b* shows the number of these nets that are certainly interchanged. Column *c* shows the number of nets that are reported as problematic. Column *d* shows the number of these nets that are certainly interchanged. For STRICT, column *e* shows the number of nets that are reported as being connected to the wrong terminal class, and column *f* shows the number of nets that are certainly connected to the wrong terminal class. We see that up to 6 affected components are reported with 100% accuracy by STRICT. Note that even though the number of nets that are reported as problematic can become very large (97 nets for 25 affected components), the faults can be located with relatively little effort, because the more specific report concerning the terminal classes is given.

Note that Gemini does not report all the affected nets. Rather it reports the components involved as being problematic. This is not shown here.

The results in table (5.7) show that even when 10 components are affected, STRICT can give an error report which is completely correct.

| # affected | STRICT | | | | | | -t2 | | | | | | Gemini | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | c | d | e | f | a | b | c | d | e | f | a | b | c | d |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 | 0 | 0 | 3 | 3 | 3 | 3 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 5 | 5 | 5 | 5 | 0 | 0 | 5 | 5 | 5 | 5 | 0 | 0 | 2 | 2 |
| 3 | 0 | 0 | 7 | 7 | 4 | 4 | 0 | 0 | 7 | 7 | 4 | 4 | 0 | 0 | 4 | 4 |
| 4 | 0 | 0 | 7 | 7 | 4 | 4 | 0 | 0 | 7 | 7 | 4 | 4 | 0 | 0 | 4 | 4 |
| 5 | 0 | 0 | 10 | 10 | 7 | 7 | 0 | 0 | 10 | 10 | 7 | 7 | 0 | 0 | 6 | 6 |
| 6 | 0 | 0 | 10 | 10 | 8 | 8 | 0 | 0 | 10 | 10 | 8 | 8 | 0 | 0 | 6 | 6 |
| 7 | 0 | 0 | 13 | 13 | 12 | 10 | 0 | 0 | 13 | 13 | 12 | 10 | 0 | 0 | 6 | 6 |
| 8 | 0 | 0 | 30 | 15 | 9 | 9 | 0 | 0 | 30 | 15 | 9 | 9 | 0 | 0 | 6 | 6 |
| 9 | 0 | 0 | 58 | 18 | 17 | 7 | 0 | 0 | 58 | 18 | 17 | 7 | 0 | 0 | 8 | 8 |
| 10 | 0 | 0 | 58 | 20 | 13 | 11 | 0 | 0 | 58 | 20 | 13 | 11 | 0 | 0 | 9 | 9 |
| 15 | 0 | 0 | 40 | 29 | 15 | 6 | 0 | 0 | 40 | 29 | 15 | 6 | 0 | 0 | 11 | 11 |
| 20 | 0 | 0 | 78 | 36 | 15 | 9 | 0 | 0 | 75 | 36 | 20 | 14 | 0 | 0 | 12 | 12 |
| 25 | 0 | 0 | 92 | 45 | 19 | 6 | 0 | 0 | 97 | 47 | 20 | 8 | 0 | 0 | 18 | 18 |

Table 5.6: Effect of interchanged component terminals on error report. Set 1.

| # affected | STRICT | | | | | | -t2 | | | | | | Gemini | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | c | d | e | f | a | b | c | d | e | f | a | b | c | d |
| 1 | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 7 | 4 | 3 | 3 | 0 | 0 | 7 | 4 | 3 | 3 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 9 | 7 | 8 | 8 | 0 | 0 | 9 | 7 | 7 | 7 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 10 | 9 | 10 | 10 | 0 | 0 | 10 | 9 | 10 | 10 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 12 | 12 | 12 | 12 | 0 | 0 | 12 | 12 | 12 | 12 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 13 | 13 | 13 | 13 | 0 | 0 | 13 | 13 | 13 | 13 | 0 | 0 | 2 | 2 |
| 7 | 0 | 0 | 14 | 14 | 15 | 15 | 0 | 0 | 14 | 14 | 15 | 15 | 0 | 0 | 2 | 2 |
| 8 | 0 | 0 | 16 | 16 | 13 | 13 | 0 | 0 | 16 | 16 | 13 | 13 | 0 | 0 | 4 | 4 |
| 9 | 0 | 0 | 18 | 18 | 17 | 17 | 0 | 0 | 17 | 17 | 13 | 13 | 0 | 0 | 7 | 7 |
| 10 | 0 | 0 | 20 | 20 | 12 | 12 | 0 | 0 | 20 | 20 | 12 | 12 | 0 | 0 | 4 | 4 |
| 15 | 0 | 0 | 33 | 31 | 13 | 10 | 0 | 0 | 33 | 31 | 16 | 11 | 0 | 0 | 12 | 12 |
| 20 | 0 | 0 | 54 | 38 | 20 | 14 | 0 | 0 | 54 | 38 | 17 | 9 | 0 | 0 | 11 | 11 |
| 25 | 0 | 0 | 64 | 47 | 28 | 13 | 0 | 0 | 64 | 49 | 44 | 30 | 0 | 0 | 12 | 12 |

Table 5.7: Effect of interchanged component terminals on error report. Set 2.

### Design 1

Table (5.8) shows a set of effects of interchanging component terminals in design 1 .

| # affected | STRICT | | | | | | -t2 | | | | | | Gemini | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | c | d | e | f | a | b | c | d | e | f | a | b | c | d |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 | 0 | 0 | 3 | 3 | 3 | 3 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 5 | 5 | 4 | 4 | 0 | 0 | 5 | 5 | 6 | 6 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 6 | 6 | 9 | 9 | 0 | 0 | 6 | 6 | 7 | 7 | 0 | 0 | 2 | 2 |
| 4 | 0 | 0 | 8 | 8 | 11 | 11 | 0 | 0 | 8 | 8 | 11 | 11 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 12 | 12 | 12 | 12 | 0 | 0 | 12 | 12 | 13 | 13 | 0 | 0 | 4 | 4 |
| 6 | 0 | 0 | 23 | 15 | 12 | 6 | 0 | 0 | 15 | 15 | 12 | 12 | 0 | 0 | 12 | 7 |
| 7 | 0 | 0 | 27 | 13 | 12 | 4 | 0 | 0 | 19 | 13 | 11 | 3 | 0 | 0 | 4 | 4 |
| 8 | 0 | 0 | 29 | 15 | 19 | 5 | 0 | 0 | 29 | 15 | 18 | 4 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 31 | 17 | 17 | 5 | 0 | 0 | 31 | 17 | 17 | 5 | 0 | 0 | 10 | 10 |
| 10 | 0 | 0 | 24 | 18 | 15 | 8 | 0 | 0 | 24 | 18 | 13 | 8 | 0 | 0 | 11 | 11 |
| 15 | 0 | 0 | 47 | 34 | 22 | 12 | 0 | 0 | 45 | 32 | 25 | 11 | 0 | 0 | 20 | 13 |
| 20 | 0 | 0 | 41 | 38 | 29 | 19 | 0 | 0 | 48 | 24 | 24 | 10 | 0 | 0 | 24 | 17 |
| 25 | 0 | 0 | 63 | 50 | 29 | 13 | 0 | 0 | 55 | 46 | 30 | 18 | 0 | 0 | 33 | 26 |

Table 5.8: Effect of interchanged component terminals on error report.

**Design 2**

Table (5.9) shows a set of effects of interchanging component terminals in design 2 .

| # affected | STRICT | | | | | | -t2 | | | | | | Gemini | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | c | d | e | f | a | b | c | d | e | f | a | b | c | d |
| 1 | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 4 | 4 | 2 | 2 | 0 | 0 | 4 | 4 | 2 | 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 7 | 6 | 5 | 4 | 0 | 0 | 7 | 6 | 2 | 2 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 8 | 8 | 4 | 4 | 0 | 0 | 8 | 8 | 4 | 4 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 10 | 10 | 3 | 3 | 0 | 0 | 10 | 10 | 3 | 3 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 12 | 12 | 5 | 5 | 0 | 0 | 12 | 12 | 5 | 5 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 14 | 14 | 9 | 9 | 0 | 0 | 14 | 14 | 9 | 9 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 17 | 16 | 7 | 7 | 0 | 0 | 17 | 16 | 6 | 6 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 18 | 18 | 9 | 9 | 0 | 0 | 18 | 18 | 11 | 11 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 20 | 19 | 13 | 13 | 0 | 0 | 20 | 19 | 15 | 15 | 0 | 0 | 2 | 2 |
| 15 | 0 | 0 | 34 | 32 | 4 | 4 | 0 | 0 | 34 | 32 | 4 | 4 | 0 | 0 | 14 | 2 |
| 20 | 0 | 0 | 69 | 39 | 6 | 5 | 0 | 0 | 72 | 42 | 4 | 4 | 0 | 0 | 14 | 2 |
| 25 | 0 | 0 | 121 | 50 | 6 | 6 | 0 | 0 | 120 | 50 | 7 | 7 | 0 | 0 | 14 | 2 |

Table 5.9: Effect of interchanged component terminals on error report.

## 5.4 High level designs

The designs considered so far were made up of components with terminal permutation sets that could be described using terminal classes. Therefore the results of Gemini and STRICT could be compared. As discussed one of the primary motivations for building STRICT was the fact that Gemini cannot handle designs with components with 'recursive' terminal permutation sets. In order to judge the performance of STRICT for a realistic design (consisting of 3814 high level components), it was matched using STRICT, and a conversion was made to a flat netlist (allowing a comparison using Gemini). The conversion to a flat netlist took 3.0 seconds in total for both the reference and the test netlist.

# Chapter 6

# Conclusions

A method has been described to compare descriptions of electronic designs at equal levels of abstraction in the structural domain. The descriptions were first transformed into a form with components with simple permutation sets. After this the netlists could be compared with traditional methods.

A program named STRICT, which performs both of these functions, has been built. Three main conclusions can be drawn; The first is that the method of converting netlists is an efficient one in terms of the size of the resulting netlists. This has been shown in chapter 2. The second conclusion concerns the verification of gate level netlists (possibly obtained by the method described). The accuracy of the discrepancy reports generated by STRICT is *generally* speaking not significantly better than that of Gemini when components are missing from the test netlist. Interchanged component terminals, however, can be located with more ease. The reason for this is that STRICT gives more specific error messages for interchanged component terminals.
Finally, STRICT uses more memory and is slower than Gemini. This is because the program has not been optimized for performance; a flexible program was built which allowed for experimentation with algorithms.

Clearly STRICT is *not* a replacement for Gemini. Together they form a powerful combination.

# Bibliography


[1] D.G. Corneil and C.C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the Association for Computing Machinery*, 17(1):51–64, 1970.

[2] D.G. Corneil and D.G. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM Journal on Computing*, 9(2):51–64, May 1980.

[3] C. Ebeling. Gemini II: A second generation layout validation program. In *Proc. ICCAD*, pages 322–325, 1988.

[4] C. Ebeling and O. Zajicek. Validating VLSI circuit layout by wirelist comparison. In *Proc. ICCAD*, pages 172–173, 1983.

[5] D.W. Harberts. Improved netlist comparison to characterize differences between electronic circuits. *PHILIPS JOURNAL OF RESEARCH*, 45(2):111–125, 1990.

[6] Ir. D.W. Harberts. The netlist comparison program gemini: Algorithms, modifications and analysis. Technical Report 6325, Philips Research Laboratories, 1988/1989.

[7] J.A.G. Jess, M.J.M.Heiligers, M.R.C.M. Berkelaar, J.F.M. Theeuwen, G.L.J.M Janssen, H.A. Hilderink, and A.H. Timmer. *Collegedictaat C.A.D systemen*. Eindhoven University of Technology, The Netherlands, 2nd edition, 1994.

[8] A.P. Kostelijk. *Verification of electronic designs by reconstruction of the hierarchy*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1994.

[9] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill, Inc., 1994.

[10] G. Pelz and U. Roettcher. Circuit comparison by hierarchical pattern matching. In *Proc. ICCAD*, pages 290–293, 1991.

[11] G. Pelz and U. Roettcher. Pattern matching and refinement hybrid approach to circuit comparison. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF ITEGRATED CIRCUITS AND SYSTEMS*, 13(2):264–276, February 1994.

[12] R.C. Read and D.G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

[13] W. Rovers, Private communication, 1995.

[14] A. van der Veen, Private communication, 1995.