Department of Computer Science
University of Manchester
Manchester M13 9PL, England

Technical Report Series
UMCS–94–4–2

Salvador Mir

HEURISTIC REASONING For An AUTOMATIC COMMONSENSE UNDERSTANDING Of LOGIC ELECTRONIC DESIGN SPECIFICATIONS

HEURISTIC REASONING
For An
AUTOMATIC COMMONSENSE UNDERSTANDING
Of
LOGIC ELECTRONIC DESIGN SPECIFICATIONS [1]

Salvador Mir

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.

Contents

Abstract

An automatic *heuristic understanding* of digital electronic design specifications is discussed in this thesis. The term understanding is used in the sense that knowledge about the functionality and purpose of the cells and signals of a design is abstracted away from the specification. An heuristic analysis which exploits implicit design semantics is carried out. The analysis bypasses the examination of detailed logical and electrical data since it is aimed at the machine simulation of an heuristic way of understanding electronic design specifications exhibited by human experts. By overlooking implicit knowledge, current automatic systems are clearly at a disadvantage with respect to human experts for the analysis, design and management of electronic data. The possibility of getting the machine to heuristically understand a specification is seen in this thesis as one way of improving this situation.

The thesis defines and classifies expert knowledge about digital electronic designs, explores ways to generate expert knowledge about a design from the heuristic analysis of its specification and discusses examples of exploiting this knowledge to plan and control automatic tasks. The experimental result of this research is a knowledge-based system aimed at providing an empirical demonstration of the convenience and viability of an automatic heuristic understanding of design specifications. The method of reasoning of the system has without question limitations, but these are also faced by human experts when they attempt this task. The current prototype of the system already indicates that valid knowledge can be generated and it implements methods for avoiding the critical computational complexity of the problem.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Preface

The author graduated from Universitat Politècnica de Catalunya, Barcelona, in July 1987 with a degree in Industrial Engineering. He gained a degree of Master of Science at the University of Manchester in October 1989. He then commenced full-time research work for the degree of Doctor of Philosophy at the University of Manchester. The research done during this period is described in this thesis.

Acknowledgements

This work evolved from the initial proposal of my supervisor, Dr. Nicholas Paul Filer, of exploiting semantic knowledge contained in an electronic design specification. I am grateful to him for this initial proposition and for his guidance and encouragement to go ahead with the work. I am also grateful to him for many corrections and comments which improved the legibility of the manuscript.

I am grateful to my family for their encouragement to get on with my life and my work, and for all those wonderful reasons which obliged me to frequently go back home to keep pace with many things that changed over there during the research time. This thesis is dedicated to my parents.

I am grateful to friends and colleagues who made the research time not only enjoyable but also possible by granting me on countless occasions the chance of enjoying their company. Pepi Aguila, Maria Teresa Arica, Jorge Artiles, Marisa Barañano, Karen Cooksey, Alain Deckers, Catherine Dubois, Amr Elleithy, Wael Fahmi, Luisa Ferriz, Ignacio Ferriz, Julio Garrido, Aurelio Gómez, Teresa Guedes, Jeremy Heald, Mark Hendrick, Choi Lin Lee, Katia Helena Lipp João, Ana Cristina Melo, Francesca Montalcini, Lidia Moraes, Georgina and Antonio Noriega, Iñaki Onandia, Maria Jose Pardo, Mouna Salem, Laura and Paolo Saviotti, Bahram Semsar Zadeh, Eduarda and Fernando da Silva, Georges Theodoropoulos, Georges Tsakogiannis, Eva Valero, Martine de Vlieger, Daniel Wray and Tim and Hilary Young all were, among others, very good company.

Colleagues in the CAD laboratory at the University of Manchester were useful in a number of ways. Daniel Wray produced a graphical interface for displaying the heuristic design knowledge generated. He patiently coped with several updatings of the scheme for the representation of knowledge. Discussions with him, and Jun-Kang Feng and Alan Williams, who worked on a formal representation of the heuristic knowledge generated, were useful to clarify the notation of the knowledge representation scheme. Discussions with Mike Brown helped me to understand the requirement for a formal representation of heuristic design knowledge. I am grateful to many colleagues in the laboratory who always helped me with the use of the computer system.

Discussions with Helena Mendes currently in the Department of Computation at UMIST in Manchester were of substantial interest to emphasise the applicability of this research to *reverse engineering* activities.

I am grateful to Mike Brown, Karen Cooksey, Alain Deckers, Laura Saviotti, Tim Young and Daniel Wray for proof-reading the manuscript.

This work was supported by a grant from the *Comisión Interministerial de Ciencia y*

Tecnología (CICYT), Madrid, to whom I am greatly indebted for the financial support provided over the research years and for the promptness and understanding demonstrated when dealing with me as a grantee.

Abbreviations

AI	Artificial Intelligence
ALU	Arithmetic-Logic Unit
Av	Average
CAD	Computer-Aided Design
CAM	Content-Addressable Memory
CIN	Control Input
Ck, Clk	Clock
CM	Control Memory
COMP	Comparator
COUT	Control Output
CP	Control Processor
CPU	Central Processor Unit
CS	Combinational System
CTR	Control
D	Delay
DBMS	Database Management System
DDL	Data Definition Language
DFT	Design-For-Testability
DIN	Data Input
DKB	Deep Knowledge-Based
DMA	Direct-Memory Access
DML	Data Manipulation Language
DMM	Design Methodology Management
DOUT	Data Output
ECAD	Electronic CAD
EDIF	Electronic Design Interchange Format
EX	Execution
FF	Flip-Flop
IC	Integrated Circuit
ID	Instruction Decoding
IF	Instruction Fetch
I/O	Input/Output
JK	JK Flip-Flop
$\mathcal{K}$-ext	Knowledge-extraction
$\mathcal{K}$-gen	Knowledge-generation
$\mathcal{K}$-pro	Knowledge-propagation
KB	Knowledge-Based
KBS	Knowledge-Based System
M	Memory
MOS	Metal-Oxide Semiconductor
MUX	Multiplexer
$n^{\underline{o}}$	number
n/a	not applicable
OF	Operand Fetch
OP, OPER	Operator
P	Processor
PE	Processing Element or Port Electronic Function
PG	Port Generic Function
PLA	Programmable-Logic Array
PROM	Programmable ROM
PSA	Programmable-Sequential Array
RA	Register Array
RAM	Random-Access Memory
REG	Register
ROM	Read-Only Memory
RTL	Register-Transfer Level
SE	Signal Electronic Function
SG	Signal Generic Function
SH	Shifter
SR	Set-Reset
STO	Storage
T	Toggle
VLSI	Very Large Scale Integration

Symbols

A_i .. confidence/complexity pair for C_i
$A_{i,j}$ confidence/complexity pair for $C_{i,j}$
As_i .... candidate sets for S_i
C_c content matching $n^{\underline{o}}$ combinations
C_c^* ...... valid $n^{\underline{o}}$ combinations for C_c
C_i .... i-th cell in the design
$C_{i,j}$ .... j-th sub-cell of C_i
Cs .... set of cells of a design
CO_i .... complexity of C_i
CO_i^c .... calculated complexity of C_i
CO_i^* .... estimated complexity of C_i
$CO_{i,j}$ .... complexity of $C_{i,j}$
$CO_{i,j}^*$ .... estimated complexity of $C_{i,j}$
∂ .... partial derivative
d_k .... decision factor for the k-th slot
dx .... differential of x
Dg .... cell dependability
D_i .... decision factor for C_i
$D_{i,k}$ .... decision factor for $C_{i,k}$
Ds_i .... set of down-dependencies of C_i
e_{av} .. av. confidence in selected models
e_i .... confidence in M_i
$e_{i,j}$ .... confidence in $M_{i,j}$
E_i .... confidence in S_i
$E_{i,k}$ confidence in situation defining $C_{i,k}$
EH_i .... level of abstraction of C_i
E_1 .... design model evaluation
$\mathcal{E}$ .... effectiveness of set selection
$\mathcal{E}_{ext}$ .... k-extraction effectiveness
$\mathcal{E}m_k$ .. k-cycle av. new matches per plan
f_c .... plan matching $n^{\underline{o}}$ combinations
f_g .... $n^{\underline{o}}$ plans from model matching
f_k .... k-th cycle $n^{\underline{o}}$ solution plans
fm_k ...... k-th cycle $n^{\underline{o}}$ available plans
Fe_k k-th cycle % already existing plans
Ff_k .... k-th cycle % repeated plans
$F_{i,k}$ .... C_i-to-$C_{i,k}$ arc decision factor
h .... $n^{\underline{o}}$ of class models in the system
h_i .... depth-level of C_i
h_{max} .... largest depth-level
H_k .. k-th heuristic model in the system
HR ... typical interlevel complexity ratio
in .... list membership
I_c . interface matching $n^{\underline{o}}$ combinations
I_c^* .... valid $n^{\underline{o}}$ combinations for I_c
I_e interface name matching effectiveness
I_i .... $n^{\underline{o}}$ of instances of C_i
$I_{i,j}$ .... $n^{\underline{o}}$ of instances of $C_{i,j}$ in C_i
Ir ...... instantiation rate of the design
K .... $n^{\underline{o}}$ of reasoning cycles
K_{av} .... av. $n^{\underline{o}}$ productive cycles per cell
L_{av} .... av. $n^{\underline{o}}$ cell paths
L_k ..... k-th logged model in the system
m_{av} .... average $n^{\underline{o}}$ of models per cell
m_{av_i} .. average $n^{\underline{o}}$ of models per cell in S_i
m_g .... $n^{\underline{o}}$ of model sub-cell classes
m_i .... $n^{\underline{o}}$ of possible models for C_i and $n^{\underline{o}}$ of model input ports
m_o .... $n^{\underline{o}}$ of model output ports
mod .... modulo operator
M_i ...... heuristic model selected for C_i
$M_{i,j}$ ..... heuristic model selected for $C_{i,j}$
M_i^j .... j-th heuristic model for C_i
Ms .... solution set
Ms_i .... solution set for S_i
n .... $n^{\underline{o}}$ of cells in the design
n_{ci} ..... $n^{\underline{o}}$ of model control input ports
n_{co} .... $n^{\underline{o}}$ of model control output ports
n_{di} .... $n^{\underline{o}}$ of model data input ports
n_{do} ...... $n^{\underline{o}}$ of model data output ports
n_i .... $n^{\underline{o}}$ of plan input ports and $n^{\underline{o}}$ of sub-cells of C_i

n_{io}	nº of plan inout ports
n_o	nº of plan output ports
N	nº of candidate solution sets
N_{av}	av. nº situation sets
Nf	nº of failed situation sets
Nf_i	nº of failed sets for S_i
N_i	nº candidate sets for S_i
Nm	nº of cell models generated
Nm_{av}	av. nº of models generated per cell
Np	nº of cell plans generated
Np_{av}	av. nº of plans generated per cell
N_w	maximum nº of candidate sets
NF	nº of failed sets
NM_c	cell name matching effectiveness
NM_n	net name matching effectiveness
NM_p	port name matching effectiveness
p	nº primitive cells in the design
p_g	nº of plan sub-cell classes
p_i	nº of plan input signals
p_o	nº of plan output signals
P_{av}	av. nº cell ports
P_i^j	j-th knowledge plan for C_i
r_{av}	av. complexity deviation factor
r_i	complexity deviation factor of C_i
$r_{i,j}$	weighting factor for $C_{i,j}$
r_k	relative importance of k-th slot
$R_{i,j}$	relative importance of $C_{i,j}$
$R^*_{i,j}$	estimated value of $R_{i,j}$
s_i	nº of situations using C_i
S_i	i-th situation in a design
T	processing time
T_{av}	av. processing time per cell
T_c	% of correctly defined models
T_d	% of fully defined models
TC_i	typical complexity of C_i
u_i	uncertainty about M_i
ud_i	nº of up-dependencies of C_i
U_i	uncertainty about S_i
Us_i	set of up-dependencies of C_i
v_k	confidence in the k-th slot
$V_{i,j}$	model deviation in terms of $I_{i,j}$
$w_{i,k}$	weighted contribution of $C_{i,k}$
w_k	weighted contribution of k-th slot
$\underline{x}$	vector of values
$\underline{x_i}$	i-th sub-vector of vector $\underline{x}$
X	unknown value
$\circ$	composition of functions
$**$	power operator
$//$	integer division
Z *is* X	arithmetic expression X evaluates to Z
$Z =:= X$	numeric values are equal
$Z =\backslash= X$	numeric values are not equal
Z, X	Z and X are true
$Z; X$	Z or X are true
$P \rightarrow Q; R$	if P then Q true else R is true
$\prod$	continued product
$\sum$	continued sum
$\forall$	for all
$\in$	set membership
$\in \{a^+\}$	sub-set membership for ordered sets (sub-set from value a on)
$[\ldots]$	list of values
$\{\ldots\}$	set of values
As–$[q]$	q-th element in list As
a–b	range of values
$As \Rightarrow Bs$	data transfer from As to Bs

List of Tables

List of Figures

Als meus pares

"La majoria dels homes són uns ingenus amb molta llana al clatell: s'imaginen que el món és tal i com ells el capten. Tingues sempre en compte però, que en realitat el món no té cap forma ni color, que cada ésser viu el percep d'una manera diversa, segons la naturalesa dels propis sentits". Pep Coll, *La Mula Vella*

"Todo pasa y todo queda,
pero lo nuestro es pasar,
pasar haciendo caminos,
caminos sobre la mar".
Antonio Machado

"Orr would be crazy to fly more missions and sane if he didn't, but if he was sane he had to fly them. If he flew them he was crazy and didn't have to; but if he didn't want to he was sane and had to". Joseph Heller, *Catch 22*

"Pour un jour de synthèse il faut des années d'analyse".
Fustel de Coulanges

Part I

Expert Knowledge and Problem Formulation

Chapter 1

Introduction

The empirical observation that human experts can reason about electronic data in a way that automatic systems are unable to do triggered the research work presented in this thesis. The exploitation of heuristic design knowledge allows human experts a flexible way of understanding logic electronic design specifications which facilitates the analysis, design and management of electronic data. A procedure for the machine simulation of an heuristic way of understanding design specifications exhibited by human experts is introduced. The research work is within the area of applied Artificial Intelligence. Current research activities which can be related to this work include the re-engineering of software specifications and efforts directed to automatically reasoning about physical systems and man-made devices. The major contributions of this work are enumerated and an overview of the rest of the thesis is presented.

1.1 Motivations of the Research Work

The research work presented in this thesis is aimed at providing the groundwork for an automatic *heuristic understanding* of digital electronic design specifications. The heuristic understanding of an electronic design is based on the analysis of design semantics implicit in the specification. The analysis bypasses the detailed logical and electrical data which can rigorously describe the behaviour of a system and its components (electronic cells). The term understanding is used in the sense that knowledge about the functionality or purpose of a design and its parts is abstracted away from its specification and this knowledge can be exploited later in order to plan and control automatic tasks. It is the exploitation of implicit design semantic knowledge, with the intention of improving the performance of computer-aided design (CAD) tools, which motivated this research work.

The need to exploit design semantics stems from the empirical observation that human experts can reason about electronic data in a way that automatic systems are unable to do. Human experts can often gain an overall understanding of the intention or purpose behind a design and its parts from the interpretation of data contained in its specification. This understanding allows the experts flexible ways of reasoning in order to efficiently plan and control design tasks. On the other hand, most automatic design tools process design data blindly according to the heuristics and algorithms they are programmed to use. The electronic data must meet the requirements of the CAD tools used and no intelligent machine interpretation of the data takes place. This is seen as one of the reasons why human experts can find solutions to problems that CAD tools are often unable to reach. The possibility of getting the machine to heuristically understand the circuit that is being designed (or that is being analysed) is seen in this thesis as one way of narrowing the gap between human experts and automatic systems with respect to the processing of electronic data.

The human understanding of an electronic specification is not only based on the study of the rigorous behaviour of the design but also on the examination of implicit design semantics. The study of electrical and logical data which precisely describes the behaviour of the design and its cells is very laborious and time consuming even for designs of a modest complexity. Without doubt, human experts are able to abstract higher level models of these behaviours which allow more flexible ways of reasoning than those based on the mathematical data. This flexibility comes, in most cases, from the fact that the knowledge which forms these models, and which is abstracted away from the specification, is generally quite vague and imprecise in nature. For example, some of this knowledge allows the experts to categorise the functionality of a device and its parts. But often, the principles behind these categorisations are quite ill-defined. As a result, the knowledge obtained has low precision and resolution. It is also for this reason that this knowledge cannot be made more explicit in an electronic specification attending to the strict formalities of hardware description languages. As a consequence, in hardware descriptions, as in most types of specifications (e.g. computer programs), there are implicit semantics.

In the case of hardware descriptions, these implicit semantics often relate to the functionality or purpose of the design and its parts. When these implicit semantics are interpreted by an expert it is often possible to reach sensible conclusions which can be both explained and justified. Human experts make use of their experience and common sense in order to first analyse those semantic or heuristic aspects and data which appear most relevant for the understanding of a specification and which allow them to bypass the exploration of the most complex data. This leads to natural and pragmatic methods for the understanding of a specification. The usefulness of this kind of analysis is evident when an expert is able to understand the operation of a design in a short span of time.

This work is concerned with the machine simulation of an heuristic way of understanding electronic design specifications exhibited by human experts. The thesis defines and classifies expert knowledge about digital electronic designs, explores ways to generate expert knowledge about a design from the heuristic analysis of its specification, and discusses examples of exploiting this knowledge to plan and control electronic CAD (ECAD) tasks. It is hoped that by capturing this kind of knowledge about a design the machine will be able to process the electronic data in a way that comes closer to the ways exhibited by human experts. The experimental result of this research is a knowledge-based system (KBS) aimed at the exploration of the architecture and functionality of a digital electronic design by means of an heuristic analysis of its specification. The system developed is called HERCULES (*HEuristic Reasoning for an automatic Commonsense Understanding of Logic Electronic design Specifications*).

1.2 Exploiting Implicit Design Knowledge

A specification of an electronic design contains full or partial information about the structure and behaviour of the electronic circuits represented. The designers include in the description the data required by CAD systems for the processing of the design. In addition, the designers usually specify this data in such a way that other human designers can understand the design with a reasonable amount of effort. It is clear that a bare description of an electronic design in terms of the electrical or logical behaviours of its components and their interconnections is, in most cases, too extensive and elaborate for anybody to readily understand. For this reason, there are implicit semantics in the description of an electronic design. These implicit semantics improve the quality of the specification in the sense that they facilitate its understanding.

Some similarities exist with the specification of computer programs. In any programming language, the choice of meaningful names for variables and procedures, the use of comments, the organisation of data structures or the modularity of the program highly facilitate the task of understanding the goal of a program. This extra information is meaningful to the programmers, but it is not strictly required by the system that executes the program. The computer system will generate the same results with a well specified program as with a program with the same goal that is incomprehensible for program designers.

Similarly, the specification of an electronic design can be organised in such a way that facilitates its understanding. Some of the main techniques that are used to improve the quality of a description include the choice of meaningful names for the design objects (e.g. the names of cells and the ports of these cells), the use of comments, the arrangement of the design objects into arrayed structures (e.g. arrays of ports and arrays of signals) and the use of an adequate design hierarchy. All this information is meaningful to design engineers but most of it is unintelligible for automatic systems. Indeed, an automatic system must result in the same product from the processing of a well specified design as from the processing of a bare description, with the same goal, which is beyond human grasp.

The analysis of implicit semantics allows human experts to capture information of an heuristic nature that can be used, in many cases, to understand the functionality or intention of the cells and signals of a design. This process is initially based on the comparison of knowledge captured by interpreting electronic data contained in the specification with knowledge that the experts possess about electronic design. As a result, intuitive ideas or abstract models of the functionality or intention of the objects of a design are generated. The experts can next test and refine these ideas taking into account the context in which each object appears. This is because the objects of a design have relationships between them. For example, since an object is usually composed of other objects (e.g. a cell usually has a set of ports and a set of sub-cells), knowledge about an object must be consistent with knowledge about its parts. This provides a way of strengthening conclusions. These relationships can also be used in a more operational way. That is, they imply that knowledge about an object is useful for the inference of information about the parts of this object and vice versa. Other relationships between objects are imposed, for example, by the interconnections between cells and by the flow of signals in the circuit.

As an example, consider the identification of a device that represents an $n-$bit register. The analysis by means of the electrical or behavioural data (if available) is not straightforward since many variants of these devices exist. Alternatively, design semantics can be exploited. Examples of heuristics that can be used for the identification of a register include the name of the device, the device being constructed from an array of memory cells, the existence of clock ports, load ports and input and output data ports of the same width n, and the use or location of this device in a design (such as connected to the data output of an arithmetic-logic unit or ALU since a register is usually placed there). The identification of the clock port, for example, could be achieved by the analysis of the name of the port, by its connection to other external or internal clock ports, by the type of the signal carried, or by the dimension of the port (since other ports of the cell, such as the data ports, are generally larger).

The analysis of implicit design semantics provides, undoubtedly, an easy way to reach conclusions. Clearly, the nature of these design semantics is different from that of the electrical and logical data. The intuitive ideas which are formed about the objects of a design represent a higher level of abstraction than the level of abstraction of the mathematical data. As a result, more flexible ways of reasoning are possible which allow many unwise alternatives to be rapidly discarded. A penalty to pay with the analysis of this kind of data is a degree of uncertainty associated with the knowledge obtained (since the rigorous logical and electrical data is avoided). This is often compensated by the simplicity with which results can be attained.

In any case, the knowledge acquired can be adjusted or refined with the study of the more detailed electronic data. This study can be postponed until it is strictly necessary (e.g. when this type of knowledge is not sufficient to carry out a task or support an hypothesis) and it can be guided by the heuristic information already generated. As a matter of fact, human experts often subordinate the study of the elaborate mathematical information to the analysis of design semantics. Of course, a system that exploits only these semantics cannot understand most of the detail behind a hardware description. However, it is convenient for the capture of important design characteristics and for the presentation of the design with semantic information that can be useful for CAD tools (and for human experts too).

1.3 Methodology

A procedure for the machine simulation of the heuristic way of reasoning outlined above is discussed in this section. The procedure must exploit design semantics which are implicit in the description of the design. As discussed in the above section, human experts initially obtain intuitive ideas about the possible purpose or operation of some of the objects of a design. This is mainly based on the resemblance of semantic information to knowledge about electronic design that the experts possess. Next, they repeatedly test and refine these ideas after studying the context in which each object appears.

In fact, this way of reasoning is typical of the manner in which humans analyse different situations of the real world. The initial extraction of ideas about parts of a situation is mostly an intuitive process based on the observation of some *characteristic* information. These ideas can be seen as models abstracted from the actual reality. Next, these ideas are reorganised or modified according to the relationships between different objects. A clear example is the observation of a picture. In its simplest form, a picture drawn by a child generally contains a poor representation of a real world situation, but it allows a viewer, in most cases, to form a mental idea of the situation that the picture attempts to represent. Some of the objects in the picture will be more or less obvious. Other objects may not be clear at all, but their contexts in the picture may throw light on their intentions. The picture may be indeed simple, but the little information available often characterises quite well the real situation. For a computer, it ought to be easier to realise the real world situation from the analysis of the naive picture than from the analysis of a more elaborated picture in which most of the characteristic information may well be hidden by the large amount of data. Because the process is only driven by characteristic information, this way of reasoning reflects expertise and common sense rather than a systematic way of approaching the solution.

For the understanding of an electronic design, the system must initially capture a set of arrangements or plans for the heuristic knowledge about its cells (and its signals). These *knowledge plans* are derived from the interpretation of semantic information contained in the designer's specification. For a cell, knowledge plans can be extracted from the definition of the cell (cell object) or from each use of a cell in the design (instance objects). The functions used for the extraction of these plans are called *knowledge-extraction functions*. Examples of these functions are based on the analysis of the names of the objects, on the study of the size of the objects, on inferences drawn from the grouping of objects and on relationships derived from the hierarchy of the design.

Plans of the knowledge about a cell are compared with system information. Examples of system information include knowledge about classes of electronic cells typically used in electronic circuits and knowledge about electronic cells obtained from the processing of previous designs. The successful comparison of a plan for a cell with system information results in candidate *heuristic models* for the cell. The generation of a model for a cell implies that knowledge available in a plan is validated by the system. The model includes information about the functionality of the cell and its ports. The functions used for this comparison are called *knowledge-generation functions* since a model can contain additional knowledge provided by the system. This is supported by the fact that the items of knowledge existing in the initial plan are known to the system as 'usually' being associated with some other items of knowledge. In this sense, a model is an augmented plan (i.e. a plan with additional knowledge) with respect to the initial plan. A number of different model candidates may be possible for any cell since, for example, different plans can be initially formed.

Consistency must be kept between the models of the cells. This is because the cells and signals of a design have relationships between them as discussed in the previous section. Since a number of models may be possible for each cell, the system must search for a set of consistent models (*solution set*) which 'best' represents the cells of a design (considering one model for each cell of the design). The consideration of all possible combinations of models for the cells of a design leads to a combinatorial explosion (the search space grows exponentially with the number of cells in the design). Even for designs with just a few cells and several candidate models for each cell it will not be feasible in general to consider an exhaustive search for the 'best' solution set. Because of this, more heuristic mechanisms must necessarily take over in order to select the 'best' candidate set which is to be considered first without having to generate all the possible sets. These heuristic mechanisms are based on a pseudo-probabilistic measure of the confidence in each model and in the knowledge represented. The evaluation measures of the models for each cell are combined to give a pseudo-probabilistic measure of the confidence in the overall solution set.

The need for consistency between the models selected for each cell may lead to the addition of further heuristic knowledge in the models of interrelated cells and signals or to the discarding of an alternative combination of models for the cells of the design. The functions used for this examination are called *knowledge-propagation functions* since knowledge about an object can essentially be propagated from interrelated objects. Examples of these functions are based on the analysis of the connectivity of the cells, on the examination of the hierarchy of cells, and on the analysis of the flow of signals in the design. The addition of knowledge to a model of a cell results in a new plan for this cell. In the worst case, the study of each valid combination of alternative models could result in a new set of plans for each one of the cells of the design. A new set of plans can again be compared with system information to generate an additional number of new models. Broadly speaking, this starts a new *knowledge-generation/knowledge-propagation cycle* (reasoning cycle) and the process is repeated until no more new models can be obtained for any of the cells of the design from the best solution set hitherto generated.

In short, the task of the system is the generation and selection of models for the cells (and signals) of a design in a way that avoids the combinatorial explosions which can arise. The functions for the derivation of knowledge can be classified in three groups. The first two, knowledge-extraction and knowledge-generation functions, are aimed at recognising the different cells of the design and they are said to perform *recognition-targeted reasoning*. They simulate the human ability of abstracting models for the design cells. The third group of functions, knowledge-propagation functions, are aimed at studying the consistency between the models of the different cells of the design. These functions simulate the human ability of analysing the relationships between the different cells and the signals which interconnect them and they are said to perform *model-based reasoning*. In addition, the system is controlled by functions which allow the selection of models and the evaluation of the confidence in the knowledge represented in a pseudo-probabilistic way.

1.4 Artificial Intelligence

Most of the research presented in this thesis has been developed within the framework of knowledge engineering and applied Artificial Intelligence (AI). AI researchers bear in mind the initial purpose of AI of creating a computer which thinks or, at least, can simulate human mental faculties by means of computational models. After several decades of research, the feasibility of creating such machines remains very controversial. Advocates and skeptics of AI still plunge into passionate discussions about the rationale or absurdity of the project [Pen89]. This is not surprising since the question touches upon deep issues of philosophy. For instance, what does it mean to say that a human thinks or understands something? With clear answers to questions like this, the end of the AI debate would perhaps not be a long way off. However, these answers have yet to come.

Whatever the outcome of the AI debate is, the conviction has grown that this technology provides worthwhile solutions in a diverse range of problem domains. AI has branched into a number of different areas — vision, natural language, robotics, planning, learning, expert systems — which correspond to diverse intelligent human activities. All these areas, though probably still in an embryonic state of development, already provide technological solutions for current industrial problems. From a quick look at these areas of research and the results obtained, one important point immediately arises: human beings can easily perform actions which, when simulated in a computer, involve a vast body of knowledge with complex interactions. Enthusiastic AI supporters are not impressed by this. For example, the fact that humans can see in the twinkle of an eye what requires millions of computations in a computer is being attributed to evolutionary fortuities which have specially prepared human brains to perform these types of tasks. Thus, the important point behind the impressive human faculties is not only the biological structure of human brains that supports the realisation of these actions but also the actual way in which humans achieve them. Here lies the central dogma of AI. If what the brain does can be thought of, at some level, as a kind of computation, the AI project should succeed.

Encouraged by this, AI researchers have been very concerned with making machines *understand*, specially in areas such as natural language. Of course, the grounds on which it would be possible to defend a claim that a machine understands anything are not yet clear (and, most likely, neither are the arguments required to impute human understanding!) [Jac90]. However, some of the grounds that seem necessary for this understanding to occur have been less questionable and they have formed an important part of AI research. These grounds are taken into account in this work to measure the ability to heuristically understand the specification of a design. They include:

i – the power to represent knowledge about the domain and the ability to use it effectively.

ii – the ability of the machine to perceive equivalences or analogies between different representations of the same or similar situations.

iii – an ability to learn in some non-trivial way. This usually involves the integration of new information with already existing knowledge, perhaps in a way that modifies both.

The early periods of AI research are characterised by the attempt to solve problems which require these kinds of computer understanding. A main topic of research of this period corresponds to the search for general purpose problem-solving methods for large classes of problems (such as those that can be characterised as search problems [CM85]). Day after day, researchers have gained more insight about the actual ways in which humans solve problems. They have realised that general methods of problem-solving underestimate the domain-specific knowledge and common sense that human experts possess. As a consequence, the period of AI research that stretches from the second half of the 1970s to the present day has been characterised by a greater orientation towards solving domain-specific problems [Jac90]. This period of research has put emphasis on the acquisition and representation of the relevant domain knowledge which a program accesses, rather than on the actual way in which the program is executed. The knowledge of the program or *knowledge base* is kept separate from the code that executes the program or *inference engine*. These knowledge-intensive programs are referred to as knowledge-based (KB) systems [1]. The process of constructing these systems is often called knowledge engineering and is considered to be *applied AI*.

Many knowledge-based CAD tools have appeared during the last decade for tasks as diverse as diagnosis and hardware synthesis. The use of KB techniques for CAD has been very much encouraged by the perceived success of the application of this technology to a diverse range of domains, including organic chemistry, mineral exploration and internal medicine. In these domains, KB systems show, in general, quite satisfactory levels of performance when large bodies of expert knowledge are encoded in rule-based knowledge bases [HR85]. This technology has not been so successful in the domain of CAD for very large scale of integration (VLSI) designs, despite the large wealth of knowledge required for the design of these circuits and the complexity of the design process [RKCM85]. A major reason for this is to be found in the different nature of the expert knowledge required in this domain. KB systems appear to work best in domains where there is a substantial body of empirical knowledge connecting situations to actions, but there is little information about the causal mechanisms underlying the case investigated. For example, the rules of knowledge would typically reflect empirical associations of facts derived from experience instead of a theory of the way in which a device or an organism under investigation actually works. A deeper representation of the domain, in terms of spatial, temporal or causal models, is often avoided, or considered unnecessary. Thus, the knowledge used in these KB systems is often called *shallow* as opposed to what is sometimes called *deep* knowledge.

Shallow KB systems reason about a problem with a very limited understanding of the domain. They are characterised by a sparse or bare representation of knowledge. For example, a rule such as "**if** a cell is a register **then** the cell has memory" states that any electronic cell which operates as a register has memory. With such a sparse representation there are in general few possibilities in reasoning. For example, the matching of a condition (such as the cell operating as a register) to pre-defined alternatives must be

[1]KB systems are aimed at the representation and manipulation of domain knowledge, though no actual reasoning may take place. KB systems which have the ability to reason about the knowledge represented are called expert systems [Jac90]. For the sake of generality, the term KB system is used in this work to refer to all these systems.

exact. Most important of all, a problem with shallow systems is that they tend to represent only individual items of knowledge. For example, using this kind of representation it is not easy to represent the requirements for a generic electronic cell to have memory (specially if this is due to the way in which its sub-cells are interconnected). Instead, a specific-to-case approach is taken and the reality that a register cell has memory is simply stated for this particular case of electronic cells.

Deep knowledge-based (DKB) systems do not just have a single level of representation. A comparison of the most important features of these systems against their shallow counterparts is shown in table 1.1 [Mor90]. Layers of concepts are used in the representation of knowledge which provide a much richer structure and consequently a more sophisticated reasoning capability. The use of layers of concepts allows the consideration of general classes in deep systems as opposed to specific instances in shallow systems. In general, DKB systems provide more abstract ways of reasoning about a problem since the kinds of knowledge used are vague and imprecise in nature. On the other hand, shallow-based systems have a more precise representation of knowledge since they are aimed at individual items of information. However, this limits the reasoning capability of these systems. Finally, the control of deep systems is more complex because of the enhanced reasoning capability.

Features	**Shallow**	**Deep**
Levels of Concepts	single level	multiple levels
Structure	simple	complex
Distinctness	specific instances	general classes
Precision, resolution	high	low
Reasoning, representations	sparse	rich
Control	less	more

Table 1.1: Shallow and Deep Systems

A DKB system is more adequate in this work for the representation of knowledge about electronic cells since layers of concepts are required. For example, it must be possible to explicitly link knowledge about the functionality of a cell with knowledge about its parts (i.e. its ports, sub-cells and signals). Classes of electronic cells can also be represented in a more natural way instead of only considering specific instances. Knowledge representation techniques derived from AI (such as frame-like structures [FK85] and semantic networks [Woo75]) are used for the representation of knowledge about electronic cells and classes of electronic cells. Precise and quantitative electrical and logical data is avoided and vague and imprecise knowledge of an heuristic and qualitative nature is used instead. With the consideration of this kind of knowledge, it is expected to move towards a reasoning system that can demonstrate more human capabilities for the understanding of design specifications.

The system presented in this thesis works basically by means of what is called *heuristic classification* [Cla85]. Heuristic classification is recognised as a particular problem-solving methodology in AI. A wide range of KB system tasks in different domains appear to be performed by means of heuristic classification. In this work, the heuristic classification of the electronic cells and signals of a design follows three basic steps:

1. *data abstraction*: this implies the observation of salient features of the cells and signals of an electronic design to form high level interpretations of their operations. The abstraction of heuristic knowledge facilitates the match between electronic data and pre-defined categories.

2. *heuristic matching*: this allows the generation of a number of alternatives for the classification of electronic cells. The match is facilitated by considering data abstractions and broad classes of cells. This matching process is heuristic since abstracted data is considered in place of detailed information.

3. *solution test and refinement*: heuristic methods are required to make choices between the candidate models for the cells and signals of a design. Consistency between the models selected is verified, which can result in more refined solutions.

1.5 Relationship to Other Work

This work is related to research being done in the CAD group in the Computer Science department at the University of Manchester. It has long been felt in the group that it is evermore unrealistic to largely rely on human intervention for the analysis and design of electronic hardware [Kah85]. One way of addressing these problems is to incorporate into the CAD algorithms greater awareness of factors such as the design strategy or the design rules of a particular silicon technology. The approach taken in the research group is to separate algorithmic action from technological and environmental rules. A high degree of adaptability is thus obtainable and the generality of classical CAD algorithms is not lost in favour of domain-specific solutions. A database system that defines, stores and manages rule-based technology information has been developed [AK86]. The explicit extraction of technological and environmental design rules provides vital support for the development of KB applications. These rules can be accessed by specific KB applications which are supported by a rule-based expert system environment [Fil88]. The experimental KB applications undertaken [KF85, Lai86, FS87] are all shallow KB systems. These systems lack the ability to build and exploit deep semantic models of electronic data. The analysis of semantic information discussed in this work attempts to improve this situation.

This kind of analysis can be related to software engineering research aimed at recovering (*re-engineering*) unavailable information concerning software specification and system design decisions from the information available in the existing system source code. The scope of system re-engineering activities ranges from error correction (system debugging), through program optimisation, restructuring, addition or modification of functionality, all the way to system migration (e.g. to a new software/hardware platform) or renovation [KN89]. Except for the extreme cases of system migration and renovation, the rest of the activities are known as software maintenance. System maintainers often find themselves in the position of attempting to re-engineer the system design or its high level specification (usually not the whole system but parts of it). The system high level specification and design documentation may not be complete, reliable, or even available. The only reliable sources of system information are the programming language code (the lowest level of system definition) and the behaviour of the system when executed in a computer.

These re-engineering activities have clear counterparts in electronic design (needless to say, algorithmic software descriptions are used to represent the behaviour of digital hardware designs). Figure 1.1 serves to illustrate this comparison and to delimit the aims of this work. In the *forward engineering* process, system specifiers translate a requirement statement describing the application or use of a system (in a problem domain-specific language) into a domain independent language. For example, the intention may be to build a system to control the lights of a crossroad. The specification of this requirement may result, for example, in a state diagram describing the different possible situations or a high level algorithmic description of the system. The translation of domain-specific statements into a domain independent representation depends critically on human creativity. The designers produce a logical or detailed software/hardware system design from the system specification (*stepwise refinement process*). The detailed system design is passed to the implementors who write the system code or produce the required electronic circuits.

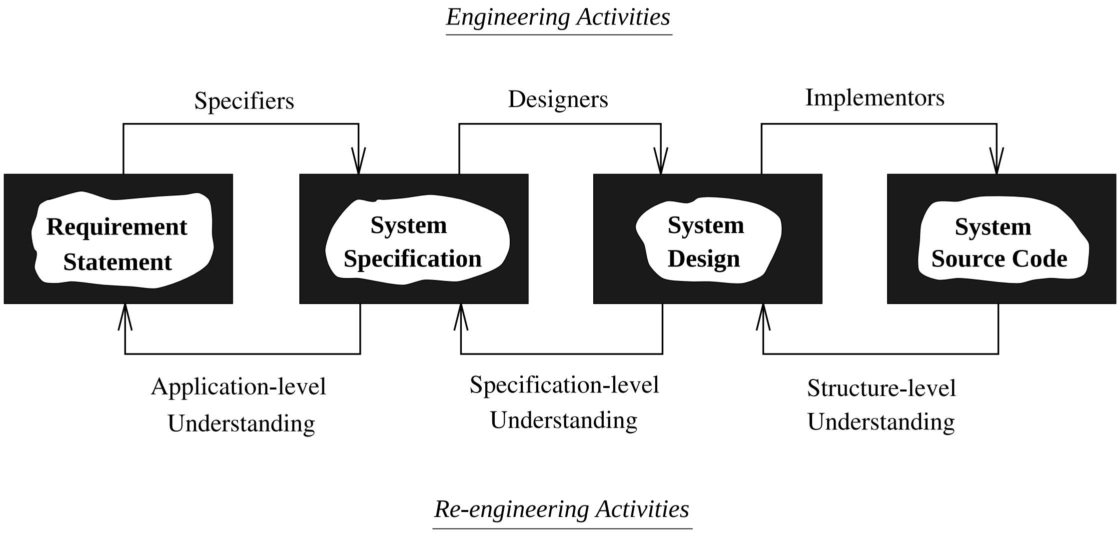


Figure 1.1: Engineering and Re-engineering Activities

The understanding of a software/hardware description refers to an abstracted representation of the system source information. This *reverse engineering* process starts with an understanding on the level of the software/hardware description language used. Language specific details are abstracted into entities of the system and an understanding of how these entities interact with each other when the system runs can be achieved. For a software description, entities such as procedures, data structures, variables and files are identified and it is possible to gain an understanding of how these entities affect each other (for example, which procedure can change the value of a variable or can call another procedure). For a hardware description, entities such as cells, ports and signals are identified. An understanding of which cells are sub-parts of other cells, in which way signals flow or which cells drive other cells can be obtained. This level of understanding is achieved by means of an analysis of the structural aspects of the system.

The structural analysis of a system does not reveal any information about the *meaning* or intended functions of its entities. The understanding at the specification level associates generic interpretations and problem solving heuristics (patterns or plans) with the identified entities. For example, in the software case, pieces of system code are associated with stereotypical implementation patterns of programming constructs, problem solving strategies, abstract data types and standardised algorithms [HN88]. As a result, it may be possible to derive, for example, that a piece of code iteratively performs an operation that can be associated with a high level statement of the type *read and accumulate* [KN89]. In the hardware case, electronic components can be associated with, for example, known electronic cells, classes of electronic cells or typical implementation patterns. The result can be the derivation of a high level statement that describes the operation of a component such as being identified as a storage component (e.g. a register), as a operator component that is used for the comparison of two values (comparator) or as a controller of other components (control component). Identified patterns can be further grouped together to facilitate higher level abstractions. The work presented in this thesis is aimed at the generation of knowledge up to this level of understanding.

The understanding at the specification level cannot relate the identified patterns and functionalities in the specification to the problem domain-specific applications. For example, a *read and accumulate* pattern of code or a group of electronic cells can be used to add up the salaries of the employees of a company. A storage entity (for example a list data structure or a list implemented in hardware) may contain the values of these salaries. The understanding at the specification level may derive these types of storage but not its domain-specific application. As mentioned above, the forward transformation carried out by human specifiers depends critically on human creativity. An artificial substitute for the reverse transformation can only rely on deep semantic information contained in the system code (such as the names given to the entities or the comments introduced).

This work also relates to research aimed at reasoning about electronic devices for the reformulation of designs [Sin87]. More generally, there are clear links with research aimed at reasoning about physical systems, particularly man-made devices. Examples of these typically include hydraulic systems, heat-transfer systems and mechanical devices. These are typical areas of concern of *commonsense reasoning* [Dav90]. In most cases, a system can be studied by considering separate components first, which are presumably simpler than the overall system, and the overall system later as the interaction of components. Commonsense theories that include the way in which solid objects interact, liquids flow or heat is transferred, are required for the understanding of these systems. The aim of commonsense reasoning is to define and apply commonsense theories to the real world to gain an understanding of everyday situations. The aim of heuristically reasoning about digital electronic design specifications is to improve the performance of automatic systems and to provide assistance to human experts. Applications of this work to current research will be detailed in chapter 9 .

1.6 Overview

As mentioned before, this work is concerned with the machine simulation of an heuristic way of understanding electronic designs exhibited by human experts. The aim is to provide the groundwork for an automatic heuristic understanding of digital electronic design specifications. This understanding is reflected on an heuristic classification of the electronic cells of the design and the signals which interconnect them. The following are considered to be the major contributions of this research work:

i – definition and classification of expert knowledge that can be used for the machine simulation of the heuristic way of reasoning about logic electronic design specifications exhibited by human experts.

ii – mechanisms required to capture expert knowledge about a design from an heuristic analysis of its specification, and mechanisms for reasoning about this knowledge which can allow a specification-level understanding of a design to be reached.

iii – application of the knowledge generated for a design to narrowing the gap between CAD systems and human experts for the analysis, design and management of electronic data.

iv – contribution towards an automatic machine understanding of digital electronic design specifications and, in general, towards the re-engineering of human specifications.

Bearing this in mind, the work presented in the remainder of this thesis can be seen as composed of three main parts. Part I is concerned with the investigation of expert knowledge that can be used for an heuristic exploration of the architecture of digital electronic systems. With this aim, chapter 2 presents an overview of the architecture of digital systems. The emphasis is on those characteristics that can be exploited for an heuristic understanding of the functionality of these circuits. Chapter 3 defines and classifies the expert knowledge that is used to build models of the operation of an electronic cell and its signals and formalises the procedure for the generation of these models. The knowledge used is occasionally quite ill-defined, but it often allows human experts to get a feeling about the design investigated.

Part II describes the mechanisms and methods that are used for the generation of expert knowledge about a design from the analysis of its specification. With the implementation of these methods, the system is endowed with a set of functions which attempt to simulate the heuristic way of understanding electronic devices exhibited by human experts. The procedure for the simulation of this process is based on the formation of knowledge plans for the cells of a design as discussed in chapter 4. Plans of knowledge about a cell are compared with system information for the generation of heuristic models of the cells. This is described in chapter 5. The control of the procedure for making choices between different alternative models for the cells of a design and for avoiding the combinatorial explosions that can occur is considered in chapter 6. The control is based on a pseudo-probabilistic measure of the confidence in the knowledge generated and it provides a mechanism for the evaluation and comparison of different solutions. Finally, chapter 7 presents a method for reasoning about the interactions between electronic cells considering that an heuristic model has been selected for each cell. This reasoning can result in the formation of additional knowledge plans for the cells of the design. This starts a new cycle of knowledge-generation/knowledge-propagation.

To conclude the thesis, part III describes the experimental work carried out and applications of this research. The current implementation of the KB system developed and the experimental results obtained are described in chapter 8. In chapter 9, the limitations of the reasoning method used are discussed, examples of exploiting heuristic design knowledge for the analysis, design and management of electronic data are examined, and directions in which this work can be extended are suggested.

Chapter 2

Architecture of Digital Electronic Systems

Electronic designers can often gain an overall understanding of the operation of a design from the observation of a few salient features which characterise its cells and signals. A major reason for this is that the interpretation of these features allows expert designers to derive knowledge for the classification of the functionality of the different objects into a small number of categories. The association of heuristic knowledge with a cell of a design narrows the range of possibilities for the operation of the cell and for the operation of the overall design. The architecture of digital electronic systems is briefly reviewed with the intention of introducing design features and functional categories of electronic cells and signals which can be exploited for an automatic heuristic understanding of logic electronic design specifications. These categories are defined and used in chapter 3 for an heuristic classification of electronic cells and signals.

2.1 Preliminaries

The analysis and design of digital systems is very complex and it is best managed by means of a structured approach. In this approach, a system (cell) is decomposed into sub-systems (sub-cells) and each sub-system is analysed or designed in a structured manner too. For example, in the design of a processor, system designers and system architects define the characteristics of the global system. Logic designers (or an automatic system) translate an algorithmic description of the system into a logic design. This involves putting together functional cells such as registers, arithmetic and logic units (ALUs) and control logic to form a network of electronic cells that meets the functional requirements of the processor. These cells are then gradually decomposed into simpler logic functions. Circuit engineers design the networks of transistors that provide the required logic functions.

The intention of this chapter is to describe digital cells and their implementation without considering the complex logical and mathematical data involved in the specification of their behaviours. Information about the functionality and complexity of electronic cells is categorised and characteristic information related to their implementation is given. The association of this kind of information with a cell of a complex design narrows the range of possibilities for the operation of the cell and for the operation of the overall design.

An automatic understanding of a complex design can then be gained without considering the complex behavioural data.

A logic cell can be seen as a black box which takes a set of input values and generates a set of output values. For binary logic systems, these values are coded as vectors of binary signals or *bit-vectors* as shown in figure 2.1. A cell performs one or more logic operations on n input bit-vectors and generates m output bit-vectors as a result. The logic behaviour of a cell is an abstraction of the electrical behaviour which relates the voltages and currents in the circuit. This abstraction is necessary since the analysis and design at the electrical level is only possible with small digital networks [HJ83].

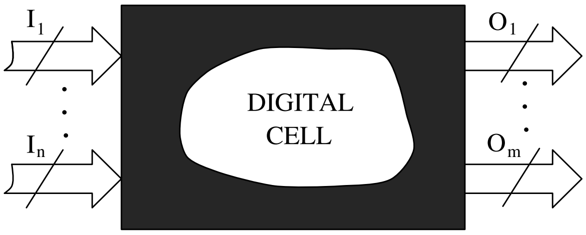


Figure 2.1: A Digital Cell

Two different models have been found useful for the analysis and design of logic digital systems: the *system* model and the *algorithmic* model [EL85]. The system model considers systems that implement relatively simple functions. These functions are categorised in section 2.2. Characteristic features of the implementation of these systems are also discussed. The algorithmic model is suitable for large designs since it facilitates global understanding. This approach relies on viewing a digital system as performing a computation described by means of an algorithm. Section 2.3 reviews this approach and classifies the digital cells required for building algorithmic models. These cells are designed and analysed by means of the system model. Characteristic features of the organisation and implementation of algorithmic systems are also discussed. General-purpose algorithmic systems (digital computers) are considered in section 2.4. The key points elaborated in this chapter are summarised in section 2.5.

2.2 System Model of Digital Systems

The system model represents the logical behaviour of relatively simple digital cells by means of two functions: a *state-transition function*, which computes the new state of a cell from its current state and the input values, and an *output function* which computes the output of a cell from its current state and the input values. The analysis by means of the system model of a network of digital cells requires the functions of the interconnected cells and parameters related to their physical implementation. Typical parameters include timing characteristics such as delay, set-up and hold times, maximum clock frequency, minimum pulse width, clock skew and loading characteristics such as load factors and fan-out.

The intention of this section is to describe the digital cells that are analysed by means of this model. The functionalities of these cells are categorised and characteristics of their implementation are given. An understanding of a complex design can then be gained without considering the output and state-transition functions of the cells involved. Most generally, the cells considered by the system model are classified into two classes: *combinational* and *sequential* systems. In combinational systems the outputs only depend on the current inputs (if zero delay is assumed). The system has no memory. Sequential systems are more general than combinational systems since the output at any given time depends not only on present inputs but also on past inputs (a sequential system has memory or state).

2.2.1 Combinational Systems

Combinational systems can only perform a limited set of operations since they do not have state. These systems can generally be classified according to the type of operations that they can perform as follows:

1. *transmission* systems: these systems perform a controlled transmission of values without transforming them. The purpose of the transmission is to bring a value to the part of the system where it is to be transformed, stored or output from the system. Examples of these systems include selectors (e.g. a multiplexer), distributors (e.g. a demultiplexer) and shifters.

2. *arithmetic* systems: these systems implement basic numerical functions such as addition, subtraction, multiplication or division (e.g. adders and multipliers) or relational functions such as equal, less and greater (e.g. comparators). Systems with multiple arithmetic/logic functions (e.g. ALUs) are typically included here.

3. *code conversion* systems: these systems represent the input and output data of the system with different code without altering its meaning (e.g. decoders and encoders). Code conversion is sometimes necessary when data is transmitted into a digital system or out of it.

4. *data transformation* systems: this type includes the rest of combinational systems (e.g. parity generators and checkers).

A combinational system can be implemented by means of networks of ad hoc or standard combinational cells, networks of *gate* circuits (random logic), and directly as a network of transistors. In a network of combinational cells, data flows directly from the inputs to the outputs of the system and no feed-back loops usually exist. This is the case of the network of figure 2.2(a) (the network is combinational since the network function can be obtained by composition of the cell functions). A network with one feed-back loop is shown in figure 2.2(b). A network with loops can be combinational or sequential. Most of these networks are sequential. The combinational cases are just curiosities with a few minor exceptions (see [EL85] for some examples).

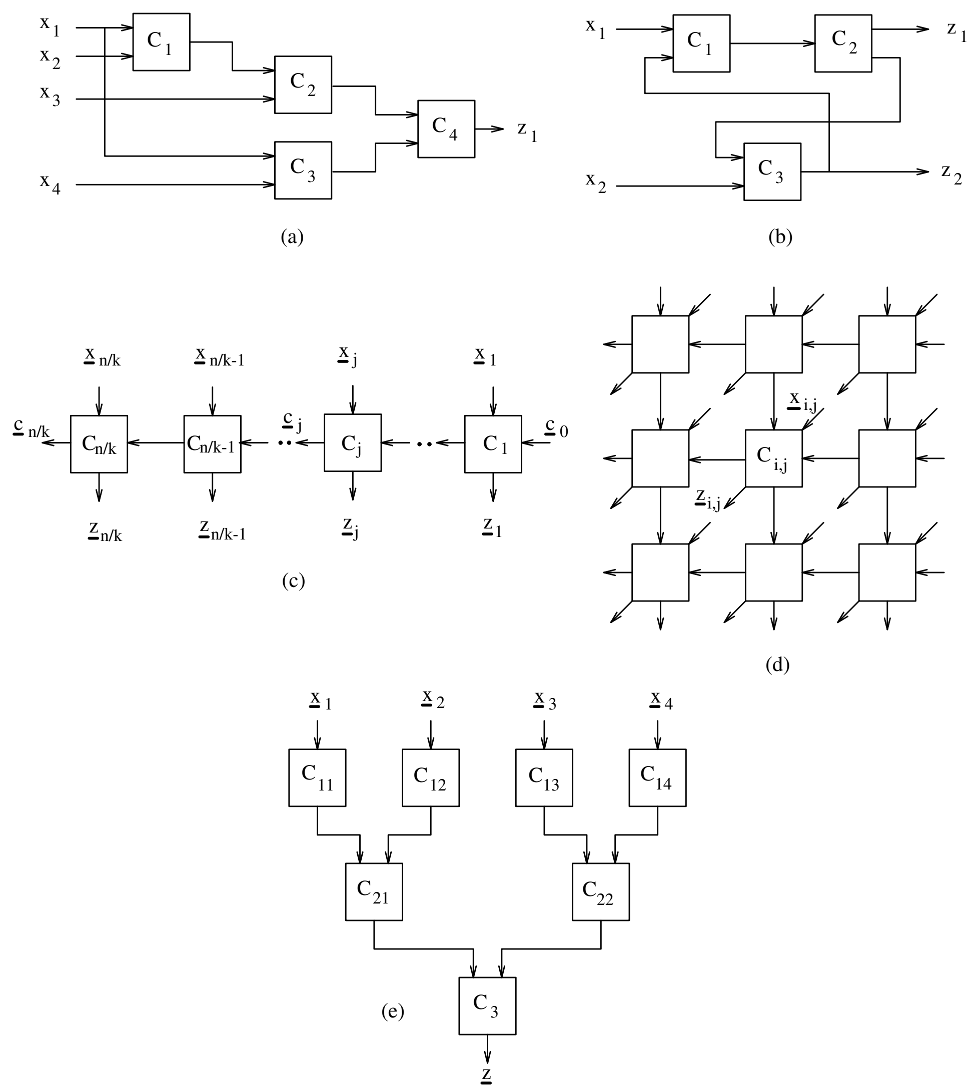


Figure 2.2: Combinational Networks: (a) a loop-free network, (b) loop network, (c) k-iterative network, (d) array network, and (e) tree network.

The choice of strategy for the implementation of a combinational system is usually a trade-off between performance and cost. For small designs, ad hoc implementations using networks of gate circuits are typical. It is also common practice to implement a small combinational system directly as a network of transistors or switches. This results in a further reduction in size and an improvement in performance [WE85, MC80]. A standard cell can be used for the implementation of small designs if performance is not critical. A relatively small set of standard cells has evolved for the design of all types of combinational systems. Examples of these are implementations with a multiplexer, decoder or a read-only-memory (ROM). The implementation with programmable cells such as programmable-logic-arrays (PLAs) is very common. This provides cheaper designs at the expense of a slower performance and larger area of silicon than well-suited fixed functions.

For large designs (networks with a large number of inputs), implementations with networks of standard cells are used. Examples of these include ROM networks, PLA networks, decoder networks, multiplexer trees and shifter networks. Some techniques have evolved which emphasise regular connections and topological simplicity by constructing a network as a regular repetition of a single cell. This reduces considerably the design cost. Examples of these techniques include:

1. *iterative* networks: they consist of several identical cells organised as a one-dimensional array. A cell is connected only to its neighbours. Figure 2.2(c) represents a k-iterative network: an implementation of a function of n bits by a network consisting of n/k identical cells. The input bit-vector is partitioned into n/k groups of k elements each and each group is applied to a different cell. These networks may be too slow for some applications because of the large delay of propagating signals through the array (e.g. an iterative adder).

2. *array* networks: these networks correspond to the extension of iterative networks to two dimensions as shown in figure 2.2(d). This organisation is typically used in arithmetic systems.

3. *tree* networks: this implementation is possible for some functions and it leads to more regular and faster networks than in the iterative case. Figure 2.2(e) shows an example of this type of implementation. Different chunks of the input data are processed in parallel and the need to propagate signals is decreased (an example of this is a tree of comparators as shown in section 7.6.2).

2.2.2 Sequential Systems

Sequential systems are more general than combinational systems since the output at any given time depends not only on present inputs but also on past inputs. This results in much more complex and useful behaviours. A sequential system takes input values and generates a change of state according to the state-transition function and output values according to the output function. The systems in which the change of state takes place at discrete instants of time as defined by a synchronising input or *clock* are called *synchronous* sequential systems. In *asynchronous* systems the state can change at any time as a function of the input changes since there is no external synchronisation. Asynchronous systems are of course more general and faster than synchronous systems, but because of their complex and sensitive behaviour they are more difficult to design and they are less frequently used.

The sequential systems analysed by means of the sequential model can generally be classified according to the type of operations that they can perform as follows:

1. *data storage* systems: because sequential systems have memory the most elementary functions are dedicated to the storage of data. Examples of these include cells such as latches, flip-flops and registers (a register is an array of latches or flip-flops). Systems that can store multiple bit-vectors include register-files, random-access-memories (RAMs) and content-addressable-memories (CAMs).

2. *sequential arithmetic* systems: these systems operate serially upon operands represented by bit-vectors. In a serial execution, the result of the operation is obtained as a sequence, one bit at a time. The operands can also be input to the system as sequences. Examples of these are sequential adders and sequential ALUs.

3. *sequential code conversion* systems: in a sequential (or serial) code converter the binary input vector is applied as a sequence, one bit per clock pulse. The output is produced when the whole input vector has been applied.

4. *controllers*: these systems issue a sequence of *control* signals to determine some actions or operations in other systems (see section 2.3).

5. *data transformation/generation* systems: this includes the rest of sequential systems. Examples of these systems include counters, pattern recognisers and pattern generators. A controller differs from a pattern generator in that the pattern of control signals generated usually depends on the values of the inputs to the system.

The state of a sequential network is stored in the memory cells. The most basic sequential circuits are latches and flip-flops. These cells store one bit of information. The storage is achieved by means of an electrical feed-back loop or by means of a dynamic storage of charge (in this case the cell is periodically refreshed). A variety of latches and flip-flops exist for the design of sequential networks including D (delay) flip-flops, SR (set-reset) latches and flip-flops, JK flip-flops and T (toggle) flip-flops. More complex sequential systems are implemented by using these cells as the basic building blocks.

The state description of a sequential system is the basis for the implementation of a sequential network. The most typical way of constructing small sequential systems is the canonical implementation shown in figure 2.3. It consists of a state register, to store the state, and a combinational network to implement the transition and output functions.

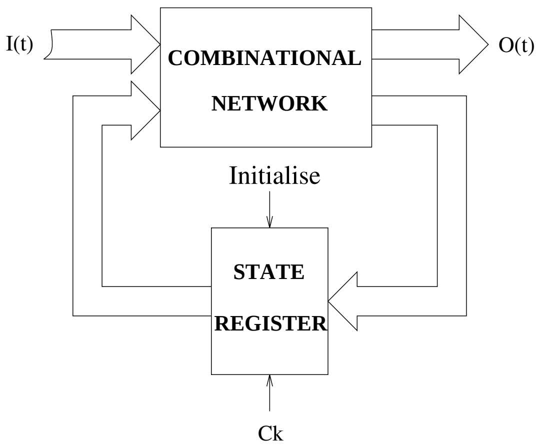


Figure 2.3: Canonical Implementation of a Sequential Network

Some functions on discrete variables can be implemented by a combinational system or a sequential one. In the combinational case, the outputs of the function are generated simultaneously (parallel output) in a network that usually contains different branches for each bit of the result [1]. In the sequential case, the function is performed in several clock pulses and the output is often generated serially (serial output), one bit per clock period. The same cell is repetitively used for the computation of the different bits of the result [2]. The choice of a combinational or sequential implementation represents a trade-off between time and space. A Sequential implementation is generally slower, since the output is calculated serially, but it almost always requires a smaller space than its combinational counterpart. The system can have a parallel or serial input and a parallel or serial output. The use of these alternatives in different parts of the system makes it necessary to convert from a parallel to a serial representation and vice versa. Multiplexers, demultiplexers and shift-registers are commonly used for this purpose.

A number of different techniques can be used for the implementation of a small sequential system with standard cells: a register and a combinational network (ROM, PLA or random logic), a counter and a combinational network, a shift-register and a combinational network or a RAM and a combinational network. However, the explicit separation between the state and the functions does not usually lead to an adequate implementation since it does not provide for modularity. In modular design, the sequential system is decomposed into sub-systems and each of these is implemented as a separate ad hoc or standard cell.

A number of techniques exist for the organisation of modular sequential networks. Multimodule implementations can be used for the implementation of large sequential networks including multimodule registers, register arrays, multimodule shift-registers, multimodule RAMs and multimodule counters. The two basic techniques for the composition of sequential networks are:

1. *cascade* composition: in this composition illustrated in figure 2.4(a), the input to cell C_1 is the input to the system. The input to cell C_j $(j > 1)$ is the output of cell C_{j-1}. The output of cell C_n is the output of the system.

2. *parallel* composition: this composition is illustrated in figure 2.4(b). The input is the same to all cells. The next state of any cell C_j $(1 \leq j \leq n)$ only depends on the present state of this cell and its input. It does not depend on the state of cell C_i for $j \neq i$.

There is no systematic approach for the decomposition problem. The design should minimise the number of cells and the number of different cells and it should simplify the interconnections between cells. Programmable sequential arrays (PSAs) can also be used for the implementation of a sequential network in a similar manner as PLAs are used for the implementation of combinational networks. A PSA includes storage cells which can be programmed to implement sequential systems.

[1] Technically this often implies that the outputs are generated within the same clock period.

[2] This is clear for example in iterative networks. A sequential implementation of an iterative combinational network is immediate since the internal variables that are passed from cell to cell in the iterative network (see figure 2.2(c)) correspond directly to the state variables in a sequential implementation. The sequential implementation only requires a single instance of the iterative cell and a state register.

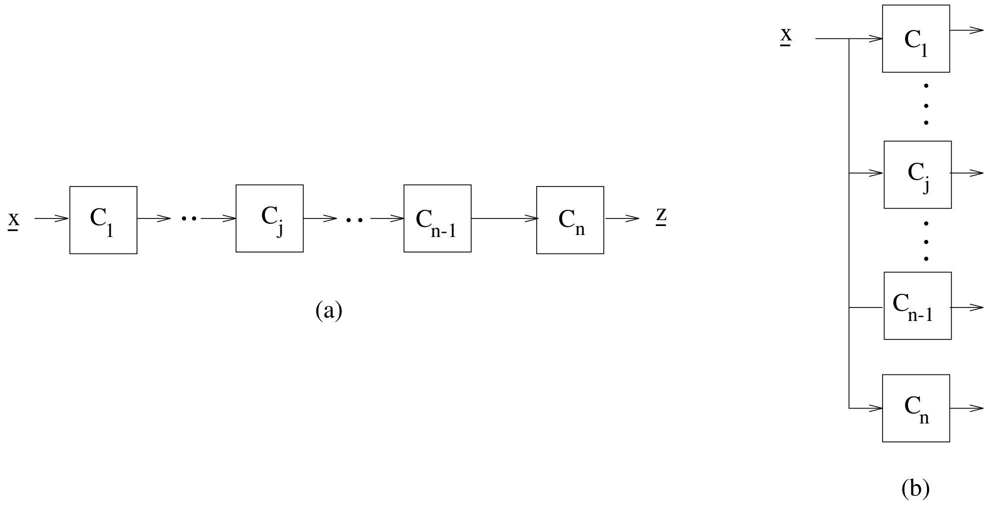


Figure 2.4: Data Flow in Synchronous Sequential Networks: (a) cascade composition, and (b) parallel composition.

2.3 Algorithmic Model of Digital Systems

The system model becomes unmanageable for large sequential systems in which no simple function can describe the state transitions of the system. These systems require an algorithmic approach in specification, analysis and design. The digital system is viewed as performing a computation (a transformation on data sets) which is described by means of an algorithm. The algorithm is the decomposition of a computation into sub-computations with an associated precedence relation that determines the order in which the sub-computations must be performed. The approach is hierarchical, ranging from high level computations at the software level down to low level primitives at the hardware/firmware level that are directly executable by the system. At the software level, the computation is specified as a set of statements of a programming language. The firmware level is an intermediate level in which a programming language is used to represent the algorithm, but the sub-computations are logical operations directly executable by the system (the corresponding programs are called *microprograms*). At the *hardware* level, the primitive computations are logical operations implemented by the hardwired connection of components.

The sequencing of sub-computations defines the *flow of control* in the system and the flow of operands and results corresponds to the *flow of data*. According to the flow of control or sequencing structure, algorithms can be classified as *sequential* and *parallel* (or *concurrent*) algorithms. An algorithm is sequential if during its execution there is only one active sub-computation at a time and parallel if several sub-computations can run at the same time. Sequential systems are simpler to develop, represent and execute. Because of this, and the fact that every parallel algorithm can always be converted into an equivalent sequential one, many systems only execute sequential algorithms. *Parallel systems* are potentially faster but they are more expensive and have a complicated control structure.

Some sub-classes of the class of parallel algorithms are specially interesting since they have the advantage of being concurrent but do not require the complex sequencing control of the general parallel case. The most significant is the sub-class of group-sequential algorithms. In a *group-sequential* algorithm, sub-computations are divided into groups. During the execution, all the sub-computations in one of these groups are initiated simultaneously. The next group is initiated when all sub-computations in the previous group have terminated. The implementation of systems that can execute group-sequential algorithms is considered below to describe and categorise some of the most relevant features of the architecture of algorithmic systems.

2.3.1 Structure of Algorithmic Systems

The algorithmic approach is reflected on the structure of the system by dividing the state of the system into two parts: the *data sub-system* and the *control sub-system*. The data sub-system consists of a set of cells which execute data transformations under the direction of the control sub-system. This decomposition largely simplifies the design of the whole system. At the hardware/firmware level of the machine sub-computations operate on bit-vectors stored in registers. Operations are performed by transferring bit-vectors between registers. This level of detail is known as the register-transfer-level (RTL).

The structure of an algorithmic system is built with elements of a small set of basic components which are separately designed. The types of components include:

1. a *storage* component which is viewed at a high level as directly storing the data sets. These components include hardware for read-write access operations which depend on the type of storage. Examples of types of storage include registers, register-files, stacks and memories. The cells used for the implementation are the data storage sequential systems described in section 2.2.2

2. an *operator* component executes the sub-computations of the high level algorithm by performing transformations on the bit-vectors. The physical implementation of an operator can be a combinational or a sequential system. Operators can often perform several functions as specified by some control inputs for the selection of the operation.

3. a *control* component controls the sequencing of sub-computations and their execution. It sends *control signals* to each operator to control the execution of the sub-computation and it receives *conditions* from the operators to make data-dependent control and sequencing decisions. The sequencing can be synchronous or asynchronous. Asynchronous sequencing is controlled by completion signals generated by the operators.

4. *data* and *control paths* are used to provide connections between components in the system for the transmission of data and control information.

5. input/output communication with the environment of the system is usually implemented using storage components or *I/O ports*. The communication is usually

controlled by a *handshaking* protocol which involves a transfer of control information between the systems in communication.

The structure of an algorithmic system maps the algorithm that it has to execute. The simplest mapping is to associate one operator and the corresponding storage with each sub-computation in the algorithm and connect the operators according to the sequencing structure. In this case, the operator, storage and data-path components are *dedicated* and *decentralised*. The resulting system will probably allow the maximum concurrency and speed, but the number of operators might be very large resulting in a very costly system. The cost of the system can be reduced by sharing some of the components. For example, an operator can be used to perform several of the sub-computations (at different times) and a storage component to store several data elements. The operator function can be completely centralised by just having one operator to perform all sub-computations. The system then executes a sequential algorithm that is equivalent to the defined algorithm.

The controller can also be centralised or decentralised as seen in figure 2.5. In a system with *centralised control* (figure 2.5(a)), there is just one controller that controls the whole execution. In a *decentralised control* (figure 2.5(b)), the operator and storage components have associated with them mechanisms to control their operation and also the sequencing. An intermediate organisation with *semicentralised control* has the control of the operation of each component decentralised but the sequencing among components is centralised (figure 2.5(c)).

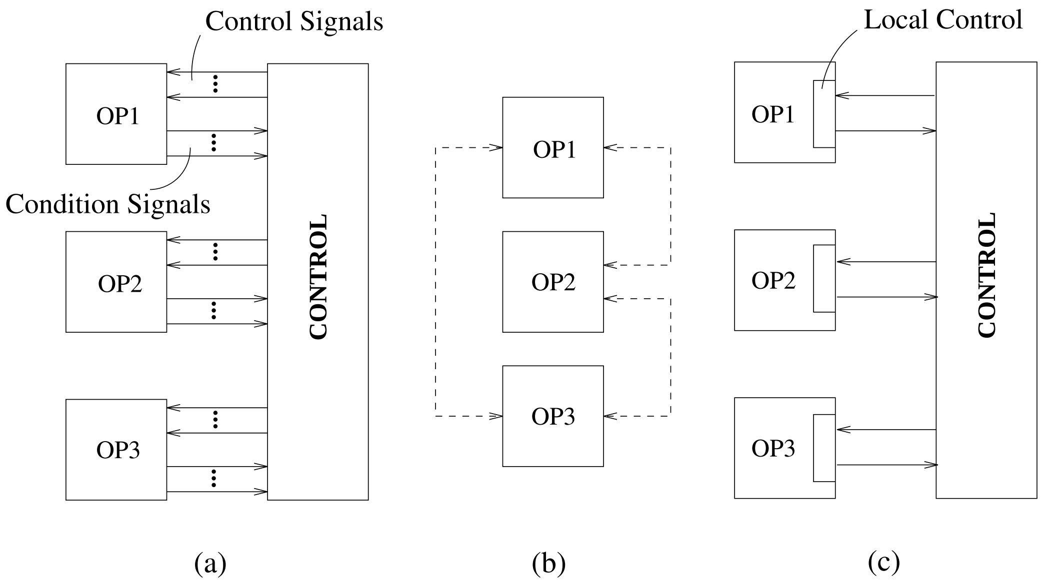


Figure 2.5: Organisation of Algorithmic Systems: a) centralised control, b) decentralised control, and c) semicentralised control

2.3.2 Implementation of Group-Sequential Algorithmic Systems

Many digital systems are implementations of group-sequential algorithms. These algorithms have the advantage of requiring only a sequential controller while providing some concurrency in order to increase speed. The hardware/firmware implementation of a group-sequential algorithm has the following characteristics:

1. the data elements are bit-vectors and arrays of bit-vectors in the implementation.

2. the sub-computations are limited to those realisable by the hardware operators. A sub-computation or *microoperation* consists of a transfer of bit-vectors between registers (*register transfer*). During the transfer, the operators can perform transformations on the bit-vectors.

3. since the algorithm is group-sequential, a group of microoperations called a *microinstruction* are executed simultaneously, but only one microinstruction is executed at a time. A sequence of microinstructions is a *register-transfer sequence* that implements the algorithm. This sequence is a *microprogram* at the firmware level. The implementation is a mapping of the register-transfer algorithm into a hardware/firmware implementation.

4. the system to execute the algorithm has centralised or semicentralised control, and the control unit is implemented as a sequential sub-system.

The structure of the system consists of a *data sub-system* and a *control sub-system* as seen in figure 2.6. The data and control sub-systems communicate by means of *control signals* and *conditions*. The system interfaces consist of *data inputs*, *data outputs*, *control inputs* and *control outputs* (control outputs issue condition signals).

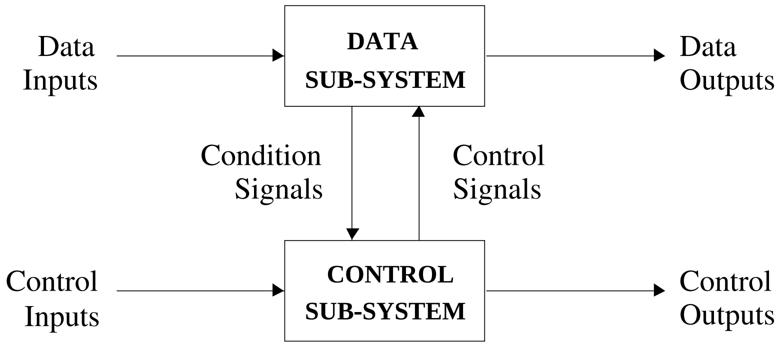


Figure 2.6: Structure of a System with Centralised Control

i) Data Sub-System:

The data sub-system is the part of the system in which the data is stored, moved and transformed. The components used in the data sub-system are storage cells, operators and data-paths. A component usually has data inputs/outputs identified as input/output bit-vectors, control inputs (*control points*) and possibly control outputs. The operation to be performed by a component is determined by the control signals generated by the control sub-system and presented at the control points. Typically, the control signals control the selection of operations, the data-paths and the register loading. The data sub-system generates conditions that are used by the control sub-system to make data-dependent sequencing decisions and for the generation of control signals.

The implementation of a data sub-system can be partitioned into cells in two basic ways as shown in figure 2.7. One way is to have a cell that implements completely each type of function required by the data sub-system. For example, separate cells implement data storage, transmission and arithmetic operations in figure 2.7(a). Alternatively, a *bit-slice* approach can be used as shown in figure 2.7(b). In this approach, a cell can implement all functions required by the data sub-system (storage, operators and data-paths) for a limited number of inputs. Several of these cells can be interconnected together if the data sub-system requires more inputs than the ones provided by a single cell.

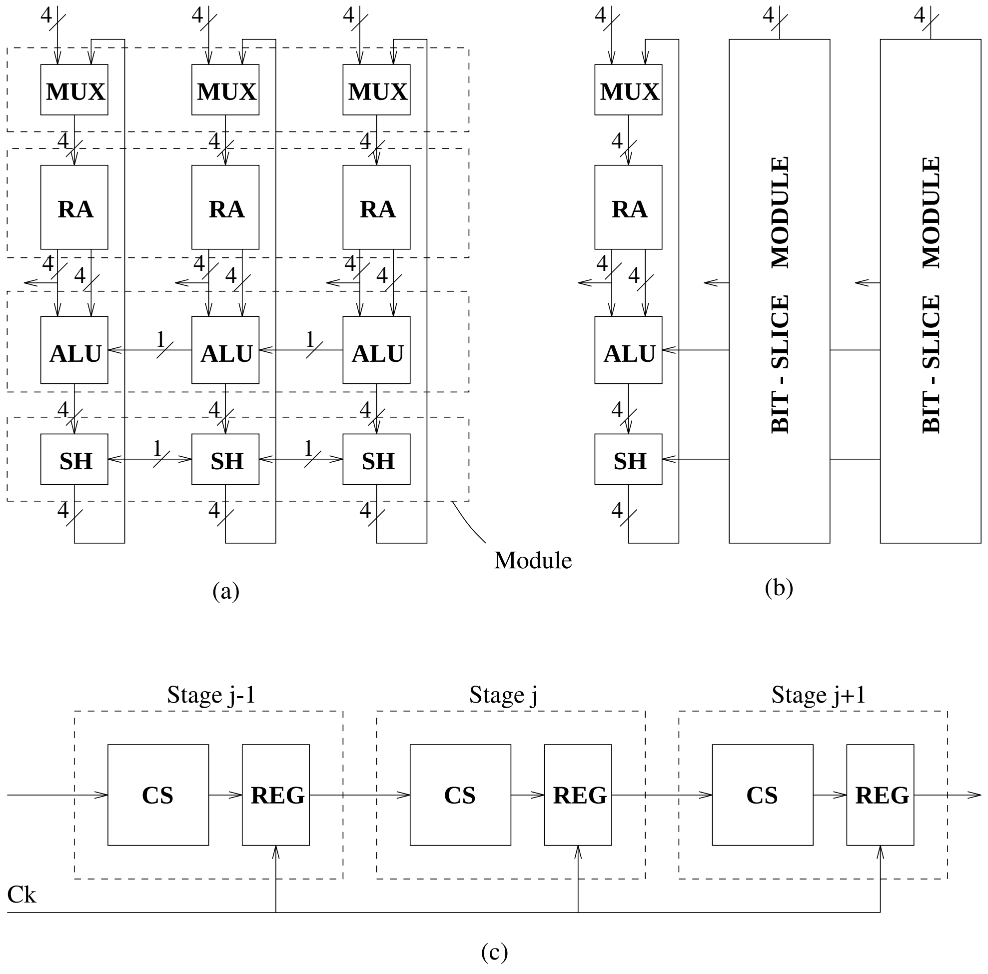


Figure 2.7: Implementation of Data Sub-Systems: (a) modular implementation, (b) bit-slice implementation, and (c) pipelining

The data sub-system often presents a *pipelined* organisation. The system can then be used in a *streaming mode*: the data inputs to the system consist of streams and the system performs the same computation on each element of the input streams. In the simple pipelined implementation of figure 2.7(c), the data sub-system consists of a combinational network per stage and registers between stages. Each stage in the pipeline can be executing operations on different elements of the input streams.

ii) Control Sub-System:

The control sub-system generates sequences of control signals to control the sub-computations in the data sub-system. Control signals are generated according to the register-transfer algorithm that specifies the hardware/firmware implementation of a given computation. The transition function of the control unit specifies the precedences of microinstructions in the algorithm. The output function specifies the control signals that activate microoperations executed by the data sub-system.

The control unit can be implemented in several ways ranging from fixed *hardwired* control units to flexible *firmware* ones. Simple systems are usually hardwired using combinational networks for the implementation of the transition and output functions. Firmware approaches are common for complex control sub-systems. In this approach, the transition and output functions are stored in a ROM or a RAM memory. A number of memory words correspond to a microinstruction of the program (or in some cases several instructions per word). The sequencing of microinstructions or microprogram is represented by the ordering of microinstructions in the memory and by binary branches. A microinstruction is usually composed of several fields that determine the control signals that are active during the execution of this instruction and the next microinstruction to be executed. A typical microprogrammed control unit is shown in figure 2.8. This unit consists of the following components:

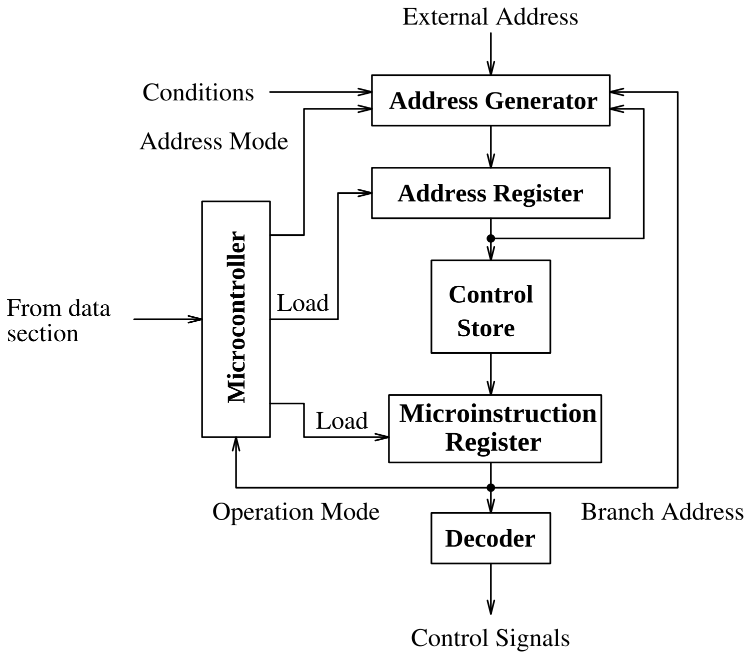


Figure 2.8: Microprogrammed Control Unit

1. a *control store* contains the microprogram representing the algorithm. A firmware-type control store can be implemented by means of a ROM, PROM (programmable ROM) or a RAM memory. PROM and RAM implementations allow microprograms to be modified. The RAM implementation allows microprograms to be written to the control store under system control. Systems with writable control stores are called *microprogrammable systems*.

2. an *address register* contains the address of the microinstruction in the control store that is going to be executed. This corresponds to the state register of a simple sequential system.

3. an *address generator* calculates the address of the next microinstruction to be fetched and executed. Typical functions of this component include incrementing the current address by one, loading an external address, loading a computed branch address or loading an initial address.

4. a *microinstruction register* contains the microinstruction being executed.

5. the fields of the microinstruction register are usually decoded by means of a *decoder* to generate the control signals.

6. a *microcontroller* controls the operations in the control unit.

The operation of a microprogrammed control unit usually consists of four main steps: 1) *instruction fetch* from the main memory; 2) *instruction decoding* for the identification of the operation to be performed; 3) *operand fetch* if needed in the execution; and 4) *execution* of the decoded arithmetic/logic operation. The example circuit of figure 2.8 is used in section 7.5 to illustrate an automatic understanding of electronic designs.

2.4 Computer Systems

A system to execute algorithms can be a special-purpose machine or a general-purpose one. A *special-purpose* machine performs one of a small number of related algorithms. These machines have a non-writable control store and can be better optimised for a particular application. The programming of a *general-purpose* machine (digital computer) is usually cost-effective for non-critical applications. This section examines the architecture of some of these systems in order to provide some groundwork for an automatic heuristic exploration of the architecture of these systems.

A computer is capable of performing a large number of different algorithms. The algorithm executed by the system is determined by the microprogram stored in the writable control store. General-purpose systems can be programmed using a high level language, a machine language or a microprogramming language. High level languages are usually compiled into a machine language before they can be executed by the computer. The fact that the machine language is machine-dependent does not necessarily limit the generality of a general-purpose computer. In most general-purpose computers the control is microprogrammed. The execution of a machine instruction corresponds to the execution of a sequence of microinstructions. A general-purpose microprogrammed system can be used to implement or *emulate* many different general-purpose computers with different machine instructions.

Most modern computer systems still follow the usual frame of reference imposed by the Von Neumann model [Bae80] (see section D.1). These machines are generally called *control flow* computers with instructions executed sequentially under the control of a program counter. A number of departures from this initial model have been introduced in the past in order to create faster (usually parallel) machines [3]. The architecture of these computer systems represents a further level of complexity in the design of digital systems.

The primitive components that are to be found in a description of the overall structure of a computer system are [BN71]:

1. a *memory* component stores information in terms of individually addressable units. Since fast storage elements are expensive, the requirement of large and fast storage is obtained by means of a hierarchy of components of varying capacity and speed: registers in the processor, cache memory for fast interface between the processor and the central memory (usually implemented by means of RAM blocks) and secondary or extension store for the storage of large amounts of data (such as disks, drums and magnetic tape units).

2. a *link* component establishes the connections among the other components providing a path for the transfer of information without altering it.

3. the *switch* component is used for establishing links for the transfer of control and data information. A switch can enable or disable a set of links associated with it. This is required, for example, when several devices are connected to a processor. Addressing mechanisms are required to control the switches.

4. a *control* unit activates the other components. With the exception of the processor component which has a control unit built within it, these components are the only active components of the system.

5. the units of information are altered in the *data-operation* components.

6. the communication to the environment is represented by means of *transducers* which can transform the representation of data without altering its meaning.

7. the *processor* component executes the sequence of instructions of a stored program. This is not a purely primitive component as the above six. It consists of a combination of local memory, a local control unit, and data-operation units interconnected

[3]To increase speed and achieve maximum parallelism *data flow* computers have been suggested [HB85]. In a data flow computer, the execution of an instruction takes place whenever its required operands become available, regardless of the physical location of an instruction in the program, and no program counters are needed.

by means of links and switches. At the system level of description, it is rare to decompose a processor in terms of its components.

The organisation of computer systems can be divided into two groups [4]: those which are built around a single processor (uniprocessor systems) for conventional applications and those which include special techniques for high-performance computer applications. Inferences about the nature of the operations performed by the components of a computer system can be drawn from an observation of the architecture of the system. The architecture of computer systems is briefly described in appendix D.

2.5 Summary of Key Concepts

This chapter has presented a brief review of the architecture of digital cells. Digital cells range from simple designs with a few transistors to complex processors and digital computers. The design, analysis and specification of these systems is best managed by means of a structured approach. Complex digital cells are built from the composition of simpler sub-cells.

The functionalities of relatively simple cells that can be analysed by means of the system model can be categorised in a small number of generic classes. Examples of these classes include data transmission cells, arithmetic cells, data storage cells, code converters, data transformation/generation cells and controllers. Different techniques are used for the implementation of these cells. These techniques often highlight the type of operations of a cell.

Algorithmic digital systems are built from cells that are designed and analysed by means of the system model. These designs usually present a clear distinction between data and control sections in the system. The components of the system are seen as storage, operator, control, data/control path and I/O components. The design implements computations which involve the transfer of data between storage cells through the data/control paths of the system. During the transfer of data, operator cells can be used to manipulate the data. The control of the computation is determined by control cells. Because of this, the structure of the system emphasises the nature of the operations performed by each component.

This is also true for general-purpose computer systems which are algorithmic systems with writable control stores (see appendix D). The typical functions of the components used to build these systems are memory, link, switch, control, data operation, transducer and processor functions (with the processor function being a combination of the other functions). The structure of the system narrows the range of possibilities for the operation of each cell. This provides the groundwork for an automatic heuristic exploration of the architecture of digital electronic systems.

[4]Traditionally, computer systems have been classified according to the uniqueness or multiplicity of instructions and data streams. See for example [Bae80].

Chapter 3

Heuristic Classification of Electronic Cells and Signals

The knowledge that is used in this work to build models for the cells and signals of an electronic design is defined and classified. This knowledge is chosen with the intention of allowing the system to exhibit a flexible way of reasoning about electronic cells, instead of a more rigid approach based on the use of logical and electrical data. The system attempts to assign this knowledge to the cells and signals of a design in order to form vague heuristic models of their operations. This corresponds to an heuristic classification of the cells and signals of the design, which narrows the range of possibilities for the operation of each cell and which provides the grounds for a flexible way of reasoning about the interaction between electronic cells. The problem of classifying the cells of a design is formally defined. The complexity of the problem is analysed and strategies to tackle it are introduced.

3.1 Introduction

A high level understanding of the architecture of digital electronic systems was discussed in chapter 2. This understanding is based on the categorisation of specific aspects of the cells and signals of a design. Examples of these include categories to represent the functionality of an electronic cell and categories to represent the functionalities of its interfaces. These categories are often quite general and imprecise in nature but when assigned to a cell they narrow the range of possibilities for the operation of the cell. This provides the grounds for a flexible way of reasoning about the interaction between electronic cells, instead of a more rigid approach based on the use of logical and electrical data.

A category is selected from an enumerated list of possibilities. The selection of a category for an aspect of a cell relies on salient features of this cell obtained from the specification of the design, on knowledge that is contained in the system and on the analysis of the interaction between the different cells. The assignment of these categories to a cell corresponds to an heuristic classification of the cell (as defined in section 1.4). The classification of a cell results in an *heuristic model* of its operation. Since logical and electrical data (as may be given in the specification) is not considered to categorise the different aspects of a cell, there must be limited confidence about the knowledge represented in a model for a cell. For this reason, a number of alternative heuristic models may be possible for each cell. The system must choose which model best classifies each cell.

Figure 3.1 illustrates the process used for classifying the cells of a design. The observation of salient features of the cells and signals of an electronic design allows the formation of data abstractions from the raw electronic data. These data abstractions are matched with heuristic models of broad classes of electronic cells and heuristic models of electronic cells already known by the system. Successful matchings of this data result in possible classifications of the cell (heuristic models for this cell). The classification is facilitated by considering data abstractions and heuristic models of cells and classes of cells. The matching process is heuristic since data abstractions are considered in place of detailed information. The most attractive models for the cells of a design must then be selected.

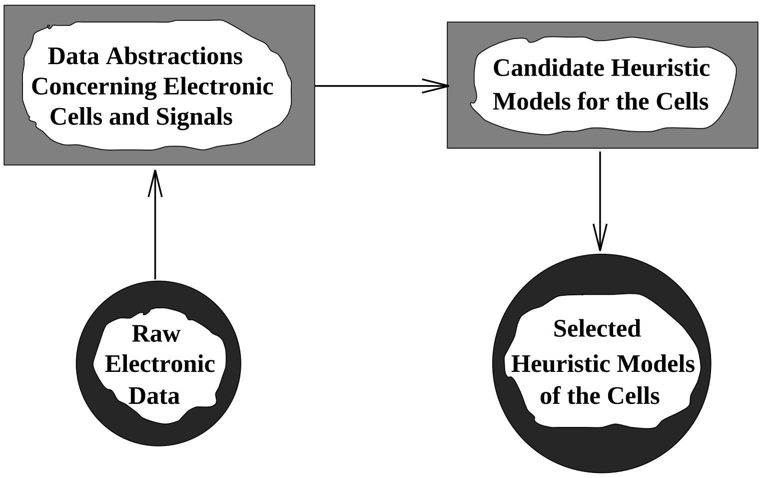


Figure 3.1: Heuristic Classification of Electronic Cells

This chapter defines and classifies the knowledge that is used in this work for an automatic heuristic understanding of logic electronic design specifications. It also formalises the problem of classifying the cells of a design. The hierarchical approach to the description of electronic designs is reviewed in section 3.2 in order to introduce some useful notation. The knowledge used to represent the different aspects of concern for electronic cells is described in section 3.3 (the use of this knowledge to represent heuristic models of classes of cells and the organisation of models in the system will be discussed in section 5.2 and section 5.3 respectively). The categories used to describe the signals of a design are discussed in section 3.4. The scenario for reasoning about a design is considered in section 3.5. Section 3.6 formally defines the problem of classifying the cells of a design as a problem of search. Finally, section 3.7 analyses the complexity of the problem and strategies to tackle it.

3.2 Design Hierarchy

A structured approach is essential for the description of large electronic designs. This approach is represented by means of a hierarchical description. A description is hierarchical in the sense that instances of a cell are used to represent the contents of another cell. An example of a hierarchical description for a design is shown in figure 3.2. Figure 3.2(a) represents the structure of a register that stores four bits of information. The implementation of this cell consists of four instances of a D-type flip-flop. The structure of this flip-flop is shown in figure 3.2(b). In this case, the flip-flop is constructed out of logic gates. A hierarchical description of the register is given in figure 3.2(c). The representation consists of two types of nodes: nodes which represent the cells of the design or *cell nodes* (which are filled in the figure) and nodes that represent instances of cells or *instance nodes* (which are not filled). The fact that an instance node references a cell node is indicated with an arrow from the instance node to the corresponding cell node.

A representation of a design which contains only cell nodes is also used in this work. This representation or *hierarchy graph* consists of a set of nodes $Cs = \{C_1, \ldots, C_n\}$, which represent the n cells of a design, and a set of arcs or links that indicate the hierarchical relationships in the design. Each link is an ordered pair of cells $< C_t, C_h >$ and it indicates that the head of the link C_h is a sub-cell of the tail of the link C_t. For example, the hierarchy graph for the design of figure 3.2 is given in figure 3.3(a). By definition, the hierarchy graph is a directed acyclic graph. If the graph is not acyclic, the implementation of the design is not possible. By definition too, a hierarchical design implies that there must be a top cell C_1 such that for each other cell C_i in the design $(2 \leq i \leq n)$ there is at least one path in the hierarchy graph from the top cell C_1 to C_i. In general, a hierarchy graph may have more than one path leading to a cell and therefore it will not form a tree structure. An example of this is shown in figure 3.3(b).

In a hierarchy graph, a cell C_i will have n_i links to its n_i sub-cells and ud_i links from ud_i different cells that instantiate it. The set of n_i sub-cells of C_i is called the *down-dependencies* of C_i and is denoted as Ds_i. The set of ud_i cells that instantiate C_i is called the *up-dependencies* of C_i and is denoted as Us_i. The *depth level* h_i of a cell C_i is defined as the number of cells that must be followed from the top cell C_1 to C_i along the longest path in the hierarchy graph. For example, to reach the cell C_4 from C_1 in the example of figure 3.3(b) there are two possible paths

$$\begin{cases} C_1 & \rightarrow & C_4 \\ C_1 & \rightarrow & C_2 & \rightarrow & C_3 & \rightarrow & C_4 \end{cases}$$

The depth level of cell C_4 is therefore $h_4 = 4$.

A cell can be seen as interacting in a number of different *situations* in the design. The cell interacts with a different set of cells in each situation. A *situation* corresponds to the description of a cell in terms of its sub-cells. The situation S_i corresponds to the description of cell C_i in terms of its sub-cells Ds_i and it is represented by the pair $S_i =< C_i, Ds_i >$. The situations in which a given cell interacts include the description of this cell in terms of its sub-cells and the use of this cell in the description of higher level cells. For example, cell C_4 in the hierarchy graph of figure 3.3(b) can be seen as

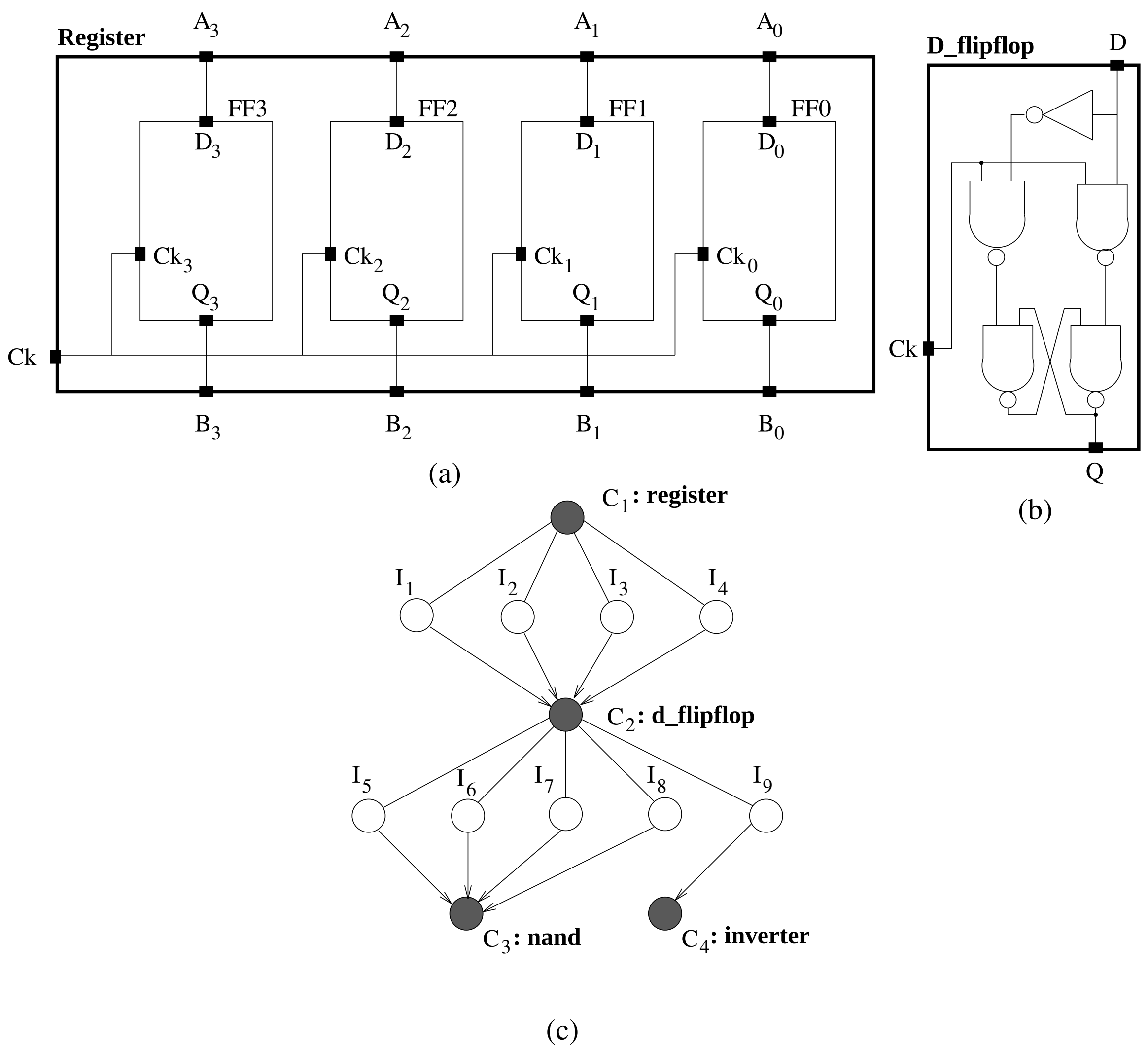


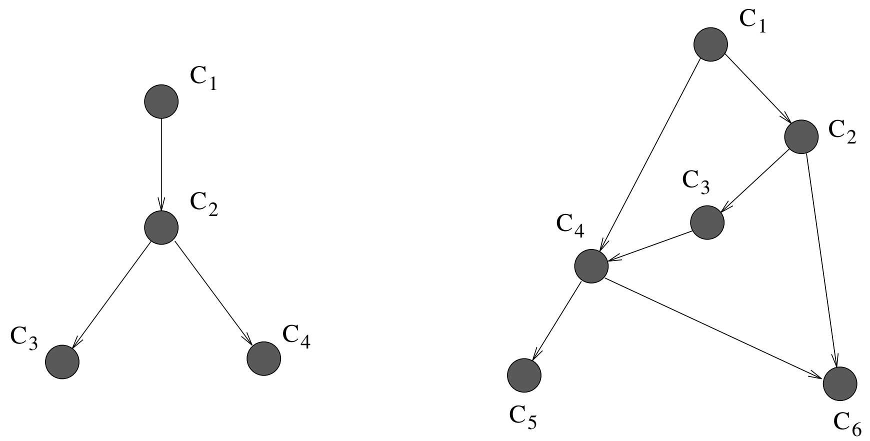
Figure 3.2: Design Hierarchy: (a) 4-bit register, (b) D-type flip-flop, and c) design hierarchy for the 4-bit register.

interacting in three different situations: $S_1 =< C_1, \{C_2, C_4\} >$, $S_3 =< C_3, \{C_4\} >$ and $S_4 =< C_4, \{C_5, C_6\} >$. In general, the number of situations in which a cell C_i interacts is given by

$$s_i = ud_i + 1 \tag{3.1}$$

3.3 Knowledge for the Classification of Electronic Cells

An heuristic analysis of the interactions between objects in a situation (e.g. cells, ports and signals) is based on heuristic knowledge obtained from the classification of the salient features of the cells and the salient features of the signals carried in the connections. The classification of a salient feature involves the selection of an alternative between a set of pre-defined categories which are used to represent this type of feature. These categories



$Ds_1 = \{C_2\}$	$Us_1 = \{\}$	$h_1 = 1$
$Ds_2 = \{C_3, C_4\}$	$Us_2 = \{C_1\}$	$h_2 = 2$
$Ds_3 = \{\}$	$Us_3 = \{C_2\}$	$h_3 = 3$
$Ds_4 = \{\}$	$Us_4 = \{C_2\}$	$h_4 = 3$

(a)

$Ds_1 = \{C_2, C_4\}$	$Us_1 = \{\}$	$h_1 = 1$
$Ds_2 = \{C_3, C_6\}$	$Us_2 = \{C_1\}$	$h_2 = 2$
$Ds_3 = \{C_4\}$	$Us_3 = \{C_2\}$	$h_3 = 3$
$Ds_4 = \{C_5, C_6\}$	$Us_4 = \{C_1, C_3\}$	$h_4 = 4$
$Ds_5 = \{\}$	$Us_5 = \{C_4\}$	$h_5 = 5$
$Ds_6 = \{\}$	$Us_6 = \{C_2, C_4\}$	$h_6 = 5$

(b)

Figure 3.3: Hierarchy Graphs: (a) 4-bit register hierarchy graph, and (b) hierarchy graph with several paths leading to a cell.

do not always have well-marked boundary lines between them. There are clear examples and clear non-examples of objects for each category but there are also examples of objects which lie in between (e.g. a port of a cell which can be functionally viewed as a select port or as an address port). The categories (or range of categories) selected to represent the salient features of a cell form an heuristic model for this cell. These categories are described in this section. The categories used to classify the signals carried in the connections are described in section 3.4.

When building the system described in this thesis, a range of choices existed to decide the form that the heuristic models of electronic cells would take. The heuristic models could include or exclude various aspects of knowledge about devices. The categories to describe these aspects could be made more or less rigorous. In general, the choices were made with the intention of supporting the way of understanding digital electronic devices described in chapter 2. The aspects of concern which are used in the classification of a cell include: a set of types for the cell, the electronic functionality of the cell, the functionality and organisation of its interfaces, the implementation of the cell and the flow of information in the cell.

i) Cell Types:

Four main heuristics are used to define the types of an electronic cell. These heuristics relate to the complexity of the cell or level of abstraction, to the type of logic used in the cell, to the typical operation or purpose of this cell in an electronic design and to the flow of data in the cell:

1. **Abstraction Level**: the level of abstraction of a cell reflects the complexity of its design. The following categories are used to represent the level of abstraction of a cell:

   (a) *transistor*: these cells correspond to primitive electrical elements such as transistors, resistors and capacitors. These cells are not further decomposed into sub-cells.

   (b) *gate*: the cells at this level include logic gates. Gates are described as the interconnection of cells at the transistor level according to the semiconductor technology used.

   (c) *bit*: a cell at the bit level performs operations upon single bits of information as opposed to groups of bits or bit-vectors (this excludes the operations performed by cells at the gate level). Examples of cells at this level include flip-flops, which store one bit of information, and full-adders which add up two bits of information. Cells at the bit level are usually described as the interconnection of cells at the gate level.

   (d) *vector*: cells at this level of complexity operate upon bit-vectors. These cells can often be obtained by arraying together a set of instances of a cell at the bit level. For example, a register is obtained as an array of flip-flops.

   (e) *processor*: this level includes large cells such as processors, memories and large controllers. The contents of these cells are described as the interconnection of cells at the vector level (such as the data-paths of a processor or the addressing mechanisms of a memory).

   (f) *computer*: a cell at this level corresponds to a full digital computer system. These cells are composed of units at the processor level such as a CPU unit, memories and controllers. Examples of these systems include uniprocessor systems (see section D.1) and high-performance systems (see section D.2).

2. **Logic Type**: the logic type of a cell classifies any digital system at the gate level or higher levels into two classes (see section 2.2):

   (a) *combinational*: combinational systems do not have state and usually there are no feedback loops in their networks.

   (b) *sequential*: these systems have memory or state. They can be synchronous, when they have clock signals for the synchronisation of operations, or asynchronous.

3. **Cell Purpose**: the purpose of a cell indicates the typical operation of a cell in a design and it is defined by one of a few generic functions. These functions are defined by considering that the cells of a system implement computations which involve the transfer of data between storage cells through the data-paths of the system. During the transfer of data, operator cells can be used to manipulate the data. The control of the computation is determined by control cells. The functions considered are (this is expanded in section 7.5):

   (a) *storage*: a storage cell is viewed as directly storing the information that flows or is used in the system.

   (b) *control*: a control cell controls the sequencing of computations and their execution in the other cells.

   (c) *operator*: these cells execute a computation upon the data applied to their input ports and the result is collected in the output ports.

   (d) *switch*: switch cells allow a controlled flow of information. They are used to build links for the transfer of control and data information.

   (e) *processor*: a processor executes the sequence of instructions of a stored program. This is a more generic function and it consists of a combination of cells implementing the other functions. At high levels of description, a processor is rarely decomposed in terms of its sub-cells.

   (f) *transducer*: transducer cells are used for communication with the environment. They usually transform the representation of data without altering its meaning.

   These generic functions are defined for cells at the bit level or higher levels. The functions considered at the bit and vector levels include storage, control, operator and switch. Bit and vector level cells have the same types of generic functions since usually a vector level cell is built from an array of instances of a bit level cell. Cells above the vector level are more specialised and they include processor and transducer functions.

   These generic functions are typical of other physical systems. For example, a system to transport or manipulate a fluid (such as water, gas or 'electrical signals') can be regarded as components or nodes interconnected by means of links or pipes. In the case of an hydraulic system, a component can be seen as a storage node (e.g. a tank of water), an operator node (e.g. a pump), a switch (e.g. a valve) or a control node for the regulation of the operation of the other nodes (e.g. the electric engine of the pump). These nodes are interconnected by pipes that transport the fluid or links for the control of the system. The whole system may be seen as a processor node (processing unit) controlled by a control unit and with a set of transducers (e.g. a heat exchange unit) for communication with the environment (e.g. heat transfer).

4. **Data Flow Type**: electronic cells at the bit level or higher levels are classified according to the flow of data in the cell into two generic types:

(a) *transporter*: these cells have some operation modes that allow the transfer of data input signals to data output signals under the control of control input signals.

(b) *modifier*: these cells do not have operation modes for the transfer of data input signals to data output signals.

The ability of a cell to transfer data is an important property to describe the operation of a cell. In a cell that is assumed to be of type transporter the transfers of data are usually achieved by means of chains of sub-cells of type transporter (if these sub-cells correspond to the bit level or higher levels of abstraction). And vice versa, the analysis of chains of transporter sub-cells can be used to determine the type of the flow of data in the cell (see section 4.5.3).

The types of an electronic cell are not independent. Table 3.1 describes the typical relationships. The symbol 'n/a' means not applicable. For cells at low levels of abstraction (transistor and gate levels) the purpose and data flow types of a cell are not applicable. The purpose and data flow types of a cell at the computer level are not considered either.

ii) Cell Electronic Functionality:

With the use of structured approaches to digital electronic design, a relatively small number of typical electronic functions have evolved in the field. Cells that implement these common functions can be used conveniently in many digital designs. Electronic designers use specific names for these typical electronic functions and for the cells that implement them from the transistor level to the computer level.

The functionalities of electronic cells at different levels of complexity were classified in chapter 2. Table 3.2 summarises these classes. The electronic functionality of a cell is also typically related to the types of this cell. These relationships are also shown in table 3.2. Knowledge about the types of a cell limits the range of possibilities for its electronic functionality.

At transistor level, the electronic functionality of a cell corresponds to the kind of electrical component such as 'transistor', 'resistor' and 'capacitor'. At gate level, the electronic functionality of a cell is described by means of the name of the boolean function that this cell implements such as 'and', 'nand', 'nor' and 'inverter'.

A classification of the electronic functionalities of cells at intermediate levels of complexity (bit and vector levels) was given in section 2.2.1 for combinational systems. These typical functionalities include data transmission, single/multiple arithmetic operations, code conversion and combinational data transformation. These general classes can be further sub-classified. For example, combinational transmission systems include selectors (e.g. a multiplexer), distributors (e.g. a demultiplexer) and shifters. Section 2.2.2 classified the electronic functionalities for sequential cells at intermediate levels of complexity (cells that can be analysed by means of the system model). These classes include data storage systems, sequential arithmetic systems, sequential code converters, controllers and data transformation/generation sequential systems.

Abstraction Level	**Cell Logic**	**Cell Purpose**	**Data Flow Type**
Transistor	n/a	n/a	n/a
Gate	Combinational		
Bit	Combinational	Control	Modifier
		Operator	Modifier/Transporter
		Switch	Transporter
	Sequential	Control	Modifier/Transporter
		Operator	
		Switch	Transporter
		Storage	
Vector	Combinational	Control	Modifier
		Operator	Modifier/Transporter
		Switch	Transporter
	Sequential	Control	Modifier/Transporter
		Operator	
		Switch	Transporter
		Storage	
Processor	Combinational	Control	Modifier
		Operator	Modifier/Transporter
		Switch	Transporter
	Sequential	Control	Transporter
		Operator	Modifier/Transporter
		Switch	Transporter
		Storage	
		Processor	
		Transducer	Modifier/Transporter
Computer	Sequential	n/a	n/a

Table 3.1: Cell Types

The electronic functionalities of cells at high levels of complexity include processor (e.g. a central processor and an I/O processor), memories (large data storage components), data-units (large multiple arithmetic/logic systems) and control-units (see section 2.3 and section 2.4).

iii) Functionalities and Organisation of the Interfaces:

The interfaces of an electronic cell correspond to the set of ports used for the interconnection of this cell. In general, the information about the ports of a cell that is directly available from its description corresponds to the direction of the ports ('input', 'output' and 'inout' or 'bidirectional') and their width or number of connection points in the port. This is represented in figure 3.4(a). This information does not provide any insight about the use or functionality of these ports.

Cell Electronic Functionality	Abstract. Level	Cell Logic	Cell Purpose	Data Flow Type
Electrical Component *transistor, resistor...*	Transistor	n/a	n/a	n/a
Primit. Logic Function *and, nand, or...*	Gate	Combinational	n/a	n/a
Single Arithmetic Oper. *adder, comparator...*	Bit — Vector	Combinational Sequential	Operator	Modifier
Multi. Arith/Log. Oper. *ALU, data unit ...*	Bit — Processor	Combinational Sequential	Operator	Transporter
Data Transmission *multiplexer, shifter...*	Bit — Processor	Combinational Sequential	Switch Transducer	Transporter
Code Conversion *decoder, encoder...*	Vector — Processor	Combinational Sequential	Operator Transducer	Modifier
Combin. Data Transf. *parity generator, checker...*	Bit — Processor	Combinational	Control Operator	Modifier
Data Storage *register, flipflop, RAM...*	Bit — Processor	Sequential	Storage	Transporter
Controllers *clock gener., control unit...*	Bit — Processor	Combinational Sequential	Control	Modifier Transporter
Processors CPU, I/O processor...	Processor	Sequential	Processor	Transporter
Seq. Data Transf/Gener. *counter, pattern generator...*	Bit — Processor	Sequential	Control Operator	Modifier Transporter

Table 3.2: Cell Functionality and Cell Types

Heuristic knowledge about the ports of a cell is first concerned with the typical use or purpose of these ports defined by means of a few *generic functions*. The generic functions which are used for the classification of the ports are:

1. *data_in*: the values provided in the ports of this class can be used by the cell to operate with them, they can be stored in the cell or they can be used by the cell to make decisions.

2. *data_out*: these ports provide data values to other cells or the external world.

3. *control_in*: the values provided in these ports are used to set the cell to perform one of the functions that can be executed with it and therefore to determine the result presented at its outputs.

4. *control_out*: these ports provide values indicating the state of a cell or a result of an operation that can be used to make data-dependent control and sequencing decisions.

5. *power*: these ports provide the electrical power to feed the cell.

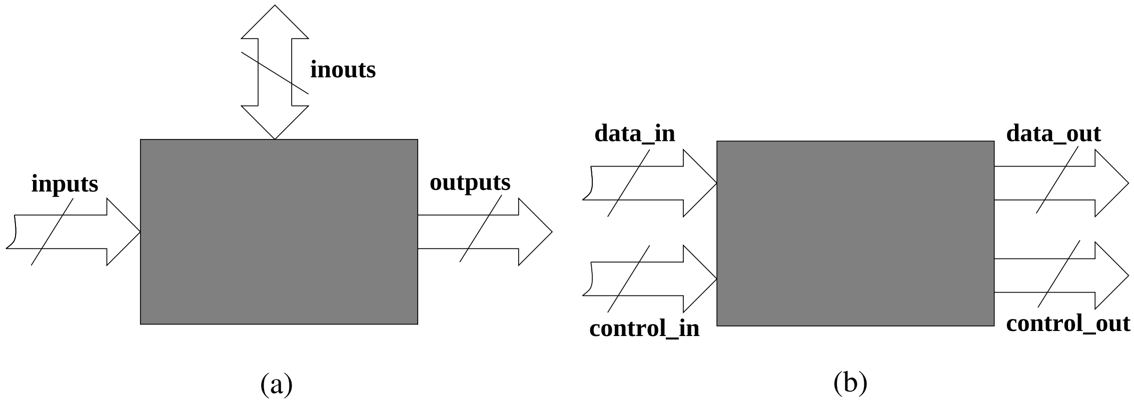


Figure 3.4: Cell Interface: (a) cell interface description, and (b) meaningful cell interface.

These generic functions are used to classify the ports of a cell in order to model its interfaces as shown in figure 3.4(b) [1]. Some classes of ports are not applicable to all levels of abstraction. For example, logic gates do not have any control ports (they perform a unique function which does not require any control). Power ports are interesting at the transistor level since they are required for the identification of logic functions (i.e. the identification of a logic gate in a transistor level design requires the consideration of the power signals). For cells at higher levels of abstraction, the consideration of the power ports of a cell does not provide any extra information about this cell since all cells must be powered.

Further heuristic knowledge about the use of the ports is defined by sub-classifying the ports which belong to a class of ports according to their *electronic functionalities*. The electronic functionality of a port refers to one of a small set of functions that a port can perform. These functions are typical of the ports of many cells and electronic designers have given them representative names. Examples of these functions are given in table 3.3 for each class of ports.

Port Class	**Port Sub-class**
data_in/data_out	data, carry, scan ...
control_in	clock, load, enable, select, address ...
control_out	overflow, ready, parity, greater_than ...
power	vcc, ground

Table 3.3: Typical Port Electronic Functionalities

[1] In the representation of figure 3.4(b) all the ports are unidirectional. Bidirectional ports in the initial description of figure 3.4(a) can be seen as duplicated into one input port (usually a 'data_in' port) and one output port (usually a 'data_out' port) in figure 3.4(b).

The knowledge about the functionality and types of a cell can be used to infer knowledge about the functionalities of its ports and vice versa. For example, a cell of type switch requires select ports to determine the transfers of data in the cell and a cell with clock ports is necessarily sequential.

iv) Cell Contents and Implementation:

Heuristic knowledge about the contents of a cell refers to the class and quantity of the sub-cells used in the construction of the cell. Knowledge about the cell can be derived from knowledge about the sub-cells and vice versa. For example, if a cell contains sub-cells with an electronic functionality of type flip-flop then the cell must be sequential and for a cell to be a register it must contain sub-cells with an electronic functionality of type flip-flop.

Knowledge about the implementation style of a cell can also be used in order to narrow the range of possibilities for the rest of knowledge about the cell. For example, the use of PSAs in the design of a cell (see section 2.2.2) implies that this cell has a logic type of sequential.

v) Data Flow Information:

According to the flow of data, electronic cells are classified as transporter or modifier cells. Transporter cells have some operation modes that allow the transfer of data from 'data_in' ports to 'data_out' ports under the control of 'control_in' ports without altering this data. The 'control_in' ports are used to determine the path used for the transfer of data. These paths are called *data transfer paths*.

The types of communication of input/output data include any combination of serial/parallel input with serial/parallel output. The type of communication of data relates to the types of the cell and the functionality of its ports. For example, a cell with serial input of data (e.g. a serial adder) is necessarily sequential and must have clock ports.

The flow of data is determined by the type of network formed by the interconnection of its sub-cells (see section 2.2.1 and section 2.2.2 for examples of these types). The type of network allows information about the types and functionality of a cell to be derived. For example, memory cells are implemented by a regular structure of 1-bit memory sub-cells and iterative networks are commonly used for the implementation of combinational arithmetic circuits (operators).

3.4 Knowledge for the Classification of Design Signals

A signal represents the information that is transmitted between interconnected ports. The heuristic knowledge about a signal of a design is often related to the functionalities of the ports involved in the transmission of the signal. As with the ports of a cell, knowledge about the signals is first concerned with their intention or purpose which is defined by one of a few generic functions. The generic functions of a signal include:

1. *data*: these signals carry the data upon which the system or cell performs its function. These signals usually come from storage cells of the system or from the external world. The operators of the system act upon these signals and the result of the operations are new data signals that are stored in storage cells or transmitted from the cell usually by means of transducers.

2. *control*: these signals control the operation performed by the cells of the system. Control signals can be external to the system, obtained from storage cells of the system or generated internally according to the current data. The destination of these signals are the cells to be controlled.

3. *condition*: these signals carry information about the state of a cell or the result of its operation. They are usually originated in the system's operators. As opposed to pure data signals, the destination of these signals are usually control cells which use them to make data-dependent control and sequencing decisions.

4. *power*: these signals are used to provide power to the cells of the system.

An example of these signals for a cell at the processor level was given in figure 2.6. Control signals are issued from cells in the control section to control cells in the data section. These signals reflect the state of the control section. Condition signals are issued from the data section to the control section to make data-dependent decisions in the control unit (for the control section, condition signals are data upon which the unit can make decisions). This approach is hierarchical in the sense that the control unit may be provided with an internal control unit in which case further internal control and condition signals can be identified inside the control cell.

The signals of a cell flow from driving or *origin* ports to driven or *destination* ports. Origin ports correspond to input port groups (data_in/control_in) of the cell and output port groups (data_out/control_out) of the instances in the cell. Destination ports correspond to output port groups of the cell and input port groups of the instances. For example, in the design of figure 3.2(a) the input port $\mathbf{A_3}$ of the cell is an origin port. The input port $\mathbf{D_3}$ of the instance **FF3** is a destination port. The signal flows in the connection from the origin port to the destination port.

The generic function of a signal (data/control/condition) is related to the generic function of the ports involved in the transfer of the signal (data_in/data_out/control_in/-control_out) and to the location of the origin and destination ports (a port is located either in the cell or in an instance). This will be discussed in section 7.4.1. As with the ports of a cell, the functionality of a signal is further sub-classified according to its electronic functionality. The set of categories that are used to represent the electronic functionality of a signal is the same as the set of categories that are used to represent the electronic functionalities of the ports. Examples of these categories were given in table 3.3.

The electronic functionality of a signal is not necessarily the same as the electronic functionality of the ports involved in the transmission of the signal. The electronic functionality of a port of a cell corresponds to the function performed by the port from the point of view of the cell to which the port belongs. The electronic functionality of a signal corresponds to the function of that signal in the cell where it is used. For example, in figure 3.5 the signals arriving to ports $\mathbf{P_1}$ and $\mathbf{P_2}$ and coming out from port $\mathbf{P_4}$ are data signals from the point of view of the multiplexer. $\mathbf{P_1}$ and $\mathbf{P_2}$ are data_in ports of the multiplexer cell. The cell selects a signal among $\mathbf{S_1}$ and $\mathbf{S_2}$ which is transferred to the data_out port $\mathbf{P_4}$. The signal coming out of port $\mathbf{P_4}$ ($\mathbf{S_4}$) is used to trigger a flip-flop cell. The signals $\mathbf{S_1}$, $\mathbf{S_2}$ and $\mathbf{S_4}$ are clock signals from the point of view of the design but some of the ports of the instantiated cells which transmit these signals are not clock ports ($\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_4}$). In the other cases, the functionality of a port and the signal related to it is the same ($\mathbf{P_3} - \mathbf{S_3}$, $\mathbf{P_5} - \mathbf{S_4}$, $\mathbf{P_6} - \mathbf{S_5}$ and $\mathbf{P_7} - \mathbf{S_6}$).

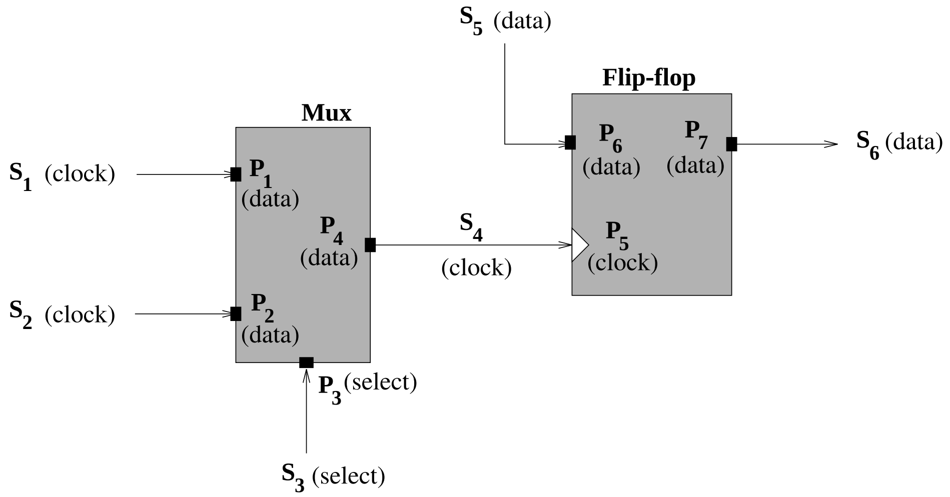


Figure 3.5: Port and Signal Electronic Functionalities

3.5 Reasoning About a Situation

An heuristic classification of the cells and signals of a design symbolises an understanding of the specification: heuristic knowledge which describes the purpose and functionality of the design objects (i.e. cells, ports and signals) is generated. The knowledge about an object must be consistent in all the situations in the design in which the object appears. Consequently, reasoning about a situation (and the design as the composition of its separate situations) implies and can result in:

1. the determination of whether the information about an object is consistent with the information about the other objects in the situation.

2. the establishment of relationships between the different objects such that information about an object can be used for the derivation of information about other objects.

3. the grouping of objects to form a more meaningful description of the situation (i.e. to gather a set of instances which can naturally be seen as forming a new cell in the hierarchy of a design and to form arrays of ports and arrays of signals).

The heuristic model of a cell in a design is used for modelling the corresponding cell node and all the instance nodes of the cell node. For example, in the design of figure 3.2(c) the node C_1 is modelled according to a model for this cell. The instances contained in this cell (nodes I_1, I_2, I_3 and I_4) are modelled according to a model for sub-cell C_2. Relationships between the knowledge which represents the cell, sub-cells and signals in the situation are established, for example, by the hierarchy of the situation, by the connectivity between objects and by the flow of signals in the situation. These relationships must be consistent according to heuristic knowledge about electronic design (see chapter 7). Thus, in situation S_1 (shown in figure 3.2(a)) the model that represents cell C_1 must be consistent with the model that represents cell C_2. For example, the clock port of an instantiated flip-flop sub-cell must be connected to the clock port of the register cell. The model of cell C_2 must be consistent in situations S_1 and S_2 (situation S_2 is shown in figure 3.2(b)). As another example, if the instance nodes contained in cell C_1 are identified as flip-flops (according to the model of C_2) the model of cell C_1 must have a logic type of sequential since the cell contains sequential sub-cells (see section 4.5).

An example heuristic model for cell C_1 is given in table 3.4. The model is a frame-like structure that collects the heuristic knowledge about the different aspects of the cell. This structure is made of a number of slots which are filled with the types of knowledge described in section 3.3. These slots are:

i – *model name*: this slot is filled with the name of the cell for which the model is derived.

ii – *cell types*: this slot is made of four sub-slots to describe the types of the cell.

iii – *functionalities and organisation of the interfaces*: this slot classifies the ports of a cell into classes according to their generic function. The ports in these classes are sub-classified according to their electronic functionalities. In the example, three sub-slots are used to describe three different classes of ports in the cell. Each class has only one sub-class in this example.

iv – *cell contents*: this slot contains sub-slots which classify the sub-cells of the cell. There is a single sub-slot in the example since all the sub-cells belong to the same class.

v – *cell electronic functionality*: this slot defines the electronic functionality of the cell.

vi – *cell constraints*: this slot establishes constraints over the variables in the model. In this example, the model is *complete* and all the slots and sub-slots are defined. A model which has undefined slots (slots filled with variables) is said to be *incomplete*. This will be considered in section 5.2.

vii – *data flow information*: this slot provides knowledge about the flow of data in the cell. In the example, the flow of data corresponds to a transfer of data between the input data ports to the output data ports under the control of the clock port.

Logged Cell Model Name: register

Cell Types	
Cell Logic	*sequential*
Abstraction Level	*vector*
DataFlow Type	*transporter*
Cell Purpose	*storage*

Cell Interface				
Port Class	Nº Signals	Port Sub-classes		
		Port Name	Port Sub-class	Nº Signals
data_in	4	$[a_3, a_2, a_1, a_0]$	*data*	4
control_in	1	ck	*clock*	1
data_out	4	$[b_3, b_2, b_1, b_0]$	*data*	4

Cell Contents		
Sub-cell Class	Sub-cell Name	Nº Sub-cells
flip_flops	d_flipflop	4

Cell Electronic Functionality: *register*

Cell Data Flow Information	
Data Transfers	(1) path: $[a_3, a_2, a_1, a_0]$ ==> $[b_3, b_2, b_1, b_0]$ control: ck

Cell Model Logged On: *Tue Nov 15 11:55:13 1991*

Table 3.4: Cell Heuristic Model Example

viii – *date log*: this slot indicates the date of generation of the model. This is used for the management of models in the system [2].

3.6 Problem Formulation

The problem of modelling the cells of a design can be seen as a problem of search. The system attempts to find a set of models $Ms = \{M_1, \ldots, M_n\}$ for the n cells of a design such that each cell C_i $(1 \leq i \leq n)$ has a model M_i that is consistent in all the situations in which C_i appears. The fact that only a partial analysis of the electronic data about the cells is carried out implies that the system cannot be absolutely certain about the model

[2]As discussed in chapter 5, knowledge about a cell can be compared with models existing in the system. A criteria for the selection of candidate models can be the date of their generation (e.g. all models generated in the last year). Information about the author or organisation that created a cell may also be convenient. Knowledge for a cell could be compared with models existing in the system which meet criteria related to their origin.

of any cell. A set of possible models Ms_i will in general be possible for each cell C_i. The best solutions are recognised by evaluating the confidence in each model generated. The task of the system includes both the generation and selection of models for the cells of the design.

The problem is characterised by an *initial state* and a *goal-state description* (according to the standard search terminology [CM85]). A state is described by a list of n elements for a design with n cells. The i-th element corresponds to the model for the i-th cell of the design. A model for a cell can be complete or incomplete. The search proceeds by transforming the initial state into another one that is expected to be closer to a *goal-state* (a state that satisfies the goal-state description). For example, the initial state may contain no model for any cell of the design and the goal-state description may be to have complete models for all cells in the design. A change of state takes place by assigning a model to a cell in the state.

A goal-state description can be seen as an existential query to the system. For example, is it possible to find a complete model M_i for each cell C_i of a design such that all models are consistent between them? In some cases, the system may be asked to look for models for a sub-set of the cells in the design. In other cases, the models of some cells may contain some information known *'a priori'*. That is, some models can be partially or fully defined from the beginning. For example, given two cells C_1 and C_2 and a model for each cell the query may want to find if these models are consistent for these cells in a given design which contains them. In addition, the problem may require finding a solution or an *optimal solution* measured in some way. Sometimes, there might not be any solution and then the object is to search until a solution is found or a solution cannot be found.

The set of states that can be reached by applying models in sequence starting at the initial state corresponds to the *search space*. The size of the search space depends on the number of models available per cell. This number depends on the information available for the cell and on the number of models that can be formed with the types of knowledge considered by the system (see section 5.3).

3.7 Complexity and Strategies

A first analysis of the complexity of the problem can be done by assuming that the possible models for each cell of a design have already been generated (this assumption will be revised below). The problem of finding the best alternative to represent the design (choosing one model for each cell of the design) is of a combinatorial nature. If n is the number of cells in the design and m_i is the number of possible models for each cell C_i, the number of alternatives that may be tried is

$$N = \prod_{i=1}^{n} m_i \tag{3.2}$$

For simplicity, considering the average number of possible models for the cells of the design given by

$$m_{av} = \frac{\sum_{i=1}^{n} m_i}{n}$$

an upper-bound for the number of alternatives N is

$$N_w = (m_{av})^n \tag{3.3}$$

This value represents the number of alternatives that will need to be considered in the worst case according to the number of models available per cell [3].

The complexity of the problem (the size of the search space) grows exponentially with the number of cells involved. For example, the analysis of a design with $n = 6$ and $m_{av} = 5$ may require the study of

$$N_w = 5^6 = 15625$$

alternatives. The problem becomes rapidly intractable for designs with a larger number of cells and several models per cell. For example, for a design with 13 cells and an average number of 5 possible models per cell, the number of alternatives to analyse in the worst case would be

$$N_w = 5^{13} = 1.2 \times 10^9$$

which is over one billion combinations for a design with only 13 cells!

To tackle this complexity, the situations of a design must be first analysed separately. A situation S_i describes cell C_i in terms of n_i sub-cells. Considering that there are m_i possible models for cell C_i and m_{ij} possible models for the j-th sub-cell of C_i $(1 \le j \le n_i)$ the average number of models per cell for this situation is given by

$$m_{av_i} = \frac{m_i + \sum_{j=1}^{n_i} m_{ij}}{n_i + 1}$$

The number of alternative representations to analyse for a situation in the worst case is

$$N_{w_i} = (m_{av_i})^{n_i+1} \tag{3.4}$$

and the number of alternatives that must be analysed for the separate situations in a design that has n cells is in the worst case

$$\sum_{i=1}^{n} (m_{av_i})^{n_i+1} \ll N_w \tag{3.5}$$

Considering that it is possible to analyse all these alternatives and that they can be ranked according to how well they represent the situation, a solution must be tried for the whole design. This involves the selection of a valid alternative for each situation

[3]This is true since $(m_{av})^n \ge m_1 * m_2 \ldots * m_n$. This is easily proved considering that $(m_{av})^2 = m_{av} * m_{av} \ge (m_{av} - m) * (m_{av} + m) = m_{av}^2 - m^2$ (note that the equality holds when all the m_i have the average value).

so that each cell has the same model in all the situations in which the cell appears. It is not possible to consider only the best alternative for each situation. This is because the best ranked alternatives for two different situations may have a different model for a cell that appears in both situations. It is then necessary to evaluate which set of coherent alternatives for the situations (an alternative per situation) is the most attractive according to a function that evaluates the quality of the representation. It is shown in chapter 6 that the complexity of this problem is not exponential with respect to the number of cells or the number of models.

The problem is that the complexity of the analysis of a situation also grows exponentially with the number of cells involved according to equation 3.4. Even for situations with just a few cells and several alternative models per cell, it will not be possible to reason about each possible representation to determine the best one. The only solution then is to evaluate the different alternatives so that the most likely ones are tried first and unpromising alternatives are pruned from the search space. A series of a few correct guesses may be enough to choose adequate models. These guesses are simulated by means of two functions: a *model-ordering function* and a *state-evaluation function*. The model-ordering function sorts the available models for each cell from best to worst candidates. The state evaluation function encodes a measure of the confidence in each alternative. This helps in the search for an optimal solution. As shown in chapter 6, the evaluation and selection of the most promising alternatives does not grow exponentially with the number of cells or with the number of models involved. The evaluation function keeps track of the most plausible state as states are generated. The algorithm can then be allowed to switch back to any previously generated state that looks more promising and it can prune states that are unpromising.

The control of the search is complicated by the fact that models for the cells are generated as the reasoning proceeds. This is because knowledge in the model of a cell is linked to knowledge in the models of other cells. For example, in the hierarchy graph of figure 3.3(b), knowledge in the model of cell C_2 in the situation $S_2 = < C_2, \{C_3, C_6\} >$ is linked to knowledge in the models of cell C_3 and cell C_6. These links imply that the knowledge in the model of a cell can be used for the derivation of knowledge in the models of related cells. This is called knowledge-propagation in section 1.3. New plans for the knowledge about a cell can be formed in this way. The comparison of new knowledge plans for a cell with system information may result in more elaborated models for the cell which before could not be generated since not enough knowledge was available for the comparison.

The knowledge in the model of a cell may allow the propagation of knowledge to any other cell in the design. For example, knowledge in the model of cell C_2 can be propagated to the model of any cell in the situation S_2 and to the model of any cell in the situation $S_1 = < C_1, \{C_2, C_4\} >$ since this situation instantiates the cell C_2. In its turn, knowledge propagated to cell C_4 through the situation S_1 may result in the propagation of knowledge to cell C_5 through the situation $S_4 = < C_4, \{C_5, C_6\} >$. In this example, knowledge in the model of cell C_2 may result in the propagation of knowledge to cell C_5 through three different sequences of situations:

$$\begin{cases} S_2 \rightarrow S_1 \rightarrow S_4 \rightarrow S_5 \\ S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \\ S_2 \rightarrow S_6 \rightarrow S_4 \rightarrow S_5 \end{cases}$$

The operation of the system is as follows. The system initially extracts a set of plans for the knowledge about the cells of the design by analysing the specification. The comparison of these plans with system information allows the generation of a set of initial models for the cells. At this stage, all the situations of the design must be examined. The most adequate alternatives to represent each situation are selected and analysed. The analysis of valid alternatives results in the propagation of knowledge between the models of the cells of a situation. The enhanced models for the cells are seen as new knowledge plans which are compared with system information in order to validate them. As a result of the comparison, more refined models may be generated. The operation of the system is an iterative process of knowledge-generation/knowledge-propagation cycles. The process terminates when it is not possible to add more knowledge to any of the models in the alternatives which best represent each situation. The process must terminate since there is a limited amount of knowledge in the system and therefore of possible plans or models that can be generated for the cells.

Part II

Automatic Derivation of Heuristic Design Knowledge

Chapter 4

Formation of Knowledge Plans

A knowledge plan represents a possible arrangement of heuristic knowledge for a cell. A plan for a cell can result in an heuristic model for the cell if it is validated by the system as discussed in chapter 5. The accuracy and effectiveness of these plans relies on the quality of the description, the knowledge-derivation functions of the system and the ability to combine planning alternatives for individual items of knowledge. The quality of a description is improved by means of techniques such as the choice of meaningful names for design objects, the use of comments, the arrangement of design objects into arrayed structures and the use of an adequate design hierarchy. All these techniques, except the use of comments, are exploited for the derivation of plans for the cells of a design. Three types of knowledge-derivation functions are identified and examples of them are presented. The complexity of the problems of forming knowledge plans and dealing with large numbers of them is addressed.

4.1 Knowledge Plans

This chapter initiates the discussion about the derivation of heuristic design knowledge. The kinds of knowledge that can be derived were described in chapter 3. The derivation of heuristic knowledge represents a high level way of reasoning for the understanding of the design and its parts. A penalty to pay with this way of reasoning is a degree of uncertainty associated with the knowledge which is derived. Because the knowledge is not totally reliable, a number of alternatives for the heuristic knowledge about a cell can generally be formulated. Each alternative arrangement of knowledge for a cell is called a *knowledge plan*. Knowledge plans are presented in this section. Heuristic methods for the formation of these plans are introduced in section 4.2. The types of functions which are required to implement different methods of deriving plans are classified in section 4.3. Examples of these functions are given in section 4.4 and section 4.5. The complexity of the problem of forming knowledge plans and the problem of dealing with a large number of them are addressed in section 4.6.

In many cases, a knowledge plan will only contain a partial arrangement of knowledge for a cell, with no propositions for some of its aspects. These kinds of plans are called *incomplete* plans since some degree of freedom is still possible for the heuristic knowledge about the cell. As opposed to this, a plan with propositions of knowledge for each aspect of the cell is called a *complete* plan. Complete plans are in general more attractive than incomplete plans. This is because if a complete plan is correct it represents a complete model of the knowledge about a cell. However, complete plans are less likely to match with system knowledge for their validation (see section 4.3) since they are very constrained (they do not contain any free variables). On the other hand, incomplete plans are more likely to be matched, but they may well match in a number of different ways all of which must be analysed. Often, it will be necessary to ignore plans which only have a few items of knowledge defined since it will not be feasible to analyse all matching possibilities.

An example of a knowledge plan is illustrated for the example circuit of figure 4.1. The object of the figure corresponds to an instance of a cell named '74als257'. The instance object is called 'ic90' in the real electronic design [Doc88]. The numbers inside the object correspond to the names given to the ports of the instantiated cell. The names outside correspond to the names given to the connections or *nets* which interconnect the ports of the instance in the design. An example of a plan for this cell which is obtained from the analysis of figure 4.1 is shown in table 4.1. The procedure for the derivation of this plan will be described in section 4.4.2 and section 4.5. The plan is seen as a frame which contains slots for arranging the heuristic knowledge about the cell.

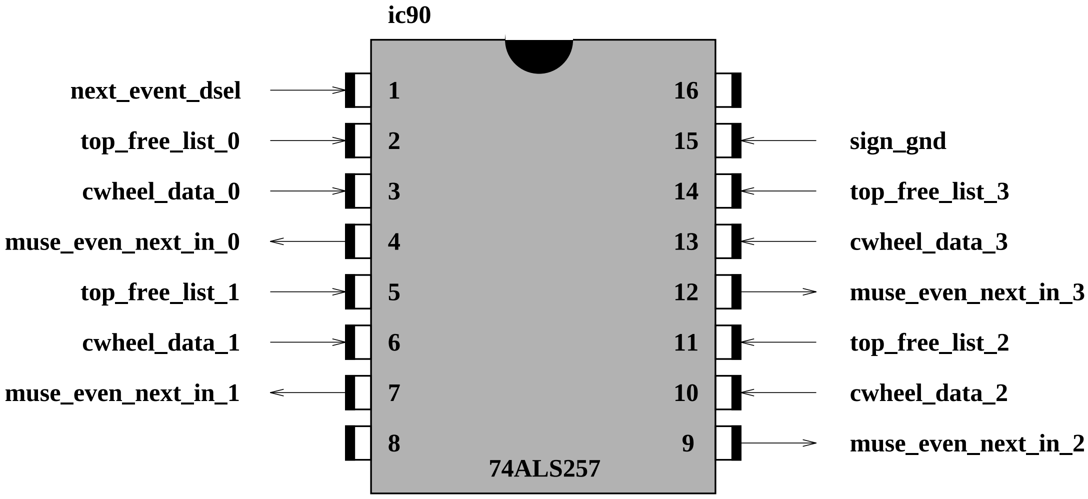


Figure 4.1: Example Circuit

The slots used to represent a knowledge plan are the same as the slots that are used to represent an heuristic cell model (see section 3.5), except that no slot is used for the date of derivation of the plan. The slots that are shown in table 4.1 are the ones that can be filled or partially filled from the analysis of figure 4.1. Examples of the form of the rest of slots are given in table 3.4 and in the tables of chapter 5. These slots are:

i– *plan name*: this slot is filled with the name of the cell to which the plan applies.

ii– *cell types*: this slot is made of four sub-slots which house the types of the cell. In table 4.1, the sub-slots are filled with variables. This indicates that the plan could

Knowledge Plan:	*74als257*

Cell Types	
Cell Logic	**A**
Abstraction Level	**B**
DataFlow Type	**C**
Cell Purpose	**D**

Cell Interface				
Port Class	Nº Signals	Port Sub-classes		
		Port Name	Port Sub-class	Nº Signals
inputs	*10*	[‘2’,‘5’,‘11’,‘14’]	*top_free_list*	*4*
		[‘3’,‘6’,‘10’,‘13’]	*data*	*4*
		‘1’	*select*	*1*
		‘15’	*ground*	*1*
outputs	*4*	[‘4’,‘7’,‘9’,‘12’]	*muse_even_next_in*	*4*
default	*2*	‘8’	**E**	*1*
		‘16’	**F**	*1*

Cell Constraints
(1) **A** *in* [*combinational,sequential*]
(2) **B** *in* [*vector,processor*]

Table 4.1: A Knowledge Plan

not resolve any of the types of the cell to any specific value. However, constraints over these variables are proposed as indicated below.

iii – *functionalities and organisation of the interfaces*: this slot classifies the ports of the cell into classes according to the direction of their signals (see section 3.3). The direction of the ports in the class *default* was not specified in the description. The ports in each class are sub-classified according to their electronic functionalities. The knowledge stored in the plan includes, for example, the fact that the port ‘1’ of the cell is assumed to have as electronic functionality ‘select’ and the fact that the four input ports [‘3’,‘6’,‘10’,‘13’] are all assumed to have as electronic functionality ‘data’ (the derivation of this knowledge takes place as examined in section 4.4.2).

iv – *cell contents*: this slot contains sub-slots which classify the sub-cells of the cell (see, for example, table 3.4). This slot is fully undefined for the example cell of figure 4.1 since this cell is a basic primitive of the design and nothing is known about its contents.

v – *cell electronic functionality*: this slot defines the electronic functionality of the cell which could not be resolved from the analysis of figure 4.1.

vi – *cell constraints*: this slot establishes constraints over the variables in the plan. The constraints given in table 4.1 specify that the level of abstraction and logic type of the cell are limited to one of the values in the sub-sets indicated (the derivation of these constraints is considered in section 4.5).

vii – *data flow information*: knowledge about the flow of data in the cell is stored in this slot. No knowledge for this slot could be derived from the inspection of figure 4.1.

The plan of table 4.1 is clearly incomplete since most of the knowledge about the cell remains undefined.

4.2 Methods and Heuristics for the Formation of Plans

Methods and heuristics for the formation of plans are introduced in this section and they are expanded as required later in the thesis. The knowledge used to form these plans (see chapter 3) is not rigorous and it is often vague in the sense that there are no clear definitions to limit its scope of application. Most important of all, this kind of knowledge is often gratuitous since it is not made explicit in a design description. However, the inclusion of this knowledge in the description of a design (in one way or another) firmly improves the quality of the description. High quality descriptions make extensive use of techniques such as the choice of meaningful names for the design objects, the use of comments, the arrangement of the design objects into arrayed structures and the use of an adequate design hierarchy. These techniques facilitate the understanding of a design but they are not, at least in principle, essential. The knowledge provided by the use of these techniques is meaningful to design engineers but it cannot be exploited by automatic systems in most cases.

Among the techniques mentioned above, it is probably the use of meaningful names which provides, at first glance, more information about the objects of a design. Following good design practice and methodology, designers use sensible names for the objects of their designs such as cells, interface ports and signals. By exploiting semantic information carried by the names of the objects of a design, designers can attempt to deduce the functionality or purpose of these objects. This belief is examined and automated in section 4.4 to form plans for the knowledge about cells and signals.

Another important technique that is used to improve the quality of a description involves the grouping or *arraying* of objects. Groups of objects tend to facilitate the analysis and description of the design by allowing a set of separate objects to be viewed as a whole. This is similar to computer programming where suitable data structures are preferred to a large number of separate variables. In electronic design, designers tend to collect together objects which share some common characteristics. For example, the ports of large cells are likely to be organised as groups of ports, as in the case of 'address' and 'data' ports (these groups often represent the bits required to represent in the binary logic implementation of the cell a value of a high level description of the cell). By the same token, connections that carry related signals are often grouped as in the case of 'address' and 'data' busses. The grouping of a set of ports or signals in the specification may be used to support an hypothesis that these objects share the same electronic functionality. Conversely, an assumption about the functionality of an object can be deemed unpromising after observing the group of objects to which the object belongs. For example, a design has zero, one or very few 'clock' signals. If a signal is assumed to be a 'clock' signal, and later it is observed that it belongs to a large array of signals, the strength of the assumption is weakened.

Groups of cells are defined by designers when forming the hierarchy for their designs. In the description of a cell, designers can make use of other cells already designed, thus describing a cell as the interconnection of a set of sub-cells. If too many instances of the sub-cells of a cell are required to represent the contents of a cell, designers may decide to group some of these instances to form a higher level sub-cell which contains these instances. This simplifies the understanding of the contents of a cell since there are fewer instances (or sub-cells) to consider and the interactions between a cell and its contents become clearer (this is expanded in section 7.7) [1].

The hierarchy of a design establishes relationships between its cells. This is because the operation of a cell must be achieved by means of the operation of its sub-cells and the interconnections between instances. As a result, knowledge about a cell can be used to derive knowledge about its sub-cells and vice versa. For example, the types of a cell are tied to the types of its sub-cells (e.g. the level of abstraction of a cell cannot be lower than the level of abstraction of any of its sub-cells). A way of planning a set of plausible types for a cell based on the types of its sub-cells is described in section 4.5. Considering that the types of a cell are related to the electronic functionality of this cell as shown in table 3.2, it is clear that knowledge about the types of the sub-cells limits the range of possible operations for the cell.

Relationships between objects in a design are also established by means of the interconnections between the cells of the design. For example, a 'clock' port of a cell often is connected to the 'clock' ports of its sub-cells. Knowledge about the interfaces of different cells becomes interrelated as a result of the connections in the design. For example, if two ports are grouped in a cell (and therefore it is possible to hypothesise that they have the same electronic functionality) and they are both connected to two separate ports of another cell, it is logical to propose that these last two ports share the same functionality. The connectivity of a design is an important factor for the derivation of knowledge plans for the interfaces of the cells. This is examined and automated in section 7.4. Knowledge about objects which are not directly interconnected can also be interrelated taking into account the flow of signals in the design. For example, the case of a single cell which issues signals to control a group of cells or a data-path can be used to interrelate knowledge about objects in the data-path which are not directly connected. This is examined

[1] Cells are also frequently grouped to form libraries of cells. The cells in a library usually share a number of characteristics, the most common of these being the technology used for their implementation. Designers may choose to organise libraries of cells according to their functionalities or their levels of abstraction. Examples of this include a library of flip-flop cells and a library of logic gates. In these cases, knowledge about one of the cells in the library helps for the examination of the rest of cells. These heuristics are not considered in the current implementation of the system.

in section 7.5.

Another important way of planning knowledge about a cell is by considering that expert designers can propose further knowledge about the cell by considering the knowledge already available about it. In other words, this implies that some items of knowledge are known to the experts as 'usually' being associated with some other items of knowledge. For example, the fact that a cell is supposed to have an input 'clock' port and input/output 'data' ports of the same width can allow a designer to support an hypothesis of the cell being a register. Operationally, the system can respond to the recognition of a pattern of knowledge for a cell by forming a new plan with additional knowledge. This is automated in section 5.5.1. Similarly, the system can enhance the knowledge representing a set of interconnected cells by comparing the knowledge available with heuristic models of common design strategies and stereotypical implementations. This is described in section 7.6.

To conclude this introduction to the heuristics and methods used for the derivation of knowledge plans, it must be noted that a large variety of heuristics can be exploited for the derivation of plans. The ones mentioned above form part of the current implementation of the system. The implementation of methods and heuristics for the derivation of plans is discussed in the next section.

4.3 Knowledge-derivation Functions

The functions for the derivation of knowledge about the cells of a design are here classified into three types as shown in figure 4.2. These functions are:

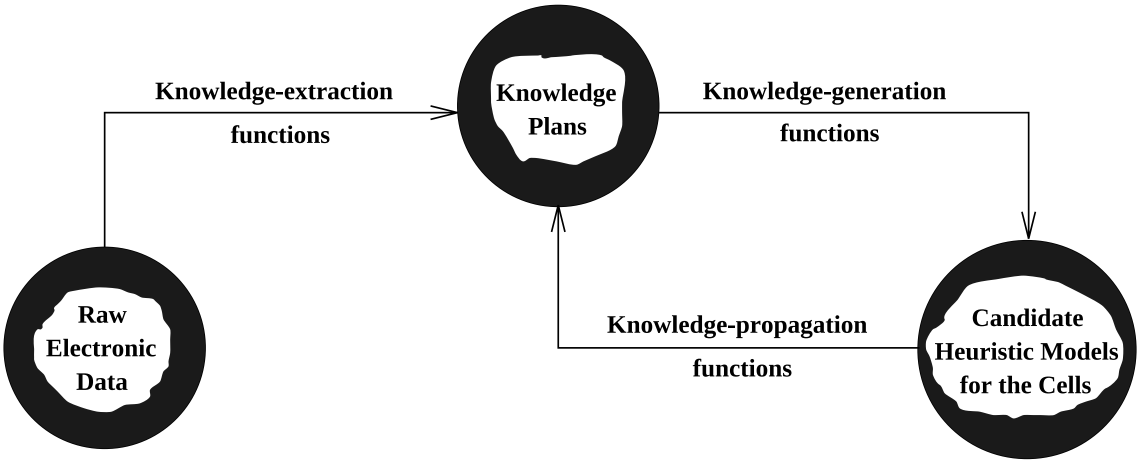


Figure 4.2: Knowledge-derivation Functions

1. *knowledge-extraction functions*: knowledge about a cell is extracted from the designer's specification by inspection of salient features of the cell present in the description. The fact that the knowledge is 'extracted' implies that information existing in the specification is used to support the validity of this knowledge. Knowledge

about a cell may be extracted from different sources. These sources include the description of the cell and each use or instantiation of this cell in the design. The knowledge extracted from each source is collected to form plans for the knowledge about a cell. A collection of extracted knowledge forms a plan since its different items of knowledge are extracted separately. The system must verify whether these items of knowledge are a compatible representation of an electronic cell.

A function $\mathcal{K}$-*ext* which extracts a set of initial plans for an arbitrary cell C_i is defined as

$$\mathcal{K}\text{-}ext: \ C_i \longrightarrow P_i^1, \ldots, P_i^q \tag{4.1}$$

where P_i^j corresponds to the j-th plan for cell C_i with $1 \leq j \leq q$. The value q is the number of different plans extracted for this cell. The knowledge-extraction function associates a strength value with each item of knowledge. This value evaluates the confidence in this item of knowledge. These values are combined to evaluate the confidence in the whole plan (see section 6.3).

2. *knowledge-generation functions*: knowledge about a cell is derived by comparison of an existing plan with system information. This includes the comparison of a plan with heuristic models which describe electronic cells or classes of electronic cells (see section 5.5.1) and the comparison of a plan with heuristic models of the use of a cell for the design of more complex cells (see section 7.6). The result of a successful comparison between a plan and one of these heuristic models is an augmented plan or *solution plan* which incorporates the heuristic knowledge contained in the model. The new plan is supported by the fact that the items of knowledge existing in the first plan are known to the system as 'usually' being associated with some other items of knowledge. The system supports the validity of the extra items of knowledge and the functions are said to generate knowledge. A solution plan is seen as a possible heuristic description of the operation of the cell and it corresponds to a candidate heuristic model for the cell (cell model). The confidence in this model depends on the number of items of knowledge present in the initial plan and the confidence in each of these items.

   A function $\mathcal{K}$-*gen* which compares a plan P_i for the cell C_i of a design with the k-th heuristic model in the system H_k is defined as

$$\mathcal{K}\text{-}gen: < P_i, H_k > \longrightarrow M_i^1, \ldots, M_i^g \tag{4.2}$$

   The comparison of plan P_i with the heuristic model H_k can generate g solution plans (a plan and an heuristic model may not match at all or match in a number of different ways). The term M_i^j with $1 \leq j \leq g$ indicates the j-th solution plan for cell C_i. A solution plan M_i^j meets the heuristic rules of the k-th heuristic model in the system. For this reason, this plan is seen as a candidate heuristic model for describing the operation of cell C_i.

3. *knowledge-propagation functions*: knowledge about a cell is derived from knowledge describing other cells and signals. The models that describe the cells of a design

have relationships between them. These relationships arise because of the design hierarchy, the connectivity of the design and the flow of signals in the design. A relationship links items of knowledge which correspond to models of different cells. Two types of relationships can be considered:

(a) a relationship that represents a *constraint* which the corresponding items of knowledge must satisfy. If the constraint is not satisfied (*consistency-checking*) the representation is incorrect according to the knowledge in the system. In this case, the set of models used for the representation must be discarded. If some of these items of knowledge are undefined, consistency may be kept by passing knowledge to the corresponding incomplete models.

(b) a relationship that represents a *plausible link* between the items of knowledge. In this case inconsistencies cannot arise in the representation but the relationship can be used to add knowledge to incomplete models. The addition of knowledge to an existing model forms a new plan for the corresponding cell.

The extra knowledge added to an incomplete model for a cell results in the formation of a new plan for the knowledge about this cell. This extra knowledge is supported by the knowledge available in the models of interrelated cells and the system. The functions are said to propagate knowledge. The new plan formed can be compared again with system information by means of the knowledge-generation functions (this forms the loop illustrated in figure 4.2).

A function $\mathcal{K}$-*prop* which propagates knowledge to a model M_i for a cell C_i that is used in the situation S_k of a design is defined as

$$\mathcal{K}\text{-}prop : < M_i, S_k > \longrightarrow P_i^1, \ldots, P_i^p \qquad (4.3)$$

where P_i^j indicates the j-th plan for cell C_i derived from the reasoning about the situation S_k and the model M_i of this cell. The value p represents the number of new plans that can be formed for cell C_i by means of this reasoning.

4.4 Example 1 — Planning by Analysis of Names

This example discusses a knowledge-extraction function which analyses names of objects of a design for the planning of knowledge about a cell. A key technique for a prompt identification of the functionality of cells and signals is based on the study of the names associated with the objects in a design. Although the name given to an object is arbitrary, the use of meaningful names is common practice. The choice of meaningful names facilitates the design of the circuit and the understanding of its contents and functionality. This is similar to computer programming where the choice of meaningful names for variables and procedures highly facilitates the understanding of a program. In the case of electronic design, the usage of names is limited to a variety of objects which include, notably, the libraries of cells, cells, ports and signals [2].

4.4.1 Heuristics Based on Naming

The planning of knowledge for a cell by an examination of names arises from the observation that the names of the design objects are frequently related to their functionalities. This observation is exploited for the investigation of the electronic functionality of a cell, the electronic functionality of its signals and, most important of all, the electronic functionalities and organisation of its interfaces. In fact, there are three different ways in which the naming of objects can be exploited:

1. knowledge can be obtained by analysis of the semantic value contained in the names of the objects. For example, the fact that a port is called 'clk' carries enough semantic information for electronic designers to guess that this port probably has the electronic functionality 'clock'. Thus, an assumption about the electronic functionality of a cell, a port or a signal can occasionally be supported by the name given to the respective object, such as a cell name (or, less frequently, the name given to an instance of this cell), a port name or the name given to the net that carries the signal. These assumptions are based on the comparison of names with typical values for the electronic functionality of cells (see table 3.2) and with typical values for the electronic functionality of ports and signals (see table 3.3) which are known to the system. In addition, names can be compared with electronic functionality values generated from the processing of previous designs. This last situation is possible since the system provides an automatic mechanism for identifying new values. The automation of the comparison of names is based on the use of semantic networks which provide a natural way of jointly storing information about names and electronic functionality values. This is described in appendix B.

2. knowledge can be captured by observing that a set of objects, mostly ports and signals, are given names which carry the same or similar semantic values, regardless of the actual meaning of these names. For example, the names of two ports such as 'top_free_list_1' and 'top_free_list_2' in figure 4.1 carry similar semantic values and, consequently, the ports are likely to have the same electronic functionality. This is reflected on a plan by grouping these two ports together. An exception to this rule occurs when all the objects considered have names with similar semantic values (e.g. all ports within a cell are called with names such as 'port_1', 'port_2', ..., 'port_n'). In this case, the semantic value of these names is assumed to be nil and the ports are not grouped.

3. knowledge can be extracted from the fact that a set of objects are tied together under the same name. For example, a set of ports are given a common name in an

[2] All these objects must have been named by one designer or another. A particular design object like a cell or a port may have a set of names associated with it. For example, a cell object has the cell name, the names of the instances of this cell and other names such as those that can be used for the display of the cell in a schematic representation.

array port. A plan can be made assuming the same electronic functionality for each port in the array since the names of the ports are often related to their electronic functionalities. Another example of this includes a net object that ties a set of ports to have the same signal. As discussed in section 3.4, the electronic functionality of a signal in a design is often the same as the electronic functionality of the ports involved in the transmission of the signal. As a result of this, the name of a signal can be used to plan the functionality of the related ports.

4.4.2 Planning the Interface of a Cell

From the above discussion, the following set of heuristics can be considered for planning the interface of a cell according to naming:

i– *"the name of a port of a cell often reflects the electronic functionality of this port".*

ii– *"if two ports have similar names they often have the same electronic functionality".*

iii– *"the name of a signal in a design often reflects the electronic functionality of this signal".*

iv– *"the electronic functionality of a signal in a design is often the same as the electronic functionality of the ports involved in the transmission of the signal"*

v– *"if two ports carry signals with similar names they often have the same electronic functionality".*

A knowledge-extraction function that makes use of these heuristics is implemented in the system. The plan for the interface section of table 4.1 was extracted from the instance object of figure 4.1. The ports are first classified according to their direction (*inputs*, *outputs* and *inouts*) as given in the specification. For the ports '8' and '16' no direction is given in the specification and they are classified into the class *default*. The ports in each class are sub-classified according to their electronic functionalities. In this example, these electronic functionalities are derived from the analysis of the names given to the nets that connect the ports of one of the instances of the cell. The analysis is as follows: the name 'next_event_dsel' is matched with the typical electronic function 'select' and a corresponding port sub-class is generated. The names 'top_free_list_0', 'top_free_list_1', 'top_free_list_2' and 'top_free_list_3' do not match any typical electronic function but they all carry similar semantic values. Thus, a port sub-class with electronic functionality 'top_free_list' and 4 elements is created. Similarly, the names 'cwheel_data_0', 'cwheel_data_1', 'cwheel_data_2' and 'cwheel_data_3' all match the typical function 'data' and they form another sub-class. A similar analysis is performed for the rest of the signal names. For the ports '8' and '16' no signal name was given in the specification and their functionality is left undefined (in fact, these ports are not connected in the specification).

4.5 Example 2 — Inference of Plausible Cell Types

Examples of knowledge-derivation functions for inferring the plausible types of a cell are examined in this section. The types of a cell represent key knowledge for the identification of the class of electronic cells to which the cell belongs. Procedurally, the system will attempt to match a knowledge plan for a cell with only those heuristic models in the system whose types are consistent with the set of plausible types derived for the cell. In order to derive the plausible types of the cell, the knowledge-extraction functions resort to information extracted from the specification; the knowledge-generation functions resort to knowledge already available for the cell; and the knowledge-propagation functions fall back on knowledge available about the types of the sub-cells of the cell. The types of a cell which are investigated here include the level of abstraction, logic and data flow types.

4.5.1 Abstraction Level Analysis

The complexities of the interfaces and contents of a cell are used to estimate a set of plausible levels of abstraction for this cell. The complexity of the interfaces is determined by the number of ports and by the electronic functionalities of these ports. On the whole, cells at higher levels of abstraction present more complex interfaces than cells at lower levels. For example, cells at the bit level have a smaller number of inputs and outputs than cells at the vector level (this is logical since cells at the vector level are often composed of arrays of cells at the bit level). Table 4.2 contains a set of typical rules or plausible relationships for the derivation of the plausible levels of abstraction for a cell from its number of ports (power ports are excluded). The table is of an heuristic nature with the rules or heuristics applying for most electronic cells.

Nº Inputs	Nº Outputs	Nº Inouts	Conditions	Abstraction Range
N	0	0	$N \geq 0$	unknown
0	M	0	$M > 0$	bit-vector
N	1	0	$N \leq 2$	transistor-bit
			$N > 2$	gate-vector
N	M	0	$N \leq 2$, $M > 1$	transistor-vector
0	0	K	$K = 1$	unknown
			$K = 2$	transistor-bit
0	1	K	$K = 1$	bit
N	M	K	$N+M+2*K \leq 10$	bit-vector
			$10 < N+M+2*K \leq 100$	vector-processor
			$N+M+2*K > 100$	processor-computer

Table 4.2: Abstraction Level Based on Number of Ports

A knowledge-extraction function implements these heuristics in order to calculate an initial set of plausible levels of abstraction. The initial set can be revised by taking into account the contents of the cell. Obviously, a cell cannot have a level of abstraction lower than the level of abstraction of any of its sub-cells. It is possible to define a function which takes into account the set of plausible levels of abstraction for each sub-cell of the cell (and the number of instances of each sub-cell) and calculates a set of plausible levels of abstraction for the cell. This function is a knowledge-propagation function since it considers knowledge about the sub-cells of the cell to derive knowledge about the cell.

A way of defining this knowledge-propagation function is to consider a function which relates the level of abstraction of a cell to the number of transistors required for its implementation. This function or *complexity estimation function* is defined in section 6.6. The derivation of the set of plausible levels of abstraction for the cell is then carried out as follows:

1. an estimate for the number of transistors (a range of values) required for the implementation of each sub-cell is obtained from the set of plausible levels of abstraction of the sub-cell by means of the complexity estimation function.

2. an estimate for the number of transistors required for the implementation of the cell is calculated from the estimates of the number of transistors required for the implementation of each sub-cell and the number of instances of each sub-cell.

3. finally, a set of plausible levels of abstraction for a cell is obtained from the estimate of the number of transistors that the cell requires by applying the complexity estimation function in reverse [3].

This knowledge-propagation function does not represent constraints which the levels of abstraction of a cell and its sub-cells must satisfy. The result of the function is an estimate of the plausible levels of abstraction. Because of this, an enlarged set is obtained from the union of the set obtained by means of the knowledge-extraction function and the set obtained by means of the knowledge-propagation function. Some unlikely options in this enlarged set can still be excluded with the aid of heuristics which keep in view the type of electronic functionality of the ports. These heuristics consider that some types of ports are only expected within a range of levels of abstraction. Examples of these heuristics include:

i– "if a cell has ports with electronic functionality address then the abstraction level of the cell is vector or higher".

ii– "if a cell has ports with electronic functionality clock then the abstraction level of the cell is bit or higher".

These heuristics represent constraints which the set of plausible levels of abstraction of a cell must satisfy with respect to other knowledge available for the cell. The heuristic

[3]It must be noted that the complexity estimation function assumes that cells at higher levels of abstraction require a larger number of transistors for their implementation than cells at lower levels. Because of this, the plausible levels of abstraction which are estimated for the cell cannot be lower than the levels of abstraction of any of its sub-cells.

model of any electronic cell must comply with these constraints which are represented in the system hierarchy of heuristic cell models (see section 5.3). The knowledge-generation functions which compare knowledge plans with heuristic models of electronic cells will then take into account these constraints.

4.5.2 Logic Type Analysis

Knowledge for the elucidation of the logic of a cell can be drawn from the examination of the interface and contents of the cell. With regard to the interface, synchronous sequential circuits are easily identified by the presence of ports with electronic functionality 'clock'. It is also worth mentioning that cells at levels of abstraction higher than vector level are often sequential.

The investigation of the sequential nature of a cell according to its contents is done as indicated in table 4.3 (X means any value). This table indicates that a cell that contains sequential sub-cells is necessarily of a sequential type. For a cell with no identified sequential sub-cells an analysis of the type of network is necessary. This analysis is based on the fact that a logic network with feed-back loops is almost always sequential (see section 2.2.1). Conversely, a loop-free network is in most cases combinational [4].

Cell Logic	Flat Cell	Sequential Sub-Cells	Logic Feed-back Loops
Combinational	no	no	no
Sequential	no	yes	X
Sequential	no	X	yes
Unknown	no	unknown	no
Unknown	no	unknown	unknown
Unknown	yes	X	X

Table 4.3: Sequentiality Analysis

The analysis of feed-back loops is adequate for networks of cells at the gate-vector levels of abstraction. This analysis is not applicable to networks of transistor cells and, as mentioned above, cells at levels higher than vector often are sequential. The analysis of feed-back loops is not carried out if there are bidirectional ports interconnected in the network since these networks are almost always sequential. The example of figure 4.3 is used to illustrate this circumstance. In this circuit, when the signal **ctr** carries a logic value 0 the bidirectional port $\mathbf{IO_1}$ gets programmed as an output and the bidirectional port $\mathbf{IO_2}$ is programmed as an input. Then, when the signal **ctr** carries a logic value 0 there are no loops in the network and the circuit is combinational if the sub-cells are combinational. On the other hand, when the signal **ctr** carries a logic value 1 the port

[4]In some technologies the sequential nature of a circuit (and the consequent storage of data) is provided by a periodical refreshment of the circuit and no electrical loops are used.

$\mathbf{IO_1}$ gets programmed as an input and the port $\mathbf{IO_2}$ is programmed as an output. In this case, there is a feed-back loop and the network is most probably sequential.

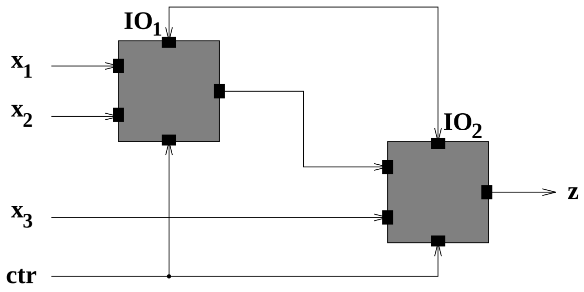


Figure 4.3: A Network with Bidirectional Ports

4.5.3 Data Transportation Analysis

The investigation of the transfer of data in a cell is based on the types of its sub-cells and their interconnections as indicated in table 4.4. In general, if a cell contains sub-cells of a transporter type, and it is possible to transport some data from the input ports of the cell to its output ports through the transporter sub-cells, the cell can be planned as being of a transporter type. The fact that some sub-cells must be of a transporter type restricts the analysis to cells whose network is at the bit level or higher.

Data Flow Type	Transporter Sub-Cells	Data Transfer Paths
Modifier	no	no
Modifier	yes	no
Transporter	yes	yes

Table 4.4: Data Transportation Analysis

Operationally, the procedure for a cell with transporter sub-cells consists of searching for the paths that allow the transfer of data from input to output ports. An example of this is illustrated by means of the circuit in figure 4.4. The transfers of data in a sub-cell are indicated by dashed lines. The data transfer paths are defined by the origin data ports, the destination data ports and the control ports required to set up the transfer. For the above example, the data transfer paths are:

$$\left.\begin{array}{lllll}
(1) & \text{path}: & \mathbf{x_1} & ==> & \mathbf{z_1}, \quad \textit{control}: [\mathbf{ck_1}, \mathrm{s}, \mathbf{ck_2}] \\
(2) & \text{path}: & \mathbf{x_1} & ==> & \mathbf{z_2}, \quad \textit{control}: [\mathbf{ck_1}, \mathrm{s}, \mathbf{ck_2}] \\
(3) & \text{path}: & \mathbf{x_2} & ==> & \mathbf{z_1}, \quad \textit{control}: [\mathbf{ck_2}, \mathrm{s}] \\
(4) & \text{path}: & \mathbf{x_2} & ==> & \mathbf{z_2}, \quad \textit{control}: [\mathbf{ck_2}, \mathrm{s}]
\end{array}\right\}$$

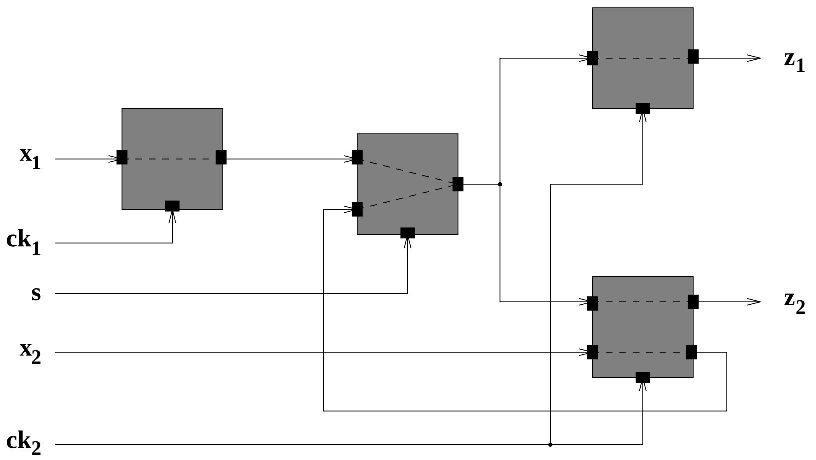


Figure 4.4: Computation of Data Transfer Paths

The computation of paths for the transfer of data in a cell provides a mechanism for consistency-checking between cells and sub-cells. The data transfer paths defined in the model of a cell must be attainable with the sub-cells instantiated according to their associated models.

4.6 Number of Knowledge Plans and Complexity

An important issue for the derivation of knowledge plans corresponds to the combination of knowledge derived for individual aspects of a cell to form a knowledge plan for the whole cell. For example, the plan of table 4.1 for the cell of figure 4.1 combines knowledge which is independently derived: the knowledge for the interfaces of the cell (derived according to the knowledge-extraction function of section 4.4) and the knowledge for the types of the cell (derived according to the knowledge-derivation functions of section 4.5). The result of the combination is a more complete plan. As mentioned in section 4.1, detailed plans are often necessary in order to make the matching with heuristic models feasible. However, a more detailed plan may exclude possibilities which separate plans would not exclude (in the case that the knowledge contained in the plan is incorrect). On the other hand, separate plans (with either knowledge for the types of the cell or knowledge for its interfaces) may be too sparse to be matched with heuristic models. In addition, the items of knowledge (sub-slots) contained either in the slot for the types of the cell or in the slot for its interfaces are independently defined and combined to give a single knowledge arrangement for each slot.

A large number of planning possibilities are excluded in this way (considering that some items of knowledge may be incorrectly defined). Unfortunately, there is no general way to alleviate this problem. There are two ways of exhaustively considering all the knowledge derived:

1. by generating all knowledge plans which can be obtained by combining all planning possibilities for the individual items of knowledge in a plan. This clearly gives rise

to a combinatorial explosion: the number of knowledge plans which can be derived grows exponentially with the number of items of knowledge considered, leading to an unmanageable number of knowledge plans. This is very often the case considering, for example, that a cell has several ports and the fact that knowledge about the electronic functionality of a port can be derived from different sources (such as each use of the cell in the design).

2. by collapsing, for each item of knowledge, all the planning possibilities to generate a constraint which does not exclude any alternative. For example, the electronic functionality of a port may be constrained to a value within a set of possible values. Thus, constraints (usually weak) for the separate items of knowledge can be calculated and a single plan (or a smaller number of plans) generated. However, plans with weak constraints and few fully represented items of knowledge are difficult to match (see section 5.5.2).

These difficulties imply that more heuristic methods for the formation of knowledge plans must necessarily take over. Some of these methods depend on the knowledge-derivation functions used. For example, the knowledge-extraction function described in section 4.4 is applied to each instance of a cell in the design, but the results obtained from the analysis of each separate instance may not be combined together. More general heuristic methods are aimed at managing a large number of knowledge plans. This is illustrated for the case of knowledge-generation functions (the same applies for the other kinds of functions). The matching of a knowledge plan P_i with the k-th heuristic model in the system H_k as given by equation 4.2

$$\mathcal{K}\text{-}gen : < P_i, H_k > \longrightarrow M_i^1, \ldots, M_i^g$$

can generate a large number g of solution plans (see section 5.5.2 and equations 5.5 and 5.6). A restricted knowledge-generation function which only considers one solution plan from the comparison of a knowledge plan with an heuristic model is defined as

$$\mathcal{K}^*\text{-}gen : < P_i, H_k > \longrightarrow M_i^* \tag{4.4}$$

The function $\mathcal{K}^*$-gen can be seen as a composition of functions

$$\mathcal{K}^*\text{-}gen \; = \; \mathcal{F} \circ \mathcal{K}\text{-}gen$$

where the function $\mathcal{F}$ is defined as

$$\mathcal{F} : < M_i^1, \ldots, M_i^g > \longrightarrow M_i^* \tag{4.5}$$

The function $\mathcal{F}$ is a filtering function which takes all plans generated by $\mathcal{K}$-gen and produces a single solution. Two major alternatives can be considered for the definition of the filter function $\mathcal{F}$:

1. to only consider the solution plan for which a highest confidence evaluation is obtained. The knowledge-derivation function associates a confidence value with each

item of knowledge and these values are combined to evaluate the confidence in the whole plan (see section 6.3). Knowledge plans which are bound to give lower confidence values can be discarded without having to generate all individual plans. The disadvantage of this choice is that the selection of a single plan (or several plans for that matter) definitively excludes the other possibilities.

2. to collapse all solution plans into a *most general plan* (or *most general model*) M_i^* which does not exclude any of the solutions and contains all knowledge which is common to all plans. In this way, each plan M_{ij} is an instance of M_i^* (since some slots that are undefined in the most general plan are filled with values). The procedure used for the determination of the most general plan is based on *variabilisation* [CM85]. This considers substituting values in the slots of the plans by variables so that the plans do not exclude each other. The variables can then be constrained according to the knowledge available in the collapsed plans.

Finally, some constraints for the knowledge about some cells can be imposed at the beginning of the reasoning process or during the processing (e.g. by the user of the system). Any heuristic model or plan for a cell must inherit these constraints which can reduce the number of possible knowledge plans to be considered. A large number of plans can be managed efficiently by organising them in a hierarchical way as discussed in the next chapter.

Chapter 5

Generation of Cell Models

A model for a cell is generated from the comparison of a knowledge plan with system information about electronic cells. The procedure is frame-based: a frame that contains a plan for the knowledge about a cell is compared with system frames. A system frame is either a class model, which represents an heuristic model of a class of electronic cells, or a cell model that corresponds to an heuristic model of an individual cell. These models can be provided by the system or obtained from the processing of past designs. A successful comparison of a plan for a cell with a system frame results in heuristic models for the cell. Since a number of plans may be available for any cell, and a plan can match various system frames in a number of different ways, a number of different alternative models may be generated for each cell. System frames and the procedure for matching them are described. The organisation of models and plans is discussed. The evaluation and selection of models for the cells of a design are discussed in chapter 6.

5.1 Knowledge Plans and Heuristic Models

This chapter continues with the derivation of knowledge for a cell by considering the comparison of knowledge plans with system information about electronic cells. The result of this comparison is the addition of some items of knowledge to an initial plan to form a more detailed plan. The resulting plan can be a complete plan if enough knowledge is provided in the initial plan. Since these extra items of knowledge are provided by the system the functions that perform this comparison are knowledge-generation functions. The new plan is supported by the fact that the items of knowledge existing in the initial plan are known to the system as 'usually' being associated with some other items of knowledge. The strength value that evaluates the new plan depends on the number of items of knowledge present in the initial plan and their associated strength values (the combination of the strength values of different items of knowledge that form a plan to evaluate the strength of the whole plan is discussed in section 6.3). On the whole, the system responds to the recognition of a pattern of knowledge for a cell by forming a new plan with this knowledge and additional knowledge provided by the system. The strength value of the new plan is higher than the strength value of the initial plan.

There are basically two kinds of system information that can be compared with the knowledge for a cell. The first type of information corresponds to heuristic models of the operation of electronic cells. Each one of these models contains a set of heuristics which describe an electronic cell or a class of electronic cells. The result of a successful comparison between a plan for a cell and one of these models is a solution plan which incorporates the heuristic knowledge contained in the model. Such a plan is seen as a possible heuristic description of the operation of the cell or *cell model*. The second type of information that can be used for the comparison of a knowledge plan concerns heuristic knowledge about the use and applications of electronic cells. This is examined in chapter 7.

The subject of this chapter is the generation of knowledge by comparison with *class models* and *cell models* existing in the system. The frame structure of a class model is discussed in section 5.2. Cell models were discussed in section 3.5. Class and cell models can be classified as either *system models* or *logged models*. A *logged model* corresponds to a model obtained from the processing of past designs. Section 5.3 discusses the hierarchical organisation of class and cell models in the system. Section 5.4 discusses the logging of new models into the system. The function used for the comparison of a knowledge plan for a cell with a class model or a cell model is presented in section 5.5. The result of this function is the generation of possible cell models for the cell. The process is illustrated in figure 5.1. The evaluation of a resulting cell model will be considered in section 6.3. Finally, section 5.6 discusses the organisation of knowledge plans in the system.

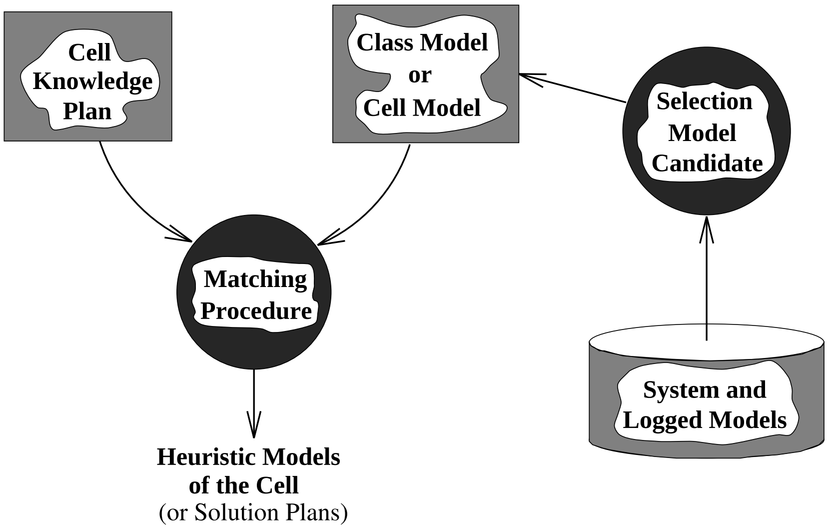


Figure 5.1: Generation of Heuristic Cell Models

5.2 Class Models

A class model is a frame structure that gathers heuristic knowledge about a class of electronic cells. This knowledge concerns heuristics and rules that apply to all electronic cells that are members of the class. Any cell that satisfies the heuristics and rules of a class model can be classified as a member of the class. An example of cells which are members of a class of cells is given in figure 5.2. This example is used for illustrating the representation of a class model. The kinds of knowledge used to build models for electronic cells were described in chapter 3.

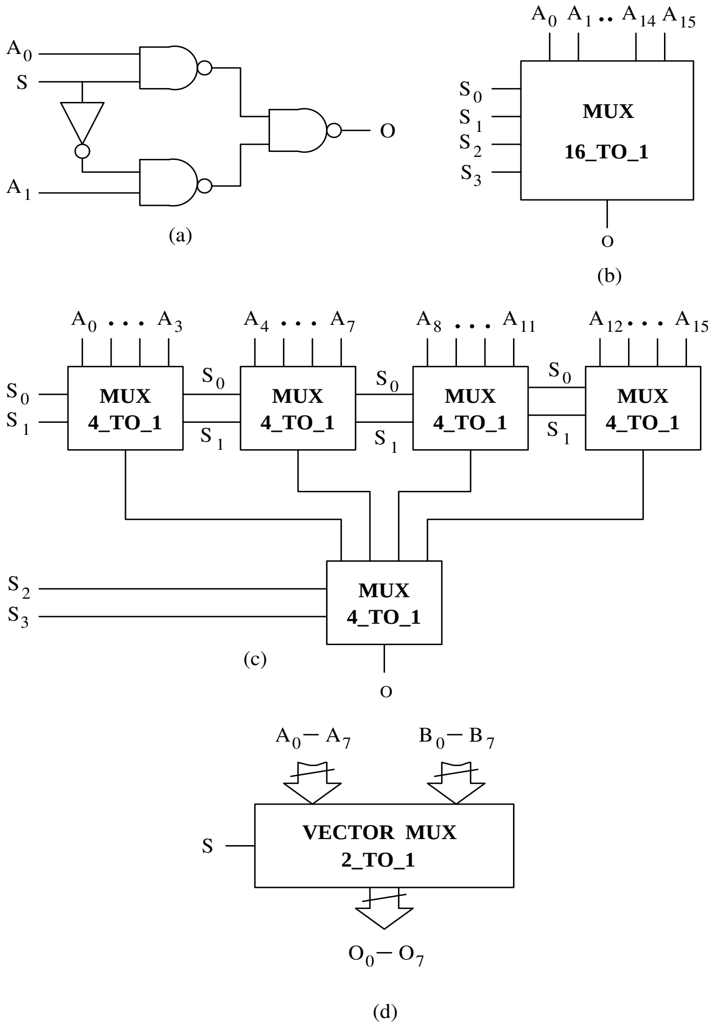


Figure 5.2: Multiplexer Cells: (a) a construction of a 2_to_1 multiplexer, (b) a 16_to_1 multiplexer, (c) a construction of a 16_to_1 multiplexer, and (d) a 2_to_1 vector-multiplexer.

All cells in figure 5.2 are examples of *multiplexer* cells. A multiplexer is a basic building block for the design of electronic circuits. The function of a multiplexer is to establish a path for the transfer of data. For example, the cell in figure 5.2(a) is a '2_to_1_multiplexer': the signal **S** of the cell is used to decide which one of the two input data bits **A0** and **A1** is transmitted to the output line. In the figure, this kind of multiplexer is implemented by means of logic gates. An example of a larger multiplexer is shown in figure 5.2(b). This multiplexer requires four input signals in order to decide which one of the sixteen data input bits is transmitted to the single data output line. This multiplexer is constructed in figure 5.2(c) by means of a combination of '4_to_1_multiplexers'. A final example of a member of this class of cells is shown in figure 5.2(d). This example corresponds to a vector-multiplexer which allows the transfer of bit-vectors. The multiplexer selects which of the two input bit-vectors is transferred as the output bit-vector.

A class model is represented by an 8-tuple. This tuple is viewed as containing eight top level slots which contain knowledge about the class of cells. These slots are the same as the slots used to represent cell models in section 3.5 and knowledge plans in section 4.1. The slots of a class model are filled with *terms*. A term can be a constant (an atom or an integer), a variable (which stands for some definite but unidentified value associated with a slot), a term that is viewed as a number of sub-slots or a mathematical or logical expression. These expressions are heuristic rules that constrain the variables used in the model. The expressions are written using the syntax of the logic programming language Prolog [CW88]. The class model used to represent the class multiplexers is shown in table 5.1. The slots used to represent a class model are:

i – *class name*: this slot provides a name for the class of cells. In the example, all cells which have a model derived from this class are said to belong to the class *multiplexers*.

ii – *cell types*: this slot contains four sub-slots to describe the types of the cells which are members of the class. The example of table 5.1 specifies that all members of the class have the same logic, data flow and purpose types as defined in the model. The level of abstraction, however, depends on each member of the class as indicated by the variable **A**. Constraints for the value of any variable in the model are indicated in the sixth slot.

iii – *functionalities and organisation of the interfaces*: this slot describes the interfaces of the members of the class in terms of the number of ports which they can have and the generic and electronic functionalities of these ports. The ports of a cell are first clustered into classes according to their generic functionalities. The ports in a class are further organised into sub-classes of ports. The ports in each sub-class have the same electronic functionality. In the example, the number of ports in all these collections (classes and sub-classes) is left undefined since different multiplexers may have different numbers of ports in each collection. For example, a multiplexer has a number of select ports **F** which depends on each member of the class (the filling of the sub-slot which corresponds to the number of select ports with the integer value 2, for instance, would imply that all members of this class ought to have 2 select ports). Another sub-slot is used to hold the names of the ports. A name sub-slot references the ports that are allocated into this slot for a given matching plan as discussed in section 5.5.1.

iv – *cell contents*: this slot defines the classes of sub-cells that can be used to implement the cells of the class. In the example, a multiplexer can be constructed from a number of logic gates (such as the one in figure 5.2(a)), from a number of other

Cell Class Name: *multiplexers*

Cell Types	
Cell Logic	*combinational*
Abstraction Level	**A**
DataFlow Type	*transporter*
Cell Purpose	*switch*

Cell Interface				
Port Class	Nº Signals	Port Sub-classes		
		Port Name	Port Sub-class	Nº Signals
data_in	**B**	**Cs**	*data*	**B**
control_in	**D**	**Es**	*select*	**F**
		Gs	*enable*	**H**
data_out	**I**	**Js**	*data*	**I**

Cell Contents		
Sub-cell Class	Sub-cell Name	Nº Sub-cells
multiplexers	**Ks**	**L**
logic_gates	**Ms**	**N**

Cell Electronic Functionality: *multiplexer*

Cell Constraints	
(1)	$\mathbf{B} > \mathbf{I}$
(2)	$\mathbf{D}$ is $\mathbf{F} + \mathbf{H}$
(3)	$\mathbf{I} > 0\,,\mathbf{B} \bmod \mathbf{I} =:= 0$
(4)	$\mathbf{B}$ is $2 ** \mathbf{F}$; $\mathbf{O}$ is $\mathbf{B}//\mathbf{I}$, $\mathbf{O}$ is $2 ** \mathbf{F}$
(5)	$\mathbf{I} > 1 \rightarrow \mathbf{A} = \textit{vector}$; $\mathbf{A} = \textit{bit}$
(6)	$\mathbf{L} > 0$; $\mathbf{N} \geq 4$
(7)	$\mathbf{H} \leq 2$

Cell Data Flow Information	
Data Transfers	(1) $\mathbf{B}$ is $2 ** \mathbf{F} \rightarrow$ $\forall \mathbf{P}$, $\mathbf{P} \in \{1 - \mathbf{B}\}$, path: $\mathbf{Cs} - [\mathbf{P}] ==> \mathbf{Js}$, control: $[\mathbf{Es}, \mathbf{Gs}]$
	(2) $\mathbf{O}$ is $\mathbf{B}//\mathbf{I}$, $\mathbf{O}$ is $2 ** \mathbf{F}$, $\mathbf{I} = \backslash = 1 \rightarrow$ $\forall \mathbf{Q}$, $\mathbf{Q} \in \{1 - \mathbf{O}\}$, path: $\mathbf{Cs} - [\mathbf{Q}] ==> \mathbf{Js}$, control: $[\mathbf{Es}, \mathbf{Gs}]$

Cell Class Created On: *Wed Nov 6 15:30:24 1991*

Table 5.1: Heuristic Model of the Class of Multiplexer Cells

multiplexers (such as the one in figure 5.2(c)) or from the combination of these two styles. The values that indicate the number of sub-cells in each class **L** and **N** are left undefined since various combinations of multiplexers and logic gates can be used to form a multiplexer. Constraints over the variables **L** and **N** are given in the sixth slot. The sub-slots for the names of the sub-cells reference the sub-cells that are allocated into each class for a given plan that matches the model.

v– *cell electronic functionality*: this slot defines the electronic functionality of the cells that belong to this class. A class model will often define a set of cells that have the same electronic functionality. The example of table 5.1 defines the class *multiplexers* as a set of cells whose electronic functionality is *multiplexer*.

vi– *cell constraints*: this slot indicates a set of constraints over the variables in the frame. The constraints in the example of table 5.1 indicate:

1– in a multiplexer, the number of data input ports **B** must be greater than the number of data output ports **I**.

2– the number of control input ports **D** is given by the number of select ports **F** plus the number of enable ports **H**.

3– the number of data output ports **I** must be greater than 0 *and* (as indicated by the operator ',') it must be possible to classify the **B** data input ports into an entire number of groups of **I** signals each. This is expressed by means of the mathematical modulo operation indicated by the operator *mod* (each one of these groups can be transported to the output since there are **I** output signals). The operator '=:=' expresses that the numerical values are equal.

4– this constraint has two possible alternatives (separated by the operator ';'). The first one indicates that the number of select signals **F** is enough to address any of the **B** possible data inputs. The number of signals that can be addressed with **F** bits is 2 to the power of **F** (as indicated by the operator '**'). The operator *is* evaluates the expression '2 ****F**' and assigns the result to **B** (this implies that the contents of **B** must be the same as the result of evaluating the expression '2 ****F**'). The second alternative considers the case of vector-multiplexers for which a group of signals is transported to the output (see figure 5.2(d)). **O** is the number of groups of signals that can be formed with the **B** input signals considering **I** signals in each group (as many signals as in the output). That is, **O** is the integer division between **B** and **I** as indicated by the operator '//'. The previous constraint guarantees that the remainder of the division is zero. Then, the number of select signals **F** must be enough to select any of these groups.

5– If there is more than one output data line the multiplexer must be used to select bit-vectors. Therefore, the level of abstraction must be vector. Otherwise, there is a single output line. The multiplexer can transfer only one of the input lines and it must have a level of abstraction of bit.

6– To construct a multiplexer, a number of smaller multiplexers **L** (**L** > 0) can be used (see figure 5.2(c)). If the multiplexer is built with logic gates, at least 4 of them are required (see figure 5.2(a)). These constraints are weaker than in reality for simplicity (other possible implementations are not reflected on this model).

7– a multiplexer may be enabled by some control signals **H**. It is not expected to have multiplexers with more than two enable signals.

vii– *data flow information*: this slot contains a representation of the flow of data for the members of the class. In the example, since these cells are of a transporter type, the data flow information includes the transfer of data in the cell. There are two alternatives for the transfer of data. The first case considers multiplexers at the bit level (as explained above these cells satisfy the constraint $\mathbf{B}$ *is* $2 ** \mathbf{F}$). A data transfer can be set up between any of the **B** data input ports referenced in $\mathbf{C_s}$ and the data output port referenced in $\mathbf{J_s}$. This is represented in table 5.1 by considering that the list **Cs** has a total of **B** components. The **P**-th component (**P** must be in the range $\{1 - \mathbf{B}\}$) is addressed as $\mathbf{Cs} - [\mathbf{P}]$. The control of the path is determined by the control input ports referenced in $\mathbf{E_s}$ and $\mathbf{G_s}$. In the second case, the cell is a vector-multiplexer and data transfers can be set up between any of the **O** groups of data input signals and the group of data output signals. This is represented in table 5.1 by considering that the list **Cs** has a total of **O** components. The **Q**-th component (**Q** must be in the range $\{1 - \mathbf{O}\}$) is addressed as $\mathbf{Cs} - [\mathbf{Q}]$. The control of the path is determined by the same signals as before.

viii– *date log*: this slot indicates the date of creation of the model (for a system model) or the date of generation of the model (for a logged model).

5.3 Hierarchy of Models

The organisation of models in the system is based on the use of *instances* and *sub-classes*. As described above, a class model embodies a conjunction of knowledge with free variables (undefined values). Sub-classes and instances of a class are obtained by giving values to some of the variables in the model. The values of the resulting model must satisfy the constraints of the model. The resulting model represents a sub-class of the cells which are represented by the class model if some variables still remain (a sub-class model is incomplete). The resulting model represents an instance of the class of cells if all the variables in the class model are given values (an instance model is complete). This complete model represents a particular *instance* of the class model because it is satisfied with defined values for all the heuristics and rules of the class. An incomplete model represents a *sub-class* of the models of the class because its variables may be satisfied by different instances with different values.

By definition, B is a *sub-class* of A if any instance of class B is an instance of class A. This is the case for the classes A and B of figure 5.3(a). The reverse case will not apply in general since not all the instances of class A will be instances of class B. By definition too, two classes A and B are said to be *mutually exclusive* if any instance of class A cannot be an instance of class B and vice versa. For example, the classes B

and C in figure 5.3(a) are mutually exclusive. On the contrary, the classes A and B in figure 5.3(b) overlap (some instances of class A are also instances of class B).

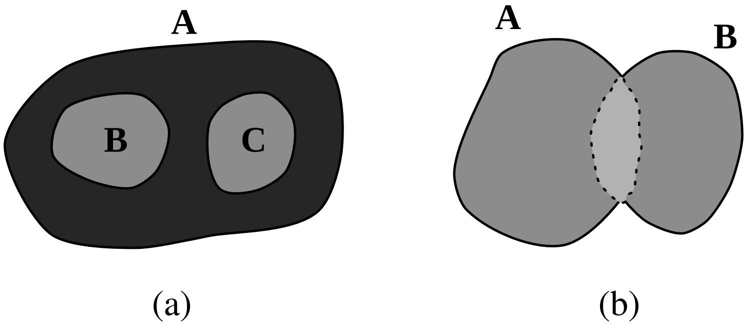


Figure 5.3: Classes and Sub-classes of Cells: (a) mutually exclusive classes, and (b) overlapped classes

The models in the system are organised as a hierarchical tree of classes by considering instances and sub-classes. The only condition for this is that all sub-classes of a class must be mutually exclusive. This condition guarantees that the models in the system are organised as a tree structure. An example of a partial hierarchy of electronic cells is shown in figure 5.4. The nodes of the tree are classes of cells. Nodes which are at higher levels in the hierarchy represent more general classes than nodes at lower levels. The branches of the tree represent links of type *isa*. The head of an *isa* link indicates a super-class of the tail of the link or sub-class. Links of type *inst* can be associated with each node. The head of this link points to a class and the tail corresponds to a model which is an instance of this class. This type of organisation is an example of an *isa* hierarchy [CM85]. The distinction between *isa* and *instance* links is clear. While *instance* says that a cell is a member of some class of cells, *isa* says that one class is a more general version of another. Plainly, the distinction is like that in set theory between element and sub-set.

The node at the top of the hierarchy represents the most general class model that is possible. It defines the different aspects of electronic cells that the system is concerned with (e.g. types, interface and contents slots) but it gives no propositions for any of them. Only constraints are possible for this class model. For example, the number of data and control interfaces of any cell must be greater than zero (a cell with no data or control interfaces is of no use). Constraints which apply to all electronic cells, such as those which relate the level of abstraction of a cell to the electronic functionality of its ports (see section 4.5.1), should be represented in this class model. The hierarchy of models must represent all heuristic knowledge about individual electronic cells. Any electronic cell must match the top class model and it is therefore an instance of this class.

The instances of the top class are grouped into two mutually exclusive sub-classes: combinational cells and sequential cells. These sub-classes are in turn sub-classified. For example, sequential cells can be synchronous or asynchronous. Clock ports must be used for the synchronisation of a cell. That is, all 'clocked' cells belong to the class of synchronous sequential cells. The model for this class of cells is given in table 5.2. The

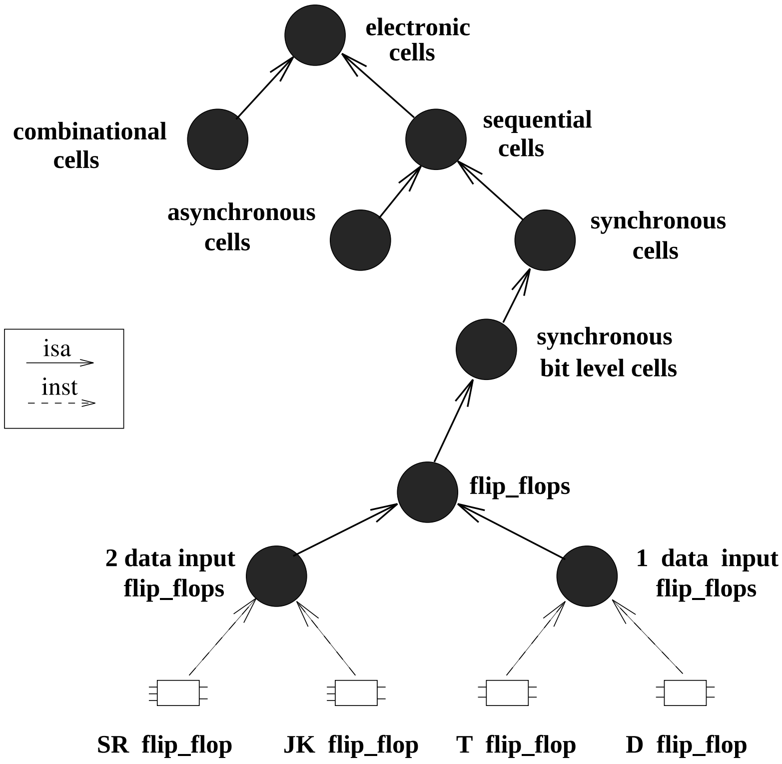


Figure 5.4: Partial Hierarchy of Electronic Cells

first constraint in the model indicates that these cells have a level of abstraction of bit or higher. The model shows that a salient feature of these cells is the existence of at least one clock port (as indicated by the second constraint). Some other ports must also exist as indicated by the third constraint. The value *default* in the model represents all other possible values except the ones indicated.

There are different ways to construct the tree of models. These ways differ in the aspects of a cell that are considered first in order to discriminate between models. A sensible implementation is to discriminate between models by using first the knowledge about the types of the cells, next the knowledge about the functionality of the cell, knowledge about the interfaces, knowledge about the contents and, finally, the data flow information. For example, in the case of figure 5.4 all the synchronous cells at the bit level form a sub-class. This sub-class is mutually exclusive with any other sub-class whose level of abstraction is not bit. An example of bit level synchronous cells are flip-flops [1]. Figure 5.4 discriminates between flip-flops by considering the number of data inputs. Flip-flops are then sub-classified by considering flip-flops which have two data inputs and flip-flops with a single data input.

The hierarchy of classes guarantees the *inheritance of properties* in the tree. For example, any cell that matches the model of a flip-flop cell must also match the model

[1] There is some confusion regarding an universal definition of flip-flop [McC86]. For most people, a flip-flop is a synchronous device [EL85] that stores one bit of information.

Cell Class Name: *synchronous_cells*

Cell Types	
Cell Logic	*sequential*
Abstraction Level	**A**

Cell Interface		
Port Class	Port Sub-classes	
	Port Sub-class	Nº Signals
control_in	*clock*	**B**
	default	**C**
default	**D**	**E**

Cell Constraints
(1) $\mathbf{A} \in bit^{+}$
(2) $\mathbf{B} > 0$
(3) $\mathbf{C} + \mathbf{E} > 1$

Cell Class Created On: *Fri Nov 8 12:23:14 1991*

Table 5.2: Synchronous Sequential Cells

of a synchronous sequential circuit. This is true because the sub-classes of a model and the instances of the model inherit the knowledge given for a class. For example, if a class states that its members are of a logic type *sequential* then a model for any sub-class of this class will only include cells of this logic type.

5.4 Logged Models and Learning

A class model embodies a conjunction of knowledge with free variables. When a knowledge plan for a cell satisfies the requirements of a class model, values are given to some of its free variables and a model for the cell is formed. Two different situations can arise from here:

1. if all the variables are instantiated with values then the resulting instantiated class model forms a complete model of the cell. The complete cell model is logged in the system as an instance of the class model. The model is called a logged cell model.

2. if some of the variables remain undefined after the matching process then the resulting instantiated class model forms an incomplete model of the cell. The incomplete cell model is logged in the system as a sub-class of the class model since different instances may satisfy the model in different ways. The model is called a logged sub-class model.

In the system, a class (sub-class) model represents a set of electronic cells and an instance model is intended to represent a single cell that is a member of a class. Of course, the instance model only represents some salient aspects of real electronic cells. Because of this limitation, two different real electronic cells can be represented by the same instance model. This situation arises when a knowledge plan of a new electronic cell matches a logged cell model and this model is accepted as valid for this new cell. The logged cell model was created in the past as a result of the matching of a cell of a design. In this case, either both cells are the same or the cell model applies to two different cells. If the cells are proved to be different, it may be possible to *generalise* the logged cell model into an incomplete sub-class model from the differences observed between the electronic cells (for example, observing that both cells differ in their implementation patterns or in the flow of data between their ports).

The generalisation is possible if the differences observed can be represented with the types of knowledge used by the system. In this way, a model that was an instance of a class of models becomes a sub-class of this class. The new model is incomplete with respect to the differences observed between the cells. The logged cell model of each cell is formed by adding the differences to the new sub-class model. As an example, if in figure 5.4 a new cell matches the cell model of the existing *SR flip-flop* cell, and this new cell has a different type of contents than the cell for which the model was derived, the cell model is transformed into a sub-class model with two instances. These instances differ by having different values in at least one of the slots of the contents which must be undefined for the sub-class model. Logged cell models facilitate the exploitation of knowledge acquired in the past and the determination of whether a cell is already known to the system.

5.5 Cell Model Generation

The generation of an heuristic model for a cell is based on the comparison of a knowledge plan with class models and existing cell models. The selection of the models that can be used for the comparison is facilitated by the fact that the models are hierarchically organised. The problem of generating models for a cell is seen as a search problem. The search space is the tree of models in the system. The procedure traverses the nodes of the tree searching for those nodes that match the plan. It starts at the top of the tree and proceeds top-down. As a consequence of the property-inheritance mechanism associated with the tree, once a node is matched the system can proceed with the comparison of the sub-tree of this node. If it is not possible to match with a certain node then the sub-tree of this node can be ignored.

Despite the fact that the sub-nodes of a node are mutually exclusive by construction, the matching of a sub-node does not automatically discard the consideration of the other sub-nodes of the node. This is because a plan is often incomplete, and the information that makes two nodes mutually exclusive may have been left undefined in the plan. In this case, the undefined data can be resolved differently for the matching of each node. If the data that makes two sub-nodes mutually exclusive is given in the plan, the matching of a sub-node will permit the investigation of the other sub-nodes to be ignored. In the best case, the matching of a complete plan will involve a single path in the tree. Therefore, it is clear that more detailed plans often require the traversal of a smaller number of nodes. The way in which the tree is traversed (depth-first or breadth-first) is irrelevant since all the solutions for a plan must be investigated.

A solution for a plan implies the matching of the plan with a class model or a cell model. The procedure for matching with both kinds of frames is identical since a cell model is an instance of a class model. The matching of a cell model usually is easier than the matching of the class to which it belongs. This is due to the extra knowledge available in the cell models which reduces the number of alternatives to be considered. For this reason, it is possible to think that the best strategy would be to consider first the matching with the cell models stored in the system. However, this will be impractical if a large number of cell models are stored. The hierarchical organisation of models eases significantly this problem. A class model is always tried before its cells models are attempted. Once the class model has been matched the cell models in this class can be tried.

5.5.1 Matching of Class Models

A plan is said to match a class model if the knowledge existing in the plan can be embedded into the class model in such a way that the constraints of the plan and the model are satisfied. Some of the variables in both the plan and the model will be given values. These values must not be inconsistent with the constraints imposed over the variables. As an example, the knowledge plan of table 4.1 matches the class model *multiplexers* of table 5.1. This example is used to illustrate the matching procedure. The slots matched include the cell types, cell interfaces, cell contents, cell electronic functionalities, the data flow information slots in both frames and the name of the cell in the plan with the electronic functionality of the model.

The contents in the sub-slots of the types of the plan must unify [Rob65] with the contents in the sub-slots of the types of the cells of the class. Table 5.3 describes this. The variables of the knowledge plan have been primed to avoid clashing with variables in the class model. The unification of the contents of the slots for the type of logic in both frames gives the result $\mathbf{A}' = \mathit{combinational}$. As shown in the table, this variable is restricted by the first constraint of the plan to a value of *combinational* or *sequential*. The result obtained satisfies this constraint. The unification of the slots for the level of abstraction gives $\mathbf{B}' = \mathbf{A}$. In addition, the fifth constraint of the class model relates the level of abstraction $\mathbf{A}$ of any instance of the class to its number of data output signals $\mathbf{I}$. As will be shown below, the matching of the interface of the frames gives the result $\mathbf{I} = 4$ and therefore $\mathbf{B}' = \mathbf{A} = \mathit{vector}$. The unification of the last two sub-slots gives a value for the type of data flow and purpose of the cell.

The matching of the slots for the interfaces of both frames implies that the sub-classes of ports in the plan must be allocated into the sub-classes of ports in the class model. In the plan, ports are classified according to the direction of the signals: *inputs*, *outputs*, *inouts* and *default*. In the model, the ports are classified according to their generic

Cell Types	
Cell Logic	$\mathbf{A}'$
Abstraction Level	$\mathbf{B}'$
DataFlow Type	$\mathbf{C}'$
Cell Purpose	$\mathbf{D}'$
Cell Constraints	
(1) $\mathbf{A}'$ in [*combinational,sequential*]	
(2) $\mathbf{B}'$ in [*vector,processor*]	

(a) Knowledge Plan

Cell Types	
Cell Logic	*combinational*
Abstraction Level	$\mathbf{A}$
DataFlow Type	*transporter*
Cell Purpose	*switch*
Cell Constraints	
(5) $\mathbf{I} > 1 \rightarrow \mathbf{A} = vector\,;\mathbf{A} =\ bit$	

(b) Class Model

$$
(c) \left\{ \begin{array}{ll}
\mathbf{A}' = combinational & \text{(satisfies constraint [1])} \\
\mathbf{B}' = \mathbf{A} = vector & \text{(since } \mathbf{I}\text{=4 from the matching of the interface as} \\
 & \text{shown in table 5.4. Satisfies constraints [2] and} \\
 & \text{[5])} \\
\mathbf{C}' = transporter & \\
\mathbf{D}' = switch &
\end{array} \right.
$$

Table 5.3: Matching of Cell Types: (a) plan for the cell types, (b) types for the cells of the class, and (c) matching result

function: *data_in*, *control_in*, *data_out*, *control_out* and *default*. The allocation of port sub-classes is as follows:

- a sub-class of the port class *inputs* (*outputs*) in a plan must be passed to one sub-class of the *data_in* (*data_out*), *control_in* (*control_out*) or default port classes in the model.
- a sub-class of the *inouts* port class in a plan must be passed to one sub-class of the *data_in* or *control_in* port classes in the model and to one sub-class of the *data_out* or *control_out* port classes in the model, or to the *default* port class in the model.
- a sub-class of the *default* port class in a plan must be passed to the *default* port class in a model.

The decision about which port sub-class in a model is selected for the allocation of a sub-class from the plan is based on the comparison of the electronic functionalities of both sub-classes. The allocation of a sub-class in the *default* class in a model implies that this sub-class is ignored for the matching.

This is illustrated for the example considered in table 5.4. For instance, the sub-class of the plan with the four ports ['2','5','11','14'] contains the value 'top_free_list' for its electronic functionality. Since this is a sub-class of the *inputs* port class, it can be allocated in the *data_in*, *control_in* or *default* port classes of the model. To decide where to allocate it, the value for its electronic functionality is matched against the values given in the sub-classes in the model. The name matching takes places as indicated in

Cell Interface				
Port Class	Nº Signals	Port Sub-classes		
		Port Name	Port Sub-class	Nº Signals
inputs	10	$[‘2', ‘5', ‘11', ‘14']$	*top_free_list*	4
		$[‘3', ‘6', ‘10', ‘13']$	*data*	4
		$‘1'$	*select*	1
		$‘15'$	*ground*	1
outputs	4	$[‘4', ‘7', ‘9', ‘12']$	*muse_even_next_in*	4
default	2	$‘8'$	**E’**	1
		$‘16'$	**F’**	1

(a) Knowledge Plan

Cell Interface				
Port Class	Nº Signals	Port Sub-classes		
		Port Name	Port Sub-class	Nº Signals
data_in	**B**	**Cs**	*data*	**B**
control_in	**D**	**Es**	*select*	**F**
		Gs	*enable*	**H**
data_out	**I**	**Js**	*data*	**I**

Cell Constraints
(1) $\mathbf{B} > \mathbf{I}$
(2) $\mathbf{D}$ is $\mathbf{F} + \mathbf{H}$
(3) $\mathbf{I} > 0\,,\, \mathbf{B}\ \mathit{mod}\ \mathbf{I} =:= 0$
(4) $\mathbf{B}$ is $2 ** \mathbf{F}$; $\mathbf{O}$ is $\mathbf{B}//\mathbf{I}$, $\mathbf{O}$ is $2 ** \mathbf{F}$
(7) $\mathbf{H} \leq 2$

(b) Class Model

$$\left.\begin{array}{r} \mathbf{Cs} = [[‘2', ‘5', ‘11', ‘14'], [‘3', ‘6', ‘10', ‘13']] \\ \mathbf{Es} = ‘1' \\ \mathbf{Gs} = ‘15' \\ \mathbf{Js} = [‘4', ‘7', ‘9', ‘12'] \\ \mathbf{B} = 8,\ \mathbf{D} = 2,\ \mathbf{I} = 4,\ \mathbf{F} = 1,\ \mathbf{H} = 1 \\ \mathbf{E}' = ?,\ \mathbf{F}' = ? \end{array}\right\}$$

Cs, **Es**, **Gs** and **Js** contain the name of the ports matched with each slot. **B**, **D**, **I**, **F** and **H** satisfy the constraints. **E’** and **F’** remain undefined.

(c) Matching Result

Table 5.4: Matching of Cell Interface: (a) plan for the cell interface, (b) interface for the cells of the class, and (c) matching result

appendix B. In this case, because the value 'top_free_list' does not match any of these values, a deterministic decision for this sub-class can not be taken. Instead, the only possibility is to try the different possible allocations (*data*, *select*, *enable*) or ignore the sub-class successively. Table 5.4 reflects the choice of allocating this sub-class into the *data* sub-class of the *data_in* port class. The next sub-class in the knowledge plan has the four signals ['3','6','10','13'] with a value for the electronic functionality of 'data'. This value is matched in the first sub-class of the model. The rest of the sub-classes in the plan are allocated in a similar way with the exception of the sub-classes of the *default* port class in the plan which are passed to the *default* port class of the model (as explained in section 4.4.2, the direction of the ports in the *default* port class of the plan was not specified and they are unconnected in the real design).

The possibility of allocating a sub-class of ports to a *default* class in the model (in practice ignoring these ports for the matching) is often necessary for some ports (see section 5.5.2 for limitations of this). Examples of these ports are *power* ports and ports that are left *unconnected* in all the uses of a cell in a design. Occasionally, it is possible to ignore single ports that are actually used. Designers often customise new cells which only introduce additional capabilities to those offered by standard cells. For example, a register can be provided with additional interface ports which allow elaborate ways of enabling the operation of the register or controlling the loading of data, but the main features of a register must be preserved. For example, input and output data-vectors of identical widths and a clock line (for synchronous registers) must be present in any instance of a register. After classifying some ports in the *default* port class, it may be possible to match the typical rules of the model of a register. The resulting incomplete model may still allow some reasoning about the cell and its environment.

The fact that different possible allocations must be successively tried reflects the *generate and test* approach taken for the matching of the interface. Once a possible allocation of sub-classes has been generated the constraints of the model and the constraints of the plan will test the validity of the combination. Table 5.4 reflects a choice for the allocation of the sub-classes which satisfies the constraints related to the interfaces. This is, in fact, one of two possible solutions as shown in section 5.5.2.

The matching of the contents slots of both frames is considered after the matching of the interfaces. This matching requires the allocation of the groups of sub-cells in the plan into the classes of sub-cells in the class model. In the example class model of table 5.1 two classes of sub-cells are considered: *multiplexers* and *logic_gates*. The allocation of a group of sub-cells in a plan into a class of sub-cells in a model requires that the sub-cells referenced match this class. Because it is possible to have plans for a sub-cell that match different classes, different allocations may be possible for each group of sub-cells in the plan. A *generate and test* approach is again used for the matching of the contents. Once a possible allocation of sub-cells has been generated the constraints of the model and the plan must be satisfied. In the example, the cell is a primitive of the design and no contents is specified. Therefore, the contents slot of the class model is left undefined.

It is also possible to ignore the allocation for a group of sub-cells. The justification for this is similar to the case used for ignoring sub-classes of ports. For some applications, the contents of a typical cell may be further complicated. However, some similarities between the contents of a standard design and the new design may still be found. For example, additional sub-cells may be necessary to provide a register with additional capabilities, but a typical set of bit level storage cells should still be found in the new cell.

The next step for the matching of a knowledge plan with a class model concerns the comparison of the slots for the electronic functionality of the cell. In the example, this gives the result $\mathbf{F}' = multiplexer$. The generation of a value for the electronic functionality of a cell is usually obtained by matching a system model. Occasionally, a plan may contain constraints about the value of the electronic functionality of the cell. These constraints can be obtained, for example, from the relationships between the types of a cell and its electronic functionality as given in table 3.2. For the cases in which the electronic functionality of a cell is undefined in the plan and defined in the model, the comparison of the name of the cell as given in the plan with the value for the electronic functionality of the class of cells is used to provide further evidence. The heuristic that justifies this comparison considers the fact that the name of a cell often reflects its electronic functionality (such as the example cell of figure 5.2(a) called '2_to_1_multiplexer'). The matching of names takes place as indicated in appendix B. In the example, the name given in the plan of the cell '74als257' corresponds to the name of the electronic cell. This name does not match the value for the electronic functionality 'multiplexer'. Nevertheless, if the cell is in the end considered to be a multiplexer, the name '74als257' is stored in the semantic networks and it is related to the word *multiplexer*.

The comparison of the slots for the data flow information is the next step considered for the matching of both frames. This implies that the flow of data represented in the plan is consistent with the flow of data as represented in the model. The plan of table 5.1 does not include any information about the flow of data in the cell.

In summary, the comparison of the knowledge plan with the class model *multiplexers* gives rise to two different solutions which vary in the organisation of their interfaces. The solution used in the discussion of this section instantiates the class model as summarised in table 5.5 (the other solution will be shown in the next section). In the instantiated model the variables related to the contents of the cell remain undefined. The model generated from the matching is then incomplete and it corresponds to a logged class model. The constraints related to the contents slot in the class model are passed to the logged class model. This model is given in table 5.6 (it must be observed that in the generated logged class model the slots for the names of the ports are substituted by variables in the interface and data flow slots).

5.5.2 Algorithm Complexity

The complexity of the algorithm for the matching of knowledge plans with system frames depends on the number of combinations required for the matching of the interface and contents slots of the frames, the complexity of the unification algorithm [Rob65] and the complexity of the algorithm for name matching (see appendix B). The number of combinations that can be tried for the matching of the interfaces depends on the number of sub-classes of ports in both frames. This is calculated as follows:

Cell Class Name: *multiplexers*

Cell Types	
Cell Logic	*combinational*
Abstraction Level	**vector**
DataFlow Type	*transporter*
Cell Purpose	*switch*

Cell Interface				
Port Class	Nº Signals	Port Sub-classes		
		Port Name	Port Sub-class	Nº Signals
data_in	**8**	[['**2**','**5**','**11**','**14**'], ['**3**','**6**','**10**','**13**']]	*data*	**8**
control_in	**2**	'**1**'	*select*	**1**
		'**15**'	*enable*	**1**
data_out	4	['**4**','**7**','**9**','**12**']	*data*	**4**

Cell Contents		
Sub-cell Class	Sub-cell Name	Nº Sub-cells
multiplexers	**Ks**	**L**
logic_gates	**Ms**	**N**

Cell Electronic Functionality: *multiplexer*

Cell Constraints
(1) $\mathbf{8} > \mathbf{4}$
(2) $\mathbf{2}$ is $\mathbf{1} + \mathbf{1}$
(3) $\mathbf{4} > 0 \, , \mathbf{8} \bmod \mathbf{4} =:= 0$
(4) $\mathbf{2}$ is $\mathbf{8} // \mathbf{4}$, $\mathbf{2}$ is $2 ** \mathbf{1}$
(5) $\mathbf{4} > 1 \rightarrow$ **vector** $=$ *vector*
(6) $\mathbf{L} > 0 \, ; \, \mathbf{N} \geq 4$
(7) $\mathbf{1} \leq 2$

Cell Data Flow Information	
Data Transfers	(2) $\mathbf{2}$ is $\mathbf{8}//\mathbf{4}$, $\mathbf{2}$ is $2 ** \mathbf{1}$, $\mathbf{4} = \backslash = 1 \rightarrow$ $\forall \mathbf{Q}$, $\mathbf{Q} \in \{1 - \mathbf{2}\}$, path: $[[\text{'}\mathbf{2}', \text{'}\mathbf{5}', \text{'}\mathbf{11}', \text{'}\mathbf{14}'], [\text{'}\mathbf{3}', \text{'}\mathbf{6}', \text{'}\mathbf{10}', \text{'}\mathbf{13}']] - [\mathbf{Q}]$ $==>$ $[\text{'}\mathbf{4}', \text{'}\mathbf{7}', \text{'}\mathbf{9}', \text{'}\mathbf{12}']$, control: $[\text{'}\mathbf{1}', \text{'}\mathbf{15}']$

Cell Class Created On: *Wed Nov 6 15:30:24 1991*

Table 5.5: Instantiated Model

Logged Cell Class Name: *74als257*				
Cell Types				
Cell Logic	*combinational*			
Abstraction Level	*vector*			
DataFlow Type	*transporter*			
Cell Purpose	*switch*			
Cell Interface				
Port Class	Nº Signals	Port Sub-classes		
		Port Name	Port Sub-class	Nº Signals
data_in	*8*	**As**	*data*	*8*
control_in	*2*	**Bs**	*select*	*1*
		Cs	*enable*	*1*
data_out	*4*	**Ds**	*data*	*4*
default	*2*	**Es**	**F**	*2*
Cell Contents				
Sub-cell Class	Sub-cell Name	Nº Sub-cells		
multiplexers	**Gs**	**G**		
logic_gates	**Hs**	**H**		
Cell Electronic Functionality: *multiplexer*				
Cell Constraints				
(1) $\mathbf{G} > 0$; $\mathbf{H} \geq 4$				
Cell Data Flow Information				
Data Transfers	(1) $\forall \mathbf{Q}$, $\mathbf{Q} \in \{1-2\}$, path: [**As**] – **Q** ==> **Ds**, control: [**Bs,Cs**]			
Cell Class Logged On: *Tue Nov 15 12:00:30 1991*				

Table 5.6: Heuristic Model Result

- n_{di} and n_{ci} are the number of sub-classes of the *data_in* and *control_in* port classes in a class model, respectively. The number of sub-classes of ports in the class model that have ports with direction input is:

$$m_i = n_{di} + n_{ci}$$

- n_{do} and n_{co} are the number of sub-classes of ports of the *data_out* and *control_out* port classes in a class model, respectively. The number of sub-classes of ports in the class model that have ports with direction output is:

$$m_o = n_{do} + n_{co}$$

- n_i, n_o and n_{io} are the number of sub-classes of ports of the *inputs*, *outputs* and *inouts* port classes of a plan, respectively. The number of sub-classes of ports that can feed signals into the cell is:

$$p_i = n_i + n_{io}$$

and the number of sub-classes of ports that can get signals out of the cell is:

$$p_o = n_o + n_{io}$$

Given that each of the p_i sub-classes of ports that feed signals into the cell can be allocated into any of the m_i sub-classes that have ports with direction input in a model or ignored, and that each of the p_o sub-classes that get signals out of the cell can be allocated into any of the m_o sub-classes of output ports in a model or ignored, the number of combinations I_c that can be tried for the matching of the interfaces is given by:

$$I_c = (m_i + 1)^{p_i} * (m_o + 1)^{p_o} \tag{5.1}$$

It must be noted that the sub-classes of the *inouts* port class in a plan are allocated into two sub-classes of a model at the same time. This is because bidirectional ports may feed signals into the cell and get signals out of it. For the example in table 5.4: $m_i = 3$, $m_o = 1$, $p_i = 4$ and $p_o = 1$. Then, the number of combinations to try is

$$I_c = (3 + 1)^4 * (1 + 1)^1 = 512$$

This value represents an upper bound on the number of combinations that need to be tried. Some of the combinations are ignored as follows:

1 – if a sub-class in a plan with a defined value for its electronic functionality matches a defined sub-class in the model, the rest of the combinations for this sub-class in the plan are not tried. In the example, the sub-classes *data* and *select* both have matches in the class model. This reduces the number of combinations to:

$$I'_c = (3 + 1)^2 * (1 + 1)^1 = 32$$

The value I'_c refers to the number of combinations that must be tried after allowing for this. The rate

$$I_e = \frac{I_c - I'_c}{I_c} \tag{5.2}$$

is an indicator of the effectiveness of the knowledge contained in the slots of the interface. For the example, $I_e = 0.9375$ which indicates that more than 93% of the combinations have been discarded in this way. The closer the rate is to one, the higher the quality of the organisation of the interfaces in the plan. It must be noted that if the value of I'_c is too large it will not be feasible to attempt to match the plan with the model. As an example, the matching of a class model with $m_i = 5$ and $m_o = 4$ with a plan with $p_i = 6$ and $p_o = 4$ would require

$$I_c = (5 + 1)^6 * (4 + 1)^4 = 29\,160\,000$$

combinations. Then, if the value for I_e is too low, the number of combinations I'_c will still be too large to make the matching of the interfaces feasible.

2– there must be a limit on the number of ports that can be ignored. For example, a typical limit is not to ignore more than 25% (see section 8.3) of the total number of ports. In the example, of the three sub-classes that must still be allocated ([‘2’,‘5’,‘11’,‘14’], [‘15’] and [‘4’,‘7’,‘9’,‘12’]) only the sub-class [‘15’] can be ignored. After allowing for this, the number of possible combinations is reduced to:

$$I''_c = (3+1)^1 * (3)^1 * (1)^1 = 12$$

3– finally, some class models can have a defined number of ports for some of their sub-classes or classes of ports. This limits the number of ports that can be allocated into a sub-class of ports or a class of ports. This is not the case, however, in the example considered.

Of the 12 remaining combinations, only two of them meet the constraints of the model. One of these combinations was indicated in table 5.4. The other combination involves ignoring the allocation of the sub-class represented by the port ‘15’. This port corresponds in the model of table 5.4 (and in reality) to an enable port. The class model *multiplexers* of table 5.1 allows a multiplexer cell to have a number of enable ports smaller or equal to 2. The solution of table 5.4 considers the model with one enable signal. If port ‘15’ is ignored, the model has no enable signals. In summary, the matching of the interfaces of the plan and the model has two different solutions. The solution in table 5.4 is a complete plan for the interfaces while the other solution is an incomplete one. The first solution is preferred since it involves a valid allocation of more ports (the matching of more sub-slots). This is reflected on a higher evaluation for this plan as discussed in section 6.3.

For the matching of the contents, the number of combinations that can be tried depends on the number of groups of sub-cells in both frames. Given m_g classes of sub-cells in a class model and p_g groups of sub-cells in a plan, an upper bound on the number of combinations is given by:

$$C_c = (m_g + 1)^{p_g} \tag{5.3}$$

The equation reflects the fact that each group of a plan can be allocated into any of the classes of sub-cells in the model or ignored. In practice, some of the combinations can be discarded as follows:

1– the allocation of a group of sub-cells in the plan to a sub-cell class in the model can take place if the referenced sub-cells match this class. If the matching is not possible, this allocation is ignored. The number of combinations that must be tried after considering the possible matches of the sub-cells is denoted by C'_c. The rate

$$C_e = \frac{C_c - C'_c}{C_c} \tag{5.4}$$

is an indicator of the effectiveness of the knowledge available about the sub-cells of the cell. The closer the rate is to one, the less combinations need to be tried. This usually indicates that more knowledge is available about the sub-cells.

2– there must be a limit on the number of sub-cells that can be ignored. To calculate this, an equivalent number of transistors can be estimated for the implementation of each sub-cell (see section 6.6). A typical limit is not to ignore sub-cells which amount to more than 25% of the total number of estimated transistors (see section 8.3).

3– finally, the number of sub-cells in a class of sub-cells can be defined for some class models. For example, in a register the number of flip-flops must be the same as the number of data lines. This limits the number of groups of sub-cells that can be allocated into a class of sub-cells.

Because the interface and contents slots are the only ones that allow for a number of different combinations, the total number of combinations f_c that may need to be tried for the matching of a plan is given by:

$$f_c = I_c * C_c = (m_i + 1)^{p_i} * (m_o + 1)^{p_o} * (m_g + 1)^{p_g} \tag{5.5}$$

The number of plans than can be derived from the matching between a knowledge plan and a model is then:

$$f_g = I_c^* * C_c^* \tag{5.6}$$

where I_c^* indicates the number of valid combinations for the matching of the interface and C_c^* the number of valid combinations for the matching of the contents. In the example, since $I_c^* = 2$ and $C_c^* = 1$, the number of plans derived is $f_g = 2$.

5.6 Organisation of Knowledge Plans

The plans derived for the cells of a design are organised in the hierarchy of models. This facilitates the managing of plans and the detection of newly derived plans which were already considered. This organisation is illustrated by means of the example of figure 5.5. In the figure, a plan matched with the class model H_1 in the tree in two different ways. These two solutions are represented in the tree by the plans P_1 and P_2. That is, two possible instances of the class model H_1. These plans represent specialisations of the plan that was used for the matching. This plan (which is not shown in the figure) must be linked in the tree to this class or to a class that is higher in the hierarchy.

The fact that a plan matches a class model implies that the matching of the instances of this model (logged cell models) and sub-classes of this class can be considered. In the example, the plan matches the logged cell model L_1 which is seen as solution P_3. This solution is different from solutions P_1 and P_2. As discussed before, the fact that a logged cell model is matched implies that either this cell is already known to the system or that the logged cell model should be converted into a sub-class model.

The plan may match one or more sub-classes of class H_1 (one at the most if the plan is complete). In the example, the plan matches class H_2 and gives the solution plan P_4. Due to the inheritance of properties, the solution P_4 is a more refined plan than solutions P_1 and P_2. On the other hand, solution P_3 may be a complete plan since an instance has been matched. Each one of these solution plans represents a cell model.

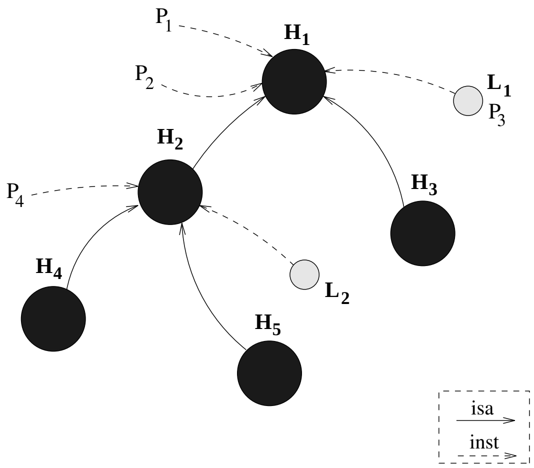


Figure 5.5: Tree of Plans

The filtering functions described in section 4.6 can be applied in order to reduce the number of plans generated as a result of the matching or to select the most convenient plan. The matching example of section 5.5.1 which gives two possible solution plans illustrates this. The two plans differ in the allocation of the port '15'. In the solution of table 5.5 this port is defined as having as electronic functionality the value enable, and this port is undefined in the second solution. For the selection of a single solution plan, an alternative is to consider a most general plan which includes both solutions. This plan must leave undefined the electronic functionality of port '15' so that no possibilities are excluded. The second solution corresponds in this case to the most general plan of the two solutions (the first solution is in fact an instance of the second). Another alternative is to consider the plan with the highest evaluation value. This corresponds to the first solution since a larger number of sub-slots are positively matched. However, this plan definitely excludes other possibilities for the electronic functionality of port '15' which are possible in the second solution.

Chapter 6

Selection of Cell Models

An alternative for the representation of a situation implies the selection of a model for each one of the cells involved. The number of possible alternatives that may be considered grows exponentially with the number of cells. To handle this complexity, the intention is to allow the system to choose which alternatives should be treated first in a way that a priori should do better than chance. An evaluation function is used for this. This function requires an evaluation of the confidence in the models of each cell involved in the situation and an estimate of the physical complexity of each cell according to each one of its models. The evaluation function can select the most attractive alternatives to represent a situation without having to generate all of them. The evaluations of the different situations are combined to evaluate the representation of the overall design. This allows the comparison of alternative solutions. Model-based reasoning is applied to the alternatives selected for the representation of each situation as described in chapter 7.

6.1 Problem Definition and Complexity

The task of modelling the cells of a design was defined as a search problem in chapter 3. The system attempts to find one model for each cell of the design which is consistent in all the situations in which the cell appears. Since each cell may have several different models, a number of alternatives can be attempted for the representation of each situation. Each alternative representation of a situation (called a *candidate set* for the representation of the situation) considers one model for each cell involved. As shown by equation 3.4, the number of candidate sets for the representation of a situation grows exponentially with the number of cells involved. Even for situations with just a few cells and several possible models per cell, it will not be feasible to generate each possible candidate set and to apply model-based reasoning to it.

The only possible solution is to heuristically evaluate the alternatives for each situation and to select the most attractive ones without having to generate all of them. A function for the evaluation of an alternative representation of a situation is required. The evaluation function gives an heuristic measure of the extent to which the models of the sub-cells support the model of the cell for a given candidate set. The function takes into account a measure of the confidence in the models of the cells involved in the situation and an estimate of the physical complexity of each cell according to its model (in terms of the number of transistors that are required for the implementation of the cell). The evaluation-function value is higher for those alternatives which:

1. have higher values for the confidence in the models involved.
2. the complexity of the cell as estimated from its model is closer to the complexity of the cell as estimated from the models of its sub-cells.

An example situation is illustrated in figure 6.1. Cell C_i has n_i sub-cells and m_i possible models. The j-th sub-cell C_{ij} has m_{ij} possible models ($1 \le j \le n_i$). A model for the j-th sub-cell C_{ij} in the situation is represented by a pair $A_{ij} = < E_{ij}, CO^*_{ij} >$. The value E_{ij} is an estimate of the confidence in the representation of the situation which represents the j-th sub-cell. The value CO^*_{ij} is a measure of the complexity of the sub-cell as estimated from its model. A pair $A_i = < e_i, CO^*_i >$ estimates the confidence in the model of cell C_i and the complexity of the cell. The value E_i which evaluates the representation of situation S_i is then calculated as

$$E_i = f(A_i, A_{i1}, \ldots, A_{in_i}) \tag{6.1}$$

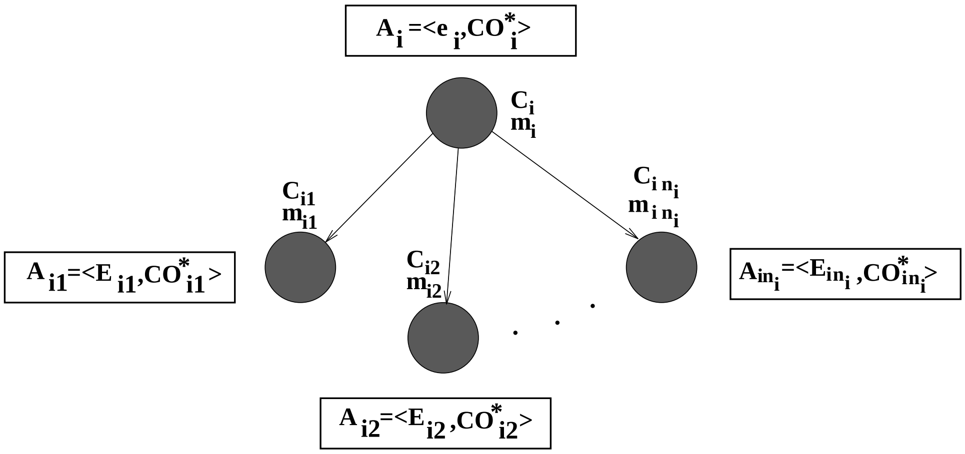


Figure 6.1: A Situation

The best alternatives for the representation of the situation are those which have models whose associated pairs optimise the value of the evaluation function f. The selection of the optimal pairs (models) can be done without having to generate all possible candidate sets. Section 6.2 introduces the basic concepts for the evaluation of alternative representations. The evaluation of the model of a cell is considered in section 6.3. The evaluation of the representation of a situation is examined in section 6.4 and section 6.5. The complexity of a cell as estimated from its model is discussed in section 6.6. The best alternatives for the representation of each situation are obtained as described in section 6.7. A solution for the whole design must be selected from the best alternatives selected for each situation. This involves selecting an alternative for each situation so that each cell has the same model in all the situations in which the cell appears. It is not possible to just consider the best ranked alternative for each situation. A cell usually appears in more than one situation and the best ranked alternatives for these situations may consider different models for this cell. It is then necessary to evaluate which set of coherent alternatives for the whole design is the most attractive. The evaluation of the different situations are combined to evaluate the representation of the overall design. The alternatives selected for the representation of each situation are analysed as described in chapter 7.

6.2 Evaluating Alternatives

The evaluation of the model of a cell and the evaluation of the set of models that represents a situation requires a mechanism to weight and to accumulate evidence (for or against the model that represents a cell and for or against the models that represent a situation). The weighting and accumulation of evidence must be made in a manner that reflects the way of reasoning of an expert. There is very little agreement among AI researchers concerning the methods that should be used for weighting and accumulating evidence. Methods based on statistics and probability theory offer well-understood ways of doing this. Indeed, if the probability values of alternative answers are known, then the best guess is the most probable one. Theoretically, probability theory may solve the problem if probability values are available [1]. These values are about long-run relative frequencies of events, and so they must be derived from empirical investigation. Unfortunately, the information required to obtain these values is not always available.

In the context of this thesis it is not clear how to calculate, for example, what is the probability of a cell being a flip-flop given the number and kinds of gates used for its implementation and the number and types of its ports. The difficulty of obtaining these numbers has caused many AI researchers to dismiss probabilistic approaches to uncertainty [CM85]. A number of alternatives to probability theory have been reported in the literature, notably certainty factors [Sho76], fuzzy logic [Zad75] and belief functions [Sha76]. Often, simplifying assumptions are also made which allow heuristic methods to take over. These heuristic methods use domain-specific knowledge to compute the impact of new evidence instead of using a domain-free calculus [Jac90].

The approach taken in this work originates from the use of certainty factors. It uses domain-specific knowledge to combine evaluation-function values of different items of knowledge that are required to support an hypothesis. As for the case of certainty factors, an evaluation-function value for a given item of knowledge must be in the range $[-1, 1]$, where -1 means definitely not, 0 means unknown and 1 means definitely yes. The method used to combine evaluation-function values is based on the method used for certainty factors. The calculation of the confidence in an hypothesis requires an heuristic measure of the relative importance of the items of knowledge with respect to the hypothesis and the evaluation-function values of these items of knowledge.

[1] Research shows that people do not appear to be probabilistic reasoners only. For example, they are apt to discount prior odds and give more weight to recently presented evidence [KT72], and they tend to be over-confident in their judgements [KST82]. The use of probability values, if available, does not necessarily result in a way of reasoning which is close to that exhibited by human experts.

On an intuitive basis, if e_k is the value for the evaluation of an hypothesis after considering k items of knowledge, and a new item of knowledge $k+1$ is provided with a value of v_{k+1}, the combination of these two values must give a new value e_{k+1} that is higher than e_k if v_{k+1} is positive and lower if v_{k+1} is negative. The difference $e_{k+1} - e_k$ can be thought of as depending on the value of v_{k+1} and on the importance of this item of knowledge with respect to the hypothesis.

The solution adopted is as follows. If e_k and v_{k+1} are both positive (negative), the value e_k rises (falls) towards 1 (-1) by an amount proportional to v_{k+1}. That is

$$\begin{array}{rcll} \frac{(1-e_k)-(1-e_{k+1})}{1-e_k} & = & r_{k+1}\, v_{k+1} & \quad e_k, v_{k+1} > 0 \\ \frac{(-1-e_k)-(-1-e_{k+1})}{-1-e_k} & = & -r_{k+1}\, v_{k+1} & \quad e_k, v_{k+1} < 0 \end{array}$$

The value r_{k+1} is referred to as a *weighting factor* for the $(k+1)$-th item of knowledge and it depends on the relative importance of this item with respect to the hypothesis. This value must be in the range $0 < r_k \leq 1$. In the case that e_k and v_{k+1} are of a different sign the result is given by

$$e_{k+1} = \frac{e_k + r_{k+1}\, v_{k+1}}{1 - min\{|e_k|, |r_{k+1}\, v_{k+1}|\}}$$

Intuitively, the sign of the result depends on which of the factors e_k or $r_{k+1}\, v_{k+1}$ has a larger absolute value. The magnitude of the result depends on the difference between these values which is dampened by the denominator.

The product

$$w_k = r_k\, v_k \tag{6.2}$$

is referred to as the *weighted contribution* for the k-th item of knowledge. Considering this, the equations above become

$$\left.\begin{array}{rcll} e_{k+1} & = & (1 - w_{k+1})\, e_k + w_{k+1} & \quad e_k, w_{k+1} > 0 \\ e_{k+1} & = & (1 + w_{k+1})\, e_k + w_{k+1} & \quad e_k, w_{k+1} < 0 \\ e_{k+1} & = & \frac{e_k + w_{k+1}}{1 - min\{|e_k|, |w_{k+1}|\}} & \quad otherwise \end{array}\right\} \tag{6.3}$$

These equations are examined in detail in appendix C. For the case $r_{k+1} = 1$, these equations correspond to the formulae used for certainty factors in systems such as Mycin [Sho76] [2]. These formulae are used to evaluate an hypothesis for which evidence is provided from separate idiosyncratic features (or symptoms in the terminology of Mycin). The product $r_k\, v_k$ can be seen as the result of a rule which says that if symptom k holds with certainty v_k, and the certainty about the hypothesis being true when the symptom k is true is r_k, the certainty of the hypothesis is $w_k = r_k\, v_k$. Then, this value updates the existing certainty for the hypothesis according to 6.3. But clearly, the probability of the hypothesis being true when a symptom is identified is not well defined

[2] An account of why certainty factors were preferred to other alternatives for representing uncertainty in the design of the Mycin system can be found in [BS84]. Probabilistic interpretations of these formulae have been attempted [Hec86], but it is generally accepted that certainty factors do not correspond to measures of absolute belief.

in the domain of this work (e.g. it is not possible to calculate the probability of a cell being a register when it has been found that the cell has a clock port). Instead of using a probabilistic value, the value r_k indicates the expected contribution of this symptom to the understanding of the whole hypothesis.

The effect of the number of values combined and their associated weighting factors is studied by means of the following case-study. The case-study considers that all the w_k are of the same sign. The result e_n of combining the evaluation-function values of n items of knowledge which support an hypothesis is calculated by means of the equations 6.3 as

$$\begin{cases} e_1 &= w_1 \\ e_2 &= (1 \pm w_2)\, e_1 + w_2 \\ &\vdots \\ e_n &= (1 \pm w_n)\, e_{n-1} + w_n \end{cases} \tag{6.4}$$

The positive sign is used for the case that $w_k < 0$ and the negative sign for the case $w_k > 0$. By substitution this gives (see appendix E.1)

$$e_n = 1 - \prod_{k=1}^{n}(1 - w_k) \qquad w_k > 0 \tag{6.5}$$

$$e_n = -1 + \prod_{k=1}^{n}(1 + w_k) \qquad w_k < 0 \tag{6.6}$$

The effect of the evaluation-function values can be examined by considering the particular case

$$w_1 = w_2 = \ldots = w_n = w$$

With this, the equations above become

$$e_n = 1 - (1 - w)^n \qquad w > 0,\ n > 1 \tag{6.7}$$

$$e_n = -1 + (1 + w)^n \qquad w < 0,\ n > 1 \tag{6.8}$$

The case $w > 0$ is pictured in figure 6.2 for different values of w and n. The function $e_n = f(n)$ is depicted with solid lines and the function $e_n = f(w)$ is depicted with dashed lines. These functions are all monotonically increasing and convex with the rate of increase always decreasing less rapidly for the second function (see proof in appendix E.2). The functions for the case $w < 0$ are symmetric to the functions for the case $w > 0$ with respect to the horizontal axis. The consequences that can be derived from figure 6.2 include:

- the first items of knowledge will find it easier to push the evaluation-function value up, but it becomes more and more difficult. Since $w \leq 1$, e_n tends to 1 when n rises and only with $w = 1$ or an infinite number of items of knowledge would be possible to reach the maximum value of 1.

- a few 'good' items of knowledge will have a higher contribution than more items of a lesser quality (see proof in appendix E.2). For example, the combination of five items of knowledge with $w = 0.1$ gives a total evaluation-function value that is lower than the combination of two items of knowledge with $w = 0.3$.

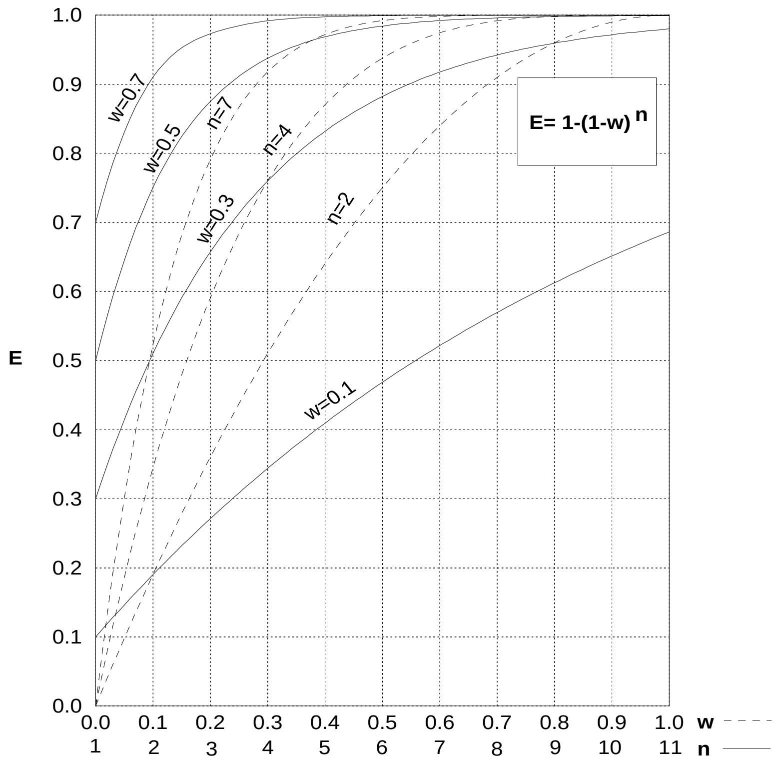


Figure 6.2: Combination of Evaluation-function Values

It is important to note that an evaluation-function value does not correspond to a measure of absolute belief. In some ways, this is not such a serious drawback because a knowledge-engineering application is seeking to represent an expert's knowledge of the domain (imperfect though it may be), rather than create a veridical model of it. It seems sensible to use a mathematically correct formalism aimed at simulating human decisions, even though it gives an indication of change of belief in place of measures of absolute belief. The main uses of these values are:

i – to provide guidance to the system in a way that simulates human reasoning. Faced with alternatives, the system uses these values to make heuristic choices that do better than chance.

ii – to compare solutions by means of the degree of confidence associated with them.

iii – to cause a model of a cell or the modelling of a situation to be deemed unpromising and pruned from the search space if its evaluation falls below a threshold value.

6.3 Evaluation of the Model of a Cell

A model for a cell emerges from the comparison of a knowledge plan with an heuristic model in the system. The comparison of a plan with a system frame involves matching the seven slots indicated in table 6.1 (see section 5.5). The evaluation-function value that measures the confidence in the model is based on the evidence provided by being able to match some of these slots. Each slot has an evaluation-function value and a weighting factor. The value of the evaluation of a slot is given by the matching algorithm. In the case of a compound slot (a slot made of several sub-slots such as the types of the cell) this value is calculated from the combination of the weighted contribution of each sub-slot by means of the equations 6.3 (each sub-slot will have a value that evaluates the matching of the sub-slot and a weighting factor which measures the relative importance of that sub-slot). The value for the evaluation of the model is given by combining the weighted contributions of each slot according to the equations 6.3. This value measures the evidence that a plan matches a system frame. Because of this, it is used as an indication of how strongly it is believed that the knowledge contained in a model represents the actual electronic cell.

Slot	**Slot Evaluation**	**Weighting Factor**	**Weighted Contribution**
Name	v_1	r_1	$w_1 = r_1\, v_1$
Types	v_2	r_2	$w_2 = r_2\, v_2$
Interface	v_3	r_3	$w_3 = r_3\, v_3$
Contents	v_4	r_4	$w_4 = r_4\, v_4$
DataFlow	v_5	r_5	$w_5 = r_5\, v_5$
Functionality	v_6	r_6	$w_6 = r_6\, v_6$
Constraints	v_7	r_7	$w_7 = r_7\, v_7$

Table 6.1: Evaluation of the Model of a Cell

The value of the weighting factor of a slot determines how much this slot can contribute to the final evaluation-function value. The effect of the weighting factors of the slots of a plan for the evaluation of the matching can be observed by considering the case

$$e_1 = e_2 = \ldots = e_7 = 1$$

This case implies that all the slots are perfectly matched. The evaluation of the model of the cell is then given by

$$e_{max} = 1 - \prod_{k=1}^{7} (1 - r_k) \tag{6.9}$$

This equation calculates the maximum value for the evaluation of a model. The value for e_{max} would be 1 in the case that any of the r_k had a value of 1. The values of the r_k are arbitrarily chosen according to the relative importance attributed to each slot and they are always smaller than 1 (see section 8.3) because they only have a partial contribution to the understanding of the cell. The evaluation of a model can never give 1 since only a partial analysis of the electronic data is carried out and a degree of uncertainty must exist about the knowledge represented.

6.4 Evaluation of a Situation

The evaluation of a situation gives an heuristic measure of the extent to which the models of the sub-cells support the model of the cell. Assuming that all the models are consistent, this measure is based on an heuristic estimation of the contribution of each sub-cell to the operation (and therefore understanding) of the cell. The evaluation of the situation defined by cell C_i with its n_i sub-cells $C_{i,j}$ $(1 \leq j \leq n_i)$ is as follows. Initially, the value for the evaluation of the situation tallies with the value e_i which corresponds to the evaluation of the model of C_i. The fact that the model of the cell is assumed to be consistent with the models of its sub-cells must result in a new value E_i for the evaluation of the whole situation. Since a sub-cell represents a part of the functionality of the cell, each sub-cell contributes its part to push the value e_i towards E_i. This contribution must depend on the extent to which this sub-cell is understood and its specific importance in the design of the cell. If the evaluation of the situation which represents sub-cell $C_{i,j}$ gave a value $E_{i,j}$, the contribution of the sub-cell is obtained by weighting this value by a factor $r_{i,j}$. This factor represents the specific importance of this sub-cell in the situation. The *weighted contribution* of the sub-cell is then given by

$$w_{i,j} = r_{i,j}\, E_{i,j} \tag{6.10}$$

In order to determine how much the contribution of each sub-cell pushes the value e_i towards E_i, the equations 6.3 can be used again. For example, for the particular case in which $w_{i,j} \geq 0$ for all j, the evaluation-function for the whole situation is given by the sequence

$$\left\{ \begin{array}{rcl} a_0 & = & e_i \\ a_1 & = & (1 - w_{i,1})\, a_0 + w_{i,1} \\ & \vdots & \\ a_{n_i} & = & (1 - w_{i,n_i})\, a_{n_i - 1} + w_{i,n_i} \end{array} \right. \tag{6.11}$$

with $E_i = a_{n_i}$. The solution of this sequence gives

$$E_i = 1 - (1 - e_i) \prod_{j=1}^{n_i} (1 - w_{i,j}) \tag{6.12}$$

The value E_i is a measure of the confidence in the model of cell C_i according to the models of its sub-cells. Considering

$$U_i = (1 - e_i) \prod_{j=1}^{n_i} (1 - w_{i,j}) \tag{6.13}$$

equation 6.12 becomes

$$E_i = 1 - U_i \tag{6.14}$$

The term U_i $(0 \leq U_i \leq 1)$ is a measure of the degree of uncertainty about the model of a cell considering the models of its sub-cells. For example, in the unattainable case that $e_i = 1$ then $U_i = 0$ and $E_i = 1$. That is, there would be no uncertainty about the model of a cell regardless of the models of its sub-cells.

The value E_i is seen as a parameter to compare solutions rather than as a parameter to indicate the degree of belief in the solution. Practically, this function is good if its value rises when the likelihood of the solution grows as assessed by an expert designer and vice versa. Then, for this measure to be effective, it must reflect the way in which human experts reason about the contents of an electronic cell.

For the understanding of a cell, a human expert would usually consider first the larger sub-cells and the smaller ones later. Larger sub-cells often perform more complex functions and therefore they contribute more to the functionality of the whole device. Generally, the more complicated the function of a device, the larger the number of transistors required for its physical implementation. Of course, the relationships between the size and functionality of a cell are not so trivial. For example, the functionality of a large memory cell is rather simple, specially if the addressing mechanisms are excluded from the cell itself, but it can easily be the largest cell of an electronic design. But for the overall reasoning process, the investigation in first place of the largest cells is often worthwhile. The reason for this is that large cells usually provide more insights about the flow of data and control information in the whole device. Therefore, the size of a cell and the size of its sub-cells can be used to weight the contribution of each sub-cell.

6.5 Weighting Factors

For the purpose of this chapter, the *complexity* of a cell is defined as the number of transistors that are required for its physical implementation. The complexity of a cell C_i can be expressed as

$$CO_i = \sum_{j=1}^{n_i} (I_{i,j}\, CO_{i,j}) \tag{6.15}$$

where CO_i is the complexity of cell C_i which contains n_i sub-cells, and $CO_{i,j}$ is the complexity of its j-th sub-cell $C_{i,j}$ which is instantiated $I_{i,j}$ times in the design of C_i. The *relative importance* of cell $C_{i,j}$ with respect to the design of cell C_i is then defined as

$$R_{i,j} = I_{i,j} \frac{CO_{i,j}}{CO_i} \tag{6.16}$$

which indicates the percentage of transistors with which sub-cell $C_{i,j}$ contributes to the design of cell C_i. Obviously

$$\sum_{j=1}^{n_i} R_{i,j} = 1 \tag{6.17}$$

and

$$R_{i,j} \leq 1 \tag{6.18}$$

The factor $R_{i,j}$ could be used for weighting the contribution of sub-cell $C_{i,j}$ to the understanding of cell C_i if it were not that the complexities of the cells in the design can only be estimated. To calculate the exact number of transistors required by each cell it is required that the hierarchy of the design has transistor cells as the primitives of the hierarchy graph. This is not always the case and, even if transistor cells are defined, they may not be recognised as such.

One way of addressing this is to consider an heuristic function that calculates an estimate of the number of transistors required by a cell from the study of its model (see section 6.6 for an example function). The estimated complexity of a cell C_i according to its model is denoted as CO_i^*. The estimated complexity of the j-th sub-cell of C_i according to its model is denoted as $CO_{i,j}^*$. Considering estimated complexities the equations 6.15 to 6.18 can be written as

$$CO_i^c = \sum_{j=1}^{n_i} I_{i,j}\, CO_{i,j}^* \tag{6.19}$$

$$R_{i,j}^* = I_{i,j}\, \frac{CO_{i,j}^*}{CO_i^c} \tag{6.20}$$

$$\sum_{j=1}^{n_i} R_{i,j}^* = 1 \tag{6.21}$$

$$R_{i,j}^* \leq 1 \tag{6.22}$$

The value CO_i^c corresponds to the complexity of C_i as calculated by means of the estimated complexities of the sub-cells. The factor $R_{i,j}^*$ could be used for weighting the contribution of sub-cell $C_{i,j}$ but it does not take into account the complexity estimated for the cell according to its model CO_i^*.

The difference $|CO_i^c - CO_i^*|$ is a parameter that indicates how much the complexity of a cell as estimated from the models of its sub-cells differs from the complexity of a cell as expected from its model. A substantial value for this parameter is an indication that the modelling of the situation may be incorrect (the excess or shortage of transistors being too large to accept that the models of the sub-cells are sensible according to the model of the cell). This should be reflected as lower values for the weighting factors of the models of the sub-cells and, consequently, as a lower value for the evaluation of the whole situation. To achieve this, the weighting factor of a sub-cell $C_{i,j}$ must be of the form

$$r_{i,j} = g(R_{i,j}^*, CO_i^c, CO_i^*) \tag{6.23}$$

This equation is derived by calculating how far the model of sub-cell $C_{i,j}$ is of being able to represent the excess or shortage of transistors. The excess or shortage of transistors is measured in terms of instances of this sub-cell. That is

$$\Delta I_{i,j} = \frac{|CO_i^c - CO_i^*|}{CO_{i,j}^*} \tag{6.24}$$

and the excess or shortage of transistors represents $\Delta I_{i,j}$ instances of sub-cell $C_{i,j}$ according to the estimated complexity of the sub-cell. The weighting factor of sub-cell $C_{i,j}$ is then defined by the ratio

$$r_{i,j} = \frac{I_{i,j}}{I_{i,j} + \Delta I_{i,j}} \tag{6.25}$$

Intuitively, this value can be illustrated by means of two different cases:

1 – *$\Delta I_{i,j}$ is high with respect to $I_{i,j}$.* In this case the value of the weighting factor will be low. Two possibilities can be considered:

- the model of sub-cell $C_{i,j}$ is wrong. If this model must account for the difference then the high value for $\Delta I_{i,j}$ indicates that the model must be quite wrong.
- the model of sub-cell $C_{i,j}$ is right. In this case, the high value for $\Delta I_{i,j}$ implies that the size of the sub-cell is rather small with respect to the complexity of the cell.

In both cases, $r_{i,j}$ must be low to indicate little contribution from this sub-cell.

2 – *$\Delta I_{i,j}$ is low with respect to $I_{i,j}$.* In this case the value of the weighting factor will be high. Again, two possibilities can be considered:

- the model of sub-cell $C_{i,j}$ is wrong. If this model must account for the difference then the low value for $\Delta I_{i,j}$ indicates that the model is not very far from being right (just slightly wrong).
- the model of sub-cell $C_{i,j}$ is right. In this case, a low value for $\Delta I_{i,j}$ implies that in terms of the complexity of this sub-cell the difference is rather small.

In both cases, $r_{i,j}$ must be high to indicate a significant contribution from this sub-cell.

To see if the value $\Delta I_{i,j}$ is low or high with respect to $I_{i,j}$, the percentage of variation in the number of instances is calculated by

$$V_{i,j} = \frac{(I_{i,j} + \Delta I_{i,j}) - I_{i,j}}{I_{i,j}} = \frac{\Delta I_{i,j}}{I_{i,j}}$$

and equation 6.25 for the weighting factor of a sub-cell can be written as

$$r_{i,j} = \frac{1}{1 + V_{i,j}} \tag{6.26}$$

This function is depicted in figure 6.3 (for $a = 1$ [3]). The weighting factor of a sub-cell decreases as the value of $V_{i,j}$ increases and it must be in the half-open interval $(0, 1]$.

6.5.1 Complexity Deviation Factor

The weighting factor of the sub-cell of a cell reflects both the relative importance of this sub-cell with respect to the design of the cell and the estimated and calculated

[3]The equation 6.26 could be made more or less severe by taking $r_{i,j} = \frac{1}{(1+V_{i,j})^a}$. As indicated in figure 6.3, if $a > 1$ the function is more severe and if $a < 1$ the function is less severe than the solution adopted in equation 6.26.

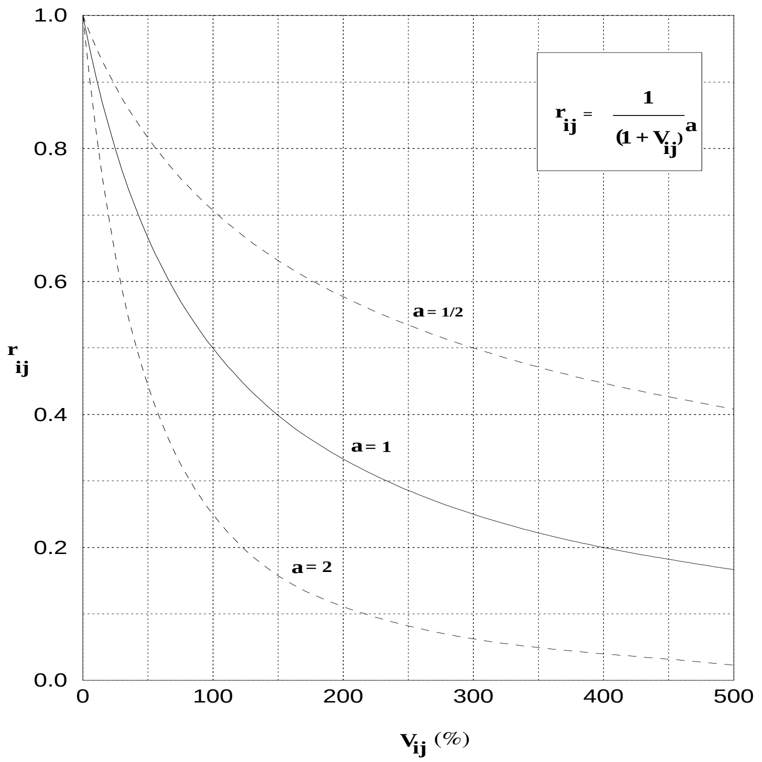


Figure 6.3: Weighting Factors

complexities of the cell. To make this explicit in equation 6.25, the *complexity deviation* factor of a cell is defined as the quotient

$$r_i = \frac{CO_i^*}{CO_i^c} \tag{6.27}$$

The factor r_i reflects how much the complexity of the cell as calculated from the expected complexities of its sub-cells deviates from the expected complexity of the cell. Since the complexity of a cell or a sub-cell is estimated from their corresponding models, this factor is an indication of how much the models of the sub-cells can support the model of the cell.

From equations 6.25 and 6.24

$$r_{i,j} = \frac{I_{i,j}}{I_{i,j} + \Delta I_{i,j}} = \frac{I_{i,j}}{I_{i,j} + \frac{|CO_i^c - CO_i^*|}{CO_{i,j}^*}}$$

and by means of 6.20 and 6.27

$$r_{i,j} = \frac{R_{i,j}^*}{R_{i,j}^* + |1 - r_i|} \tag{6.28}$$

The impact of the relative importance of a sub-cell on the value of its weighting factor is shown in figure 6.4 for different values of the complexity deviation factor (note that

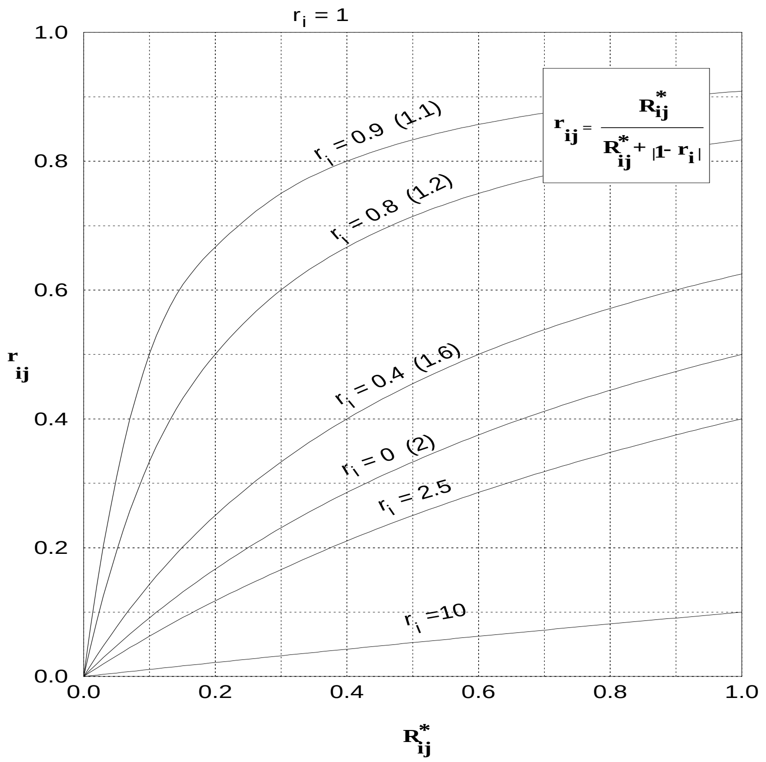


Figure 6.4: Effect of the Relative Importance

the result for $r_i = 0.9$ is the same as the result for $r_i = 1.1$). The implication of this factor is as follows:

- $r_i > 1$: the cell is *under-represented*. The complexity of the cell as estimated from its contents is lower than the complexity of the cell as estimated from its model. The fact that there might be less transistors in the contents of the cell than the number of transistors estimated from the model of the cell is severely penalised. The weighting factor of a sub-cell tends to zero when r_i increases and it increases when the relative importance of the sub-cell increases.

- $r_i = 1$: the cell is *well-represented*. The complexity of the cell as estimated from its contents is the same as the complexity estimated from its model. The models of the sub-cells are seen as totally valid to support the model of the cell and the weighting factors of all the sub-cells have the maximum value of 1 regardless of their relative importance. As a particular case, if there is only one sub-cell $C_{i,1}$ in the design of C_i then the relative importance of this sub-cell is $R^*_{i,1} = 1$. To obtain the highest weighting factor of 1 it must be $r_i = 1$ in equation 6.28. For this, it must be $CO^*_i = CO^c_i$. If there is a single instance of the sub-cell $CO^c_i = CO^*_{i,1}$ and therefore $CO^*_i = CO^*_{i,1}$. This means that the cell and the sub-cell must have the same complexity. This is clear since if there is a single sub-cell, and only one instance of it, the cell and the sub-cell must be exactly the same.

- $r_i < 1$: the cell is *over-represented*. The complexity of the cell as estimated from its contents is higher than the complexity of the cell as estimated from its model. The fact that there might be more transistors in the contents of the cell than the number of transistors expected from its model is less severely penalised than for an under-represented situation. The weighting factors tend to a value in the half-open interval $(0, 1/2]$ as r_i decreases (the limit being the case $r_i = 0$ in figure 6.4). The excess of transistors may be attributed to additional features with which the cell is provided and this cell may still preserve the essential features represented in the model.

6.5.2 A Case-Study

To quantify the impact of the deviation factor r_i of a cell C_i on the value of the weighting factors of its sub-cells and in the evaluation of the situation a case-study is considered in this section. The case-study supposes that all the sub-cells $C_{i,j}(1 \le j \le n_i)$ involved in the design of C_i have the same relative importance $R^*_{i,j}$. Therefore, from equation 6.28 all the sub-cells will have the same weighting factor $r_{i,j} = r$. From equation 6.28

$$R^*_{i,j} = |1 - r_i| \, \frac{r_{i,j}}{1 - r_{i,j}}$$

By adding together the instances of this equation that correspond to each one of the n_i sub-cells of C_i

$$\sum_{j=1}^{n_i} R^*_{i,j} = \sum_{j=1}^{n_i} \left(|1 - r_i| \, \frac{r_{i,j}}{1 - r_{i,j}} \right)$$

and considering equation 6.21

$$\sum_{j=1}^{n_i} \frac{r_{i,j}}{1 - r_{i,j}} = \frac{1}{|1 - r_i|} \tag{6.29}$$

For this case-study

$$\sum_{j=1}^{n_i} \frac{r_{i,j}}{1 - r_{i,j}} = n_i \, \frac{r}{1 - r}$$

and therefore

$$r = \frac{1}{n_i \, |1 - r_i| + 1} \tag{6.30}$$

In order to consider the impact of the weighting factors on the evaluation of the situation, this case-study supposes that every sub-cell $C_{i,j}$ has the same evaluation-function value $E_{i,j} = e > 0$ (the case for $e < 0$ can be analysed in a similar way). The evaluation of this situation can be calculated by means of equation 6.12 which gives

$$E_i = 1 - (1 - e_i)\,(1 - r\,e)^{n_i}$$

where e_i is the evaluation of the model of cell C_i. By making $e_i = 0$ to simplify things (which does not affect the conclusions of this case-study), the evaluation-function considers the contribution of the sub-cells only. This is determined by

$$\left.\begin{aligned} E_i =& \; 1-(1-r\,e)^{n_i} \\ r =& \; \tfrac{1}{n_i\,|1-r_i|+1} \end{aligned}\right\} \tag{6.31}$$

Table 6.2 illustrates this case-study for five different values of r_i. Three different cases can be considered:

- $r_i > 1$: the cell is under-represented and $r \rightarrow 0$ rapidly with the increase of r_i (see cases (a), (b) and (c) in table 6.2). The total contribution of the sub-cells E_i decreases rapidly too. The value of E_i increases slightly with the number of sub-cells n_i. This increase is more rapid for the cases that have a smaller value for e.

- $r_i = 1$: the cell is well-represented and $r = 1$ (see case (c)). If the evaluation-function values of the sub-cells is $e = 1$ the contribution of the sub-cells gives $E_i = 1$. If the value of e decreases such as $e = 0.5$, the value of E_i is smaller than 1 but tends rapidly to 1 with the number of sub-cells.

- $r_i < 1$: the cell is over-represented and $r \rightarrow \frac{1}{n_i+1}$ as r_i decreases. The value of the weighting factors decreases moderately as the value of r_i decreases, as it happens with the total contribution of the sub-cells E_i (see cases (c), (d) and (e) in table 6.2). The value of E_i increases as well with n_i, and this increase is more rapid for the cases that have a smaller value for e.

In general, the weighting factors decrease with the number of sub-cells n_i since each sub-cell will have less relative importance. However, the total contribution of the sub-cells increases slightly with n_i (a proof that the total contribution always increases with n_i is given in appendix E.3). Thus, it is slightly more significant to have more sub-cells to contribute to the understanding of a cell than having less. Operationally, this is justified by considering that it is more difficult to get the models of the sub-cells to agree with the model of the cell for those cells which have a larger number of sub-cells. The effect of n_i in E_i is more significant for lower values of the certainty of the models e. This is easily justified by considering the shape of the function $E_i = f(n_i)$ that was given in figure 6.2.

6.6 Cell Complexity Estimation Function

A simple example function for the estimation of the complexity of a cell from its model is discussed in this section. The derivation of more elaborated functions is also hinted at here but it is considered to be beyond the scope of this work. The function considered is based on the relationship between the level of abstraction of a cell and its complexity. It states that cells at higher levels of abstraction usually require a larger number of transistors than cells at lower ones. For example, a vector level cell will often require a larger number of transistors than most bit level cells and less transistors than most

a) $r_i = 10$

Cell under-represented:

e.g. $\begin{cases} CO_i^* = & 10\,000 \\ CO_i^c = & 1\,000 \end{cases}$

$\mathbf{n_i}$	r	E_i	
		e= 1	e= 0.5
1	0.1	0.1	0.05
2	0.05263	0.10249	0.05194
3	0.03571	0.10336	0.05262
100	0.00111	0.10511	0.05400

b) $r_i = 2.5$

Cell under-represented:

e.g. $\begin{cases} CO_i^* = & 1\,000 \\ CO_i^c = & 400 \end{cases}$

$\mathbf{n_i}$	r	E_i	
		e= 1	e= 0.5
1	0.4	0.4	0.2
2	0.25	0.4375	0.23438
3	0.18182	0.45229	0.24869
100	0.00662	0.48545	0.28228

c) $r_i = 1$

Cell well-represented.

$\mathbf{n_i}$	r	E_i	
		e= 1	e= 0.5
1	1	1	0.5
2	1	1	0.75
3	1	1	0.875
100	1	1	0.99999

d) $r_i = 0.4$

Cell over-represented:

e.g. $\begin{cases} CO_i^* = & 400 \\ CO_i^c = & 1\,000 \end{cases}$

$\mathbf{n_i}$	r	E_i	
		e= 1	e= 0.5
1	0.625	0.625	0.3125
2	0.45455	0.70248	0.40289
3	0.35714	0.73433	0.44575
100	0.01639	0.80851	0.56091

e) $r_i = 0.1$

Cell over-represented:

e.g. $\begin{cases} CO_i^* = & 1\,000 \\ CO_i^c = & 10\,000 \end{cases}$

$\mathbf{n_i}$	r	E_i	
		e= 1	e= 0.5
1	0.52632	0.52632	0.26316
2	0.35714	0.58673	0.32526
3	0.27027	0.61141	0.35309
100	0.01099	0.66878	0.42361

Table 6.2: Effect of the Deviation Factor

processor level cells. However, there are no clear boundaries to the range of transistors that can be used to implement, for example, a vector level storage cell such as a register. If only the level of abstraction is taken into account, the only possibility to consider is the typical number (or range) of transistors required to implement a cell of a given level.

The *typical complexity* of a cell at a given level of abstraction can theoretically be defined as the average value of the number of transistors that are used in all the electronic cells that belong to that level. For obvious reasons, this value cannot be calculated. Instead, an heuristic value is considered from the observation of a sample of cells at each level. For example, the typical complexity of a cell at the gate level may be considered to be around 10 transistors (of course, the typical complexity of a cell at the transistor level must be 1).

The *typical interlevel complexity ratio* is defined as the quotient between the typical complexity of a level of abstraction and the typical complexity of its immediate lower level predecessor. For the example above, the typical interlevel complexity ratio between the gate level and the transistor level is 10/1. In order to simplify things, this ratio is taken as a constant value HR for all levels of abstraction. With this assumption (which is justified below) the typical complexity TC_i of a cell C_i at the level of abstraction EH_i is given by

$$TC_i = HR^{EH_i} \tag{6.32}$$

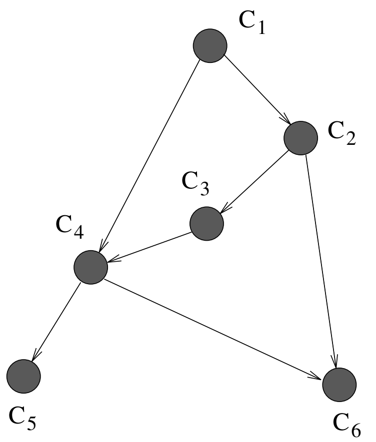
where the levels of abstraction rank from 0 for the transistor level to 5 for the system level. As an example, an acceptable value for HR is 10. Thus, a gate typically contains 10 transistors, a bit level cell typically contains 10 gates or 10^2 transistors, a vector level cell typically contains 10 bit level cells or 10^3 transistors and so on. This equation can be used for the calculation of the estimated complexity of a cell so that

$$CO_i^* = TC_i \tag{6.33}$$

Equation 6.32 is a rough approximation. For example, common registers contain 4, 8, 16, 32 or 64 bits and the typical number of 10 bit units per vector level cell seems rather arbitrary. A more complicated function for the estimation of the complexity of the cell, at vector level at least, should take into account a factor proportional to the number of data input signals (or data output signals) of the cell, since this number usually reflects the amount of bit units used. Despite the lack of a more complex function, two main considerations make the use of equation 6.32 worthwhile. Firstly, the factor r_i allows for a moderate variation around the typical complexity before penalising hard the weighting factors. Considering the example above, the typical range of bit cells in a register is $[4 - 64]$. The estimation of the typical value of 10 bit units per vector level cell gives a range for the deviation factor r_i of $[10/64 - 10/4]$, that is, $[0.16 - 2.5]$. As shown in table 6.2, for this range of values of r_i the weighting factors still allow a significant contribution. Secondly, these factors are aimed at weighting the contribution of the sub-cells so that the different alternatives can be ranked. Since the complexities of cell and sub-cells are all estimated by means of the same equation, the alternative representations of a cell must still be ranked in a sensible way and the use of equation 6.32 may allow the rejection of the most unpromising alternatives.

6.7 Selection of Alternatives

Given a set of models for each cell of the design, the evaluation function 6.12 is used to select the best candidate solution set for the representation of a situation and for the representation of the overall design. In the case $i = 1$, the function 6.12 calculates the value E_1 which evaluates the confidence in the representation of the top situation. Since the evaluation of a situation takes into account the evaluation of its sub-cell situations (see equation 6.10), the value E_1 evaluates the overall design. The candidate solution set which results in the highest value for E_1 must be considered first for model-based reasoning. This set must not violate constraints between cell models which were already calculated from the processing of previous sets as discussed in the next chapter (this guarantees that a failed set is not considered twice). The example design of figure 6.5 is used to illustrate the way in which the best candidate set is selected. Ms_i indicates the set of models available for the i-th cell of the design. For example, two models M_1^1 and M_1^2 are available for the top cell of the design C_1. For the selection of the best candidate set, it is not necessary to evaluate all possible combinations of models (24 combinations are possible for the simple example of figure 6.5 according to equation 3.2). The evaluation function requires for each model of a cell the confidence in the model and the estimation of its complexity. The situations of the design are first separately evaluated.



Set of Best Cell Models:

$Ms_1 = \{M_1^1, M_1^2\}$
$Ms_2 = \{M_2^1, M_2^2, M_2^3\}$
$Ms_3 = \{M_3^1\}$
$Ms_4 = \{M_4^1, M_4^2\}$
$Ms_5 = \{M_5^1\}$
$Ms_6 = \{M_6^1, M_6^2\}$

Ranking of Candidate Situation Sets:

$As_1 = \{M_1^1\text{–}M_2^1\text{–}M_4^1, M_1^1\text{–}M_2^3\text{–}M_4^2, \ldots\}$
$As_2 = \{M_2^3\text{–}M_3^1\text{–}M_6^2, \ldots\}$
$As_3 = \{M_3^1\text{–}M_4^2, \ldots\}$
$As_4 = \{M_4^2\text{–}M_5^1\text{–}M_6^2, \ldots\}$
$As_5 = \{M_5^1\}$
$As_6 = \{M_6^1, M_6^2\}$

Figure 6.5: An Example Design

For the evaluation of a situation independently of the other situations, the confidence e_{ij} in each model of sub-cell C_{ij} is considered in equation 6.12 instead of E_{ij}. The models Ms_i for cell C_i are sorted according to their confidence value. The first model for each cell (the model with the highest confidence value) is considered to form an initial combination of models for a situation and to get an initial value from the evaluation function for the situation. The selection of the best candidate situation set proceeds by considering what model changes in this initial combination can result in a higher value of the evaluation function. For example, after the initial combination, the evaluation function can only result in a higher value if the new models considered result in a value for r_i in equation 6.28 closer to 1. By using the confidence value and complexity of the models, it is possible to drive the search in the space of possible combinations of models and to avoid the combinatorial nature of the problem. In figure 6.5, some example candidate sets for the i-th situation are ranked in a set As_i. For instance, M_1^1–M_2^1–M_4^1 is the best ranked set of models to represent situation S_1. The situation candidate set selected must not violate constraints between cell models calculated from the processing of previous sets (a failed situation set cannot then be considered twice). Otherwise, another situation set must be selected.

A solution for the whole design is obtained by selecting an alternative for each situation in such a way that each cell has the same model in all the situations in which the cell appears. A cell usually appears in more than one situation and the best ranked alternatives for these situations may consider different models for the cell. For example, the best ranked alternative for situation S_1 uses a different model for cell C_2 than the best ranked alternative for situation S_2. In the case that a conflict occurs when considering the best ranked alternatives, the different possibilities are evaluated according to equation 6.12 to determine the most convenient overall set. Each selected situation set is analysed by model-based reasoning as discussed in the next chapter.

Chapter 7

Model-based Reasoning

Model-based reasoning about a design situation implies the derivation of relationships which link the items of knowledge used for the representation of its separate objects. Relationships are derived, for example, from the study of the hierarchy of the situation, from the analysis of the connectivity between objects and from inferences about the flow of signals. The derivation of these relationships demonstrates a deep understanding of the situation. The reasoning for a valid set of cell models (solution set for the situation) may result in the propagation of knowledge for describing objects which are not fully represented and in the reorganisation of objects to form a more meaningful structure. The addition of knowledge to incomplete cell models may lead to the formation of new knowledge plans which can be used in another knowledge-generation/knowledge-propagation cycle for the generation of more refined solution sets. Model-based reasoning may be assisted by functions aimed at the recognition of implementation patterns and problem-solving strategies.

7.1 Problem Definition

This chapter focuses on reasoning about the knowledge which represents each individual situation in a design. The initial knowledge that represents the situation S_i includes one heuristic model for cell C_i and one heuristic model for each one of its n_i sub-cells. The model of a cell or a sub-cell may be complete or incomplete. A sub-cell may be instantiated several times in the design of cell C_i and the model that represents the sub-cell equally applies to each one of the specific instances. In this context, reasoning about situation S_i implies the derivation of relationships for linking the knowledge which forms the $n_i + 1$ cell models involved and knowledge for the representation of the signals carried by the interconnections. The aims of the reasoning about a situation include the validation of the knowledge represented (the representation of the situation is valid if items of knowledge are interrelated without violating system constraints), the derivation of further knowledge plans for the cells involved and the reorganisation of objects to produce a more meaningful description of the situation.

The operative working of model-based reasoning for a situation is discussed in section 7.2. Section 7.3 reviews the scope of the understanding that can be reached with this reasoning. Relationships derived from the study of the hierarchy of the situation are used as an example. The derivation of relationships from the analysis of the connectivity between objects in the situation is presented in section 7.4. Section 7.5 discusses the derivation of relationships from a study of the flow of signals in the interior of the cell. Model-based reasoning may be eased by means of knowledge-extraction and knowledge-generation functions aimed at the recognition of stereotypical implementation patterns and problem-solving strategies as examined in section 7.6. These strategies are represented by heuristic models which describe typical uses and applications of electronic cells. The ways of reasoning described in section 7.5 and section 7.6 can result in the reformulation of the situation as examined in section 7.7.

7.2 Consistency and Knowledge-propagation

In its broadest sense, model-based reasoning for a situation ought to be able to derive that the operation of a cell (as described by its model) can be performed by means of the operation of its sub-cells (as described by their corresponding models) and the interconnections within the cell. The knowledge contained in these models represents a level of abstraction which is beyond the level required for describing a complete theory of the behaviour of the cells in question. For this reason, it is not possible to categorically state that the operation of a cell (as described for example by its electronic functionality) can be achieved according to the model of its contents. A degree of uncertainty (not necessarily founded on the confidence in the knowledge represented) must then prevail. Operationally, the intention is to reason about the knowledge which represents a situation with a double aim:

1. *consistency-checking*: the knowledge that represents an object in the situation must be consistent with the existing knowledge about the other objects in the situation. This is used as a mechanism for strengthening conclusions about the knowledge represented.

2. *knowledge-propagation*: knowledge about an object is derived from knowledge available in other objects by means of relationships established between them.

Consistency-checking and knowledge-propagation (k-propagation) are closely related. On one hand, if the items of knowledge interrelated are defined (e.g. the models of the cells interrelated are complete) and inconsistencies arise, the representation of the situation is incorrect according to the system (since it is the system that derives the possible relationships) and another set of models must be selected. On the other hand, if some of the items of knowledge required to represent an object are undefined (e.g. some cell models are incomplete) the need of consistency may result in the passing of knowledge between interrelated objects. For this reason, the functions involved in model-based reasoning are called k-propagation functions in section 4.3. As introduced there, there are two types of relationships:

1. *constraints*: a constraint is a relationship which must be satisfied by the items of knowledge which are linked. A failure to satisfy a constraint implies that the knowledge represented is incorrect according to the system. For example, sequential

sub-cells cannot exist in the design of a combinational cell. If some items of knowledge are undefined, a constraint may be used to pass knowledge between objects in a way that keeps consistency.

2. *plausible relationships*: a plausible relationship is a typical association between items of knowledge, but it may not apply to all cases. For example, the functionalities attributed to a number of interconnected ports and the signals transported are usually related. Inconsistencies cannot arise as a result of a plausible relationship but it may result in the addition of further knowledge to cells and signals which are not fully represented.

The confidence in the knowledge propagated depends on the confidence in the interrelated items which are defined before the propagation of knowledge. In the case of plausible relationships, the final confidence in a propagated item of knowledge must be decreased by a factor which measures the plausibility of the relationship used.

Figure 7.1 describes the way in which consistency-checking and knowledge-propagation operate. A k-propagation function takes the situation knowledge, which includes the models of the cells (model M_i for cell C_i and model M_{ij} for its j-th sub-cell, $1 \leq j \leq n_i$) and the description of the interconnections between objects, and derives sets of interrelated items of knowledge. Each set of interrelated items must satisfy a number of constraints provided by the function. If the constraints are satisfied, the models of the cells form a *solution set* for the situation (if this is confirmed by the rest of the k-propagation functions). Knowledge-propagation in a set of interrelated items may take place if some items are undefined: a *resolver* can derive values for the undefined items or it can elaborate new constraints about these values by considering the values contained in the defined items of the set, constraints on undefined items of the set already placed in the models of the cells, and system constraints placed by the k-propagation function.

The system constraints which are considered by the k-propagation functions are usually too weak to form a detailed plan for an undefined item (in the sense that they can not result in just one value for the item. See, for example, the system constraints discussed in section 4.5). For this reason, k-propagation functions are allowed the use of system plausible relationships in order to obtain more detailed plans for undefined items. The plans for the separate items of a model of a cell are combined to form knowledge plans for the cell as shown in figure 7.1 and discussed in section 4.6.

Model-based reasoning is applied to each one of the separate situations which form the design. A set of heuristic cell models which correctly represent the cells of a situation (as assessed by the k-propagation functions) form a solution set for the situation. The addition of knowledge to incomplete cell models may lead to the formation of new (different) knowledge plans for the corresponding cells which can be used in a new knowledge-generation/knowledge-propagation cycle: new models for some cells could be generated, a new set of models for the overall design could be selected and the design situations would again be independently analysed. If the models in the situation set analysed violate some system constraints, these constraints are stored and considered for the selection of new sets. This guarantees that a failed set for a situation is not considered twice and it helps in discarding combinations for the selection of models.

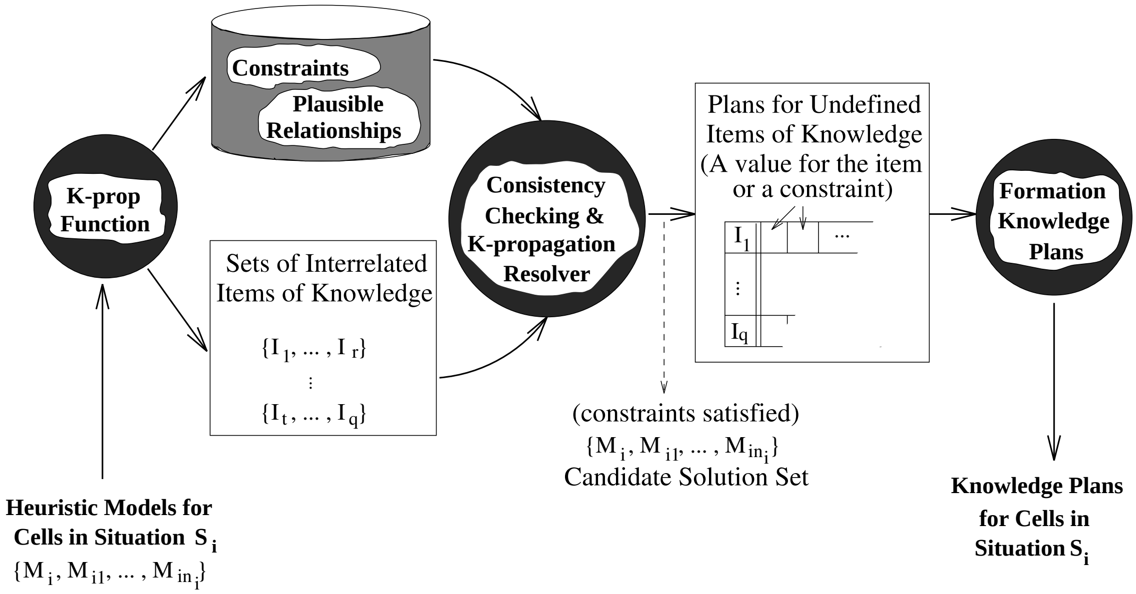


Figure 7.1: Consistency and Knowledge-propagation

7.3 Specification-level Understanding

The derivation of relationships between objects (i.e. cells, ports and signals) indicates that a deep understanding of the situation (and the design as the composition of its situations) can take place. As discussed in section 1.5, the understanding is limited to the specification-level: generic interpretations are associated with the cells and signals of a design which reveal information about their meaning, intended function or generic purpose. The interrelation of the generic purpose of the design objects by means of k-propagation functions demonstrates an overall understanding of the situation. K-extraction and k-generation functions are focussed on individual objects (recognition-targeted reasoning) and very limited understanding of the relationships between objects takes place (signals are not yet considered). Model-based reasoning moves towards the understanding of the specification since relationships between the knowledge that represents separate objects are considered.

As an example, the knowledge-propagation functions already introduced in section 4.5 illustrate the operative working of model-based reasoning. These functions interrelate the level of abstraction, logic type and data flow type in the model of a cell with the corresponding types in the models of its sub-cells. The sets of levels of abstraction, logic types and data flow types of the cells in the situation are examples of the sets of interrelated items of knowledge shown in figure 7.1. Tables 4.3 and 4.4, for instance, represent constraints on the set of logic types and data flow types, respectively. The resolver for consistency-checking and k-propagation operates as follows: given the value or a set of possible values (i.e. a constraint on the actual value) for one of these types for each sub-cell of the cell, the value (or a set of possible values) for the same type of the cell is derived. Two cases can be considered:

1. a value for the type of the cell already exists. Two sub-cases may occur:
   (a) the existing value is the same as the derived value or it belongs to the derived set (consistency-checking).
   (b) otherwise, the candidate set for the situation is inconsistent.
2. the type of the cell is undefined. The candidate set is valid if it is possible to resolve a value for the type, or a constraint on the value for the type, from the value or constraints derived and constraints which may already exist in the model with respect to the type (knowledge-propagation). Three sub-cases may occur:
   (a) no value or constraint is possible: the candidate set is inconsistent.
   (b) a single value is possible: the type of the cell is instantiated.
   (c) several values are still possible: a new constraint is placed in the cell model (possibly a refinement of existing constraints which must be removed from the cell model).

A new plan for the types of a cell can be formed in this way. The types of a cell are also related to each other according to table 3.1, and they relate to the electronic functionality of the cell as indicated in table 3.2. These tables are further examples of knowledge-propagation functions. The way of reasoning illustrated above is used again for these tables. As a result, for this example, the planning of knowledge for a type of a cell may trigger further knowledge about the rest of its types and its electronic functionality. The knowledge added can be given as the instantiation of some undefined values in the model or in the form of new and more refined constraints. As another example, hierarchical relationships between a cell and its sub-cells are also given in the contents section of the heuristic model of the cell. These relationships are plausible since it is not possible to contemplate all possible ways of constructing a cell.

Elaborate knowledge-propagation functions are described in section 7.4 and section 7.5. These functions take into account the generic purposes of the design entities and they allow an overall understanding of the situation to be gained. A discussion of the ability of the system to understand the specification of a design in relation to the central grounds of computer understanding mentioned in section 1.4 is given in section 9.1.

7.4 Example 1 — Connectivity and Knowledge-propagation

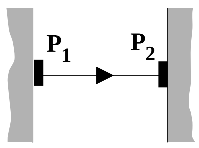
The connectivity between nodes (cell and instantiated sub-cells) in a situation of the design hierarchy establishes relationships which interrelate items of knowledge contained in the interface section of the cell models and items of knowledge which represent the signals carried by the interconnections. This section discusses the heuristics which support k-propagation functions based on connectivity. These functions derive sets of interrelated items of knowledge according to the connectivity in the situation upon which constraints and plausible relationships are applied. Knowledge plans for the interface section of incomplete cell models (i.e. models which have ports with undefined generic or electronic functionalities) can be formed. For a cell C_i which is instantiated I_i times in a design there are in general $I_i + 1$ different scenarios for the analysis of connectivity: one for the connections between the cell and its contents and one for each instance of this cell within the design of another cell (in the case of a primitive cell of the design the number of scenarios is I_i). As a result, the knowledge representing the interface section of the model for cell C_i must be consistent with knowledge representing other models and signals in $I_i + 1$ scenarios. In the case that the interface section of the model is incomplete, there are $I_i + 1$ scenarios for the formation of plans for the interfaces of the cell.

7.4.1 Plausible and Implausible Relationships

A connection between two ports of different objects is illustrated in figure 7.2(a). The connection interrelates the knowledge which represents these ports in the corresponding models and the knowledge which is used for the representation of the signal transferred. As shown in figure 7.2(b), the set of interrelated items of knowledge $\{(\mathbf{PG_1}, \mathbf{PE_1}), (\mathbf{SG}, \mathbf{SE}), (\mathbf{PG_2}, \mathbf{PE_2})\}$ includes the generic and electronic functionalities of port $\mathbf{P_1}$ of node $\mathbf{n_1}$ ($\mathbf{PG_1}$ and $\mathbf{PE_1}$, respectively), the generic and electronic functionalities of the signal that flows between ports (**SG** and **SE**, respectively) and the generic and electronic functionalities of port $\mathbf{P_2}$ of node $\mathbf{n_2}$. For convenience, a set of interrelated items of knowledge which represent two interconnected ports and the signal transferred is called in this section a *primary set*. A relationship between the items in a primary set is derived as follows:

1. *constraints* of electronic design are first applied to check for consistency between defined items in the set (for example a clock signal must not drive a select port).

2. if some items in the set are undefined, *constraints* and *plausible relationships* are applied in order to propagate knowledge to these items in a way that keeps consistency.



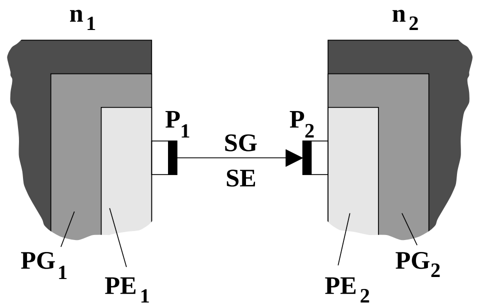


Figure 7.2: Primary Sets: (a) a 1-to-1 connection, and (b) items of knowledge for a primary set.

The use of plausible relationships is justified as follows: given a primary set with undefined items, and considering that the range of possible values for the generic and electronic functionalities of ports and signals are pre-defined, it is possible to consider all combinations of values for the undefined items which satisfy the constraints. The consideration of all combinations of values for the undefined items of the interface of a cell leads to a number of interface plans which grows exponentially with the number of ports and signals undefined. Alternatively, constraints (usually weak) for the generic and electronic functionalities of interrelated objects can be calculated and a single plan generated. However, plans for the interfaces of a cell with weak constraints and few fully represented ports are usually difficult to match (see section 5.5.2). More detailed plans for the interface section of a model are always easier to match. These plans can be formed by considering the propagation of knowledge by means of plausible relationships while discarding implausible ones.

Plausible relationships between the generic functionalities of ports and signals interrelated in primary sets depend on the type of connection as follows:

1. *short*: in this connection, the origin and destination ports of a signal are located in the cell (as opposed to its contents) as shown in figure 7.3(a). A signal is just transported from the input ports of the cell to the output ports and it does not affect the functionality of the cell. For this reason, this signal can only be seen as being of a generic *data* type from the point of view of the cell. The best candidates to be shorted are data input ports and data output ports: short signals between these ports are plausible.

2. *boundary*: a boundary connection has either the origin port in the cell and the destination port in a sub-cell instance or vice versa as shown in figure 7.3(b). Signals which flow from data (control) input ports of the cell to data (control) input ports of a sub-cell instance clearly are plausible: *data* (*control*) signals for the whole cell are passed to its sub-parts. Similarly, signals which flow from data (control) output ports of a sub-cell instance to data (control) output ports of the cell are plausible: *data* (*condition*) signals of the sub-parts of a cell are passed to the cell. The rest of the possible boundary signals are implausible. A signal from a data (control) input port of the cell to a control (data) input port of a sub-cell instance has as type *control*. The reason for this is that being considered control for either the cell or a sub-part (since a control port is connected), it should be considered control from the point of view of the design of the cell (a similar reasoning applies to the implausible boundary *condition* signals shown in figure 7.3(b)) [1].

3. *internal*: an internal connection has both the origin and destination ports located on sub-cell instances as shown in figure 7.3(c). Signals between data ports are clearly possible. A signal from a control output port of a sub-cell instance to a control input port of another sub-cell instance is also clearly plausible, but in this case the

[1] Examples of these implausible signals are found in connections from bit level cells to gate level sub-cells. For example, in the design of the flip-flop cell of figure 3.2(b) there is a boundary signal from the control input (clock) port of the cell to data input ports of two of its sub-cells (nand gates). The signal (clock) obviously is of a *control* type (though it does not control the operation of the nand sub-cells which take the signal as data).

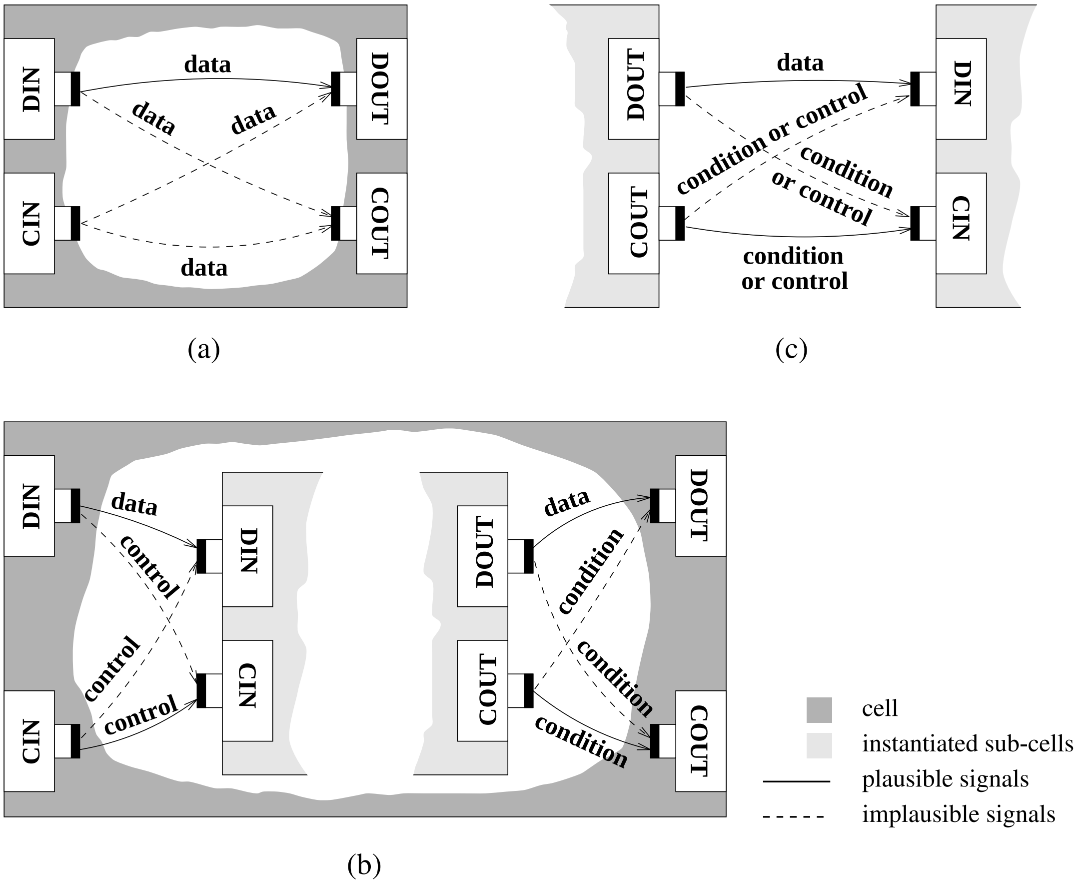


Figure 7.3: Plausible Signals: (a) short signals, (b) boundary signals, and (c) internal signals.

signal type can be either *control* or *condition*. The decision about the type of signal depends on the generic functionalities of the interconnected sub-cell instances as discussed in section 7.5. The rest of the possible internal signals are implausible and they have a *condition* or *control* type since control ports are involved (a decision about the type depends also on the generic functionalities of the sub-cells) [2].

The consideration of these plausible relationships for the objects in a primary set allows planning for undefined generic functionality values. A value for the generic functionality of a port or a signal narrows the range of possible values for its electronic functionality (see table 3.3). This range of values can be further narrowed by considering the propagation of electronic functionality values in a primary set. The propagation of these values can be done, for example, by means of the rules of table 7.1. These rules guarantee consistency between the values which are planned for the electronic functionalities of interrelated ports and signals (the results obtained always satisfy system constraints). These rules

[2]The internal signals in the example design of figure 7.9 in section 7.6.2 are implausible.

SE	**PE**$_1$	**PE**$_2$	**Action**
X	X	X	Nothing
X	X	C	Copy value C to **SE** and **PE**$_1$
X	B	X	Copy value B to **SE** and **PE**$_2$
X	B	C	If B and C consistent then copy B to **SE**
A	X	X	Copy value A to **PE**$_1$ and **PE**$_2$
A	X	C	If A and C consistent then copy A to **PE**$_1$
A	B	X	If A and B consistent then copy A to **PE**$_2$
A	B	C	The values A, B and C are consistent

(X means undefined)

Table 7.1: Propagation of Electronic Functionalities in a Primary Set

are based on plausible relationships discussed in section 3.4. The copying (propagation) of an electronic functionality value for representing a port can be done if the generic functionality of the port permits this electronic functionality (according to table 3.3). Conversely, the propagation of an electronic functionality value to represent a port also defines its generic functionality.

Items of knowledge of distinct primary sets can also be related between them to form further sets of interrelated items in two main ways:

1. by means of connections which involve more than two ports (many-to-many connections). A connection that involves one origin port and two destination ports is shown in figure 7.4(a). Two primary sets are derived from this connection. Consistency in each set, for example $\{(\mathbf{PG_1}, \mathbf{PE_1}), (\mathbf{SG}, \mathbf{SE}), (\mathbf{PG_3}, \mathbf{PE_3})\}$, is treated as discussed above. In addition, in a many-to-many connection it is possible to expect that all the origin ports have similar functionalities between them and that all the destination ports have similar functionalities too. For example, it is sensible to plan $\mathbf{PE_2}$ and $\mathbf{PE_3}$ as having the same values since $\mathbf{P_2}$ and $\mathbf{P_3}$ get the same signal from port $\mathbf{P_1}$ (e.g. if the value of $\mathbf{PE_2}$ is supposed to be *clock*, it is likely that the value of $\mathbf{PE_3}$ will be *clock* too). A set $\{(\mathbf{PG_2}, \mathbf{PE_2}), (\mathbf{PG_3}, \mathbf{PE_3})\}$ is established. Given, for example, that the value $\mathbf{PE_3}$ of port $\mathbf{P_3}$ is undefined, it is possible to plan a value for the electronic functionality of this port by propagation of the value $\mathbf{PE_2}$ [3]. Considering that the plausible relationships imposed on these sets apply more frequently than those imposed on primary sets, k-propagation for a connection with n origin ports $\{\mathbf{O_1}, \ldots, \mathbf{O_n}\}$ and m destination ports $\{\mathbf{D_1}, \ldots, \mathbf{D_m}\}$ follows the rules indicated in table 7.2.

[3] This already occurs in the case that $\mathbf{PE_1}$ is undefined by propagation in primary sets: the value $\mathbf{PE_2}$ can be copied to port $\mathbf{P_1}$ and from port $\mathbf{P_1}$ it can be copied to port $\mathbf{P_3}$. However, if $\mathbf{PE_1}$ is already defined with a value which is different to $\mathbf{PE_2}$, the value $\mathbf{PE_2}$ cannot be copied to port $\mathbf{P_3}$ by considering primary sets.

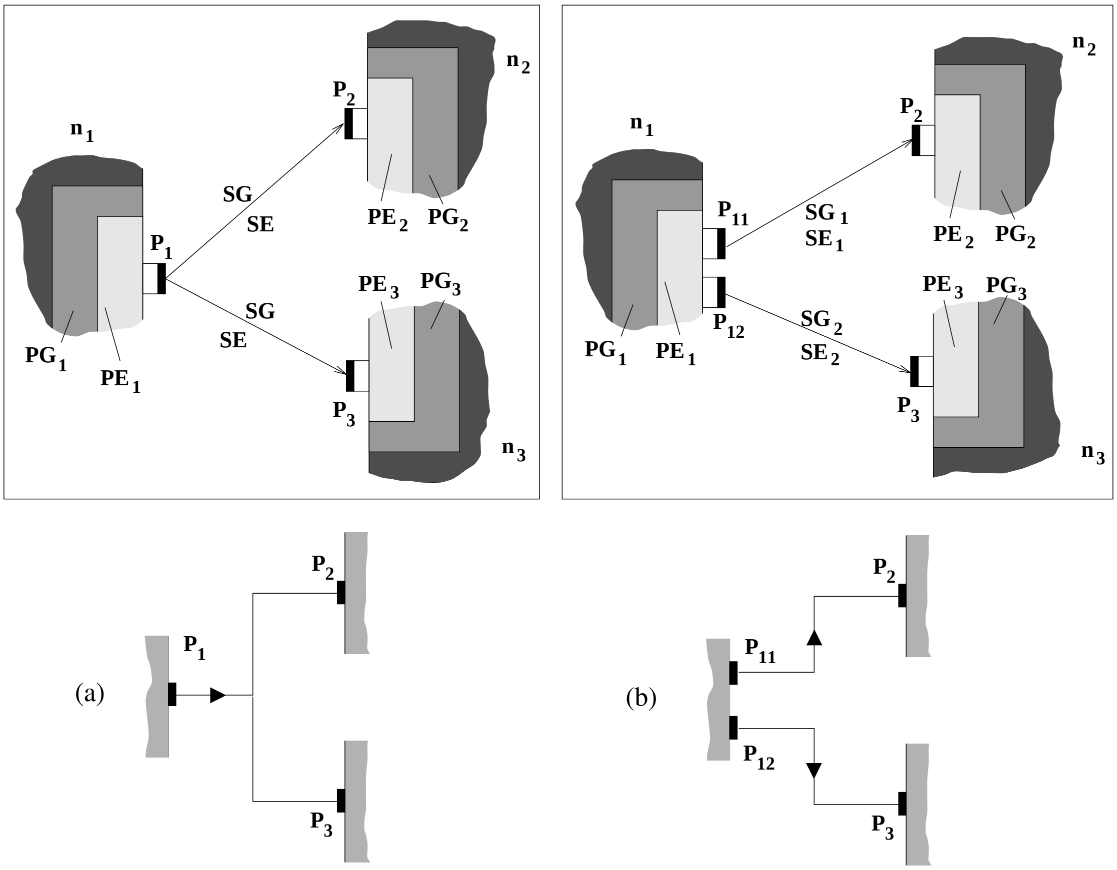


Figure 7.4: Interrelated Primary Sets: (a) a 1-to-2 connection, and (b) two 1-to-1 interrelated connections.

2. by means of relationships imposed by a cell model on its ports. This is illustrated by means of the connections in figure 7.4(b). The model of node $\mathbf{n_1}$ imposes that ports $\mathbf{P_{11}}$ and $\mathbf{P_{12}}$ have both $\mathbf{PG_1}$ as generic functionality and $\mathbf{PE_1}$ as electronic functionality. From this, it is possible to derive the set of interrelated items $\{(\mathbf{PG_2}, \mathbf{PE_2}), (\mathbf{PG_3}, \mathbf{PE_3})\}$. As a result, if the values for the functionalities of, for example, port $\mathbf{P_3}$ are undefined a possible plan (in addition to the propagation of values in the primary set of the port) includes propagating the values of the functionalities of $\mathbf{P_2}$ to $\mathbf{P_3}$ (a set $\{(\mathbf{SG_1}, \mathbf{SE_1}), (\mathbf{SG_2}, \mathbf{SE_2})\}$ can be derived in a similar way).

7.4.2 Planning Groupings of Connections

Groupings of ports are often essential to make the matching of knowledge plans feasible as discussed in section 5.5.2. Ports are grouped if they have the same functionality values. The propagation of knowledge based on connectivity discussed above may allow the planning of the functionalities of ports which are not yet fully represented and, as a consequence, the grouping of ports in a knowledge plan. In addition, the possibility

Step	Action
1	apply constraints to check consistency in each primary set
2	copy to each origin port $\mathbf{O_i}$ $(1 \leq i \leq n)$ with an undefined electronic functionality the value entrusted with the highest confidence in the set of defined electronic functionalities for the origin ports (if any)
3	copy to each destination port $\mathbf{D_j}$ $(1 \leq j \leq m)$ with an undefined electronic functionality the value entrusted with the highest confidence in the set of defined electronic functionalities for the destination ports (if any)
4	apply propagation of knowledge in each primary set
5	repeat steps 2 to 4

Table 7.2: Propagation in Many-to-Many Connections

of grouping connections (signals) according to different heuristics may result in further knowledge propagation and grouping of ports. An example of these heuristics is illustrated by means of figure 7.5. In figure 7.5(a), signals (and their corresponding ports) are grouped according to the location of their origin and destination ports. The heuristic which supports these groupings states that signals which depart from and arrive at the same nodes usually have the same functionalities. This is typical in logic electronic systems for which a high level value is coded by means of an array of binary signals (such as address and data values). A general example of this is shown in the typical circuit structure of figure 7.5(b). For each cell, the input control signal and the input data signals come from different cells and the data output signals go to the same cell (examples of circuits with this structure are shown in figure 2.7(c) and figure 2.8).

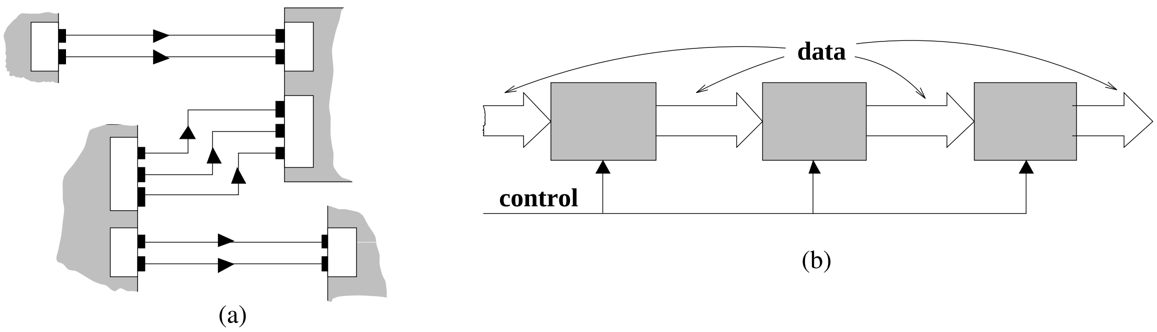


Figure 7.5: Grouping of Connections: (a) groupings according to connectivity, and (b) typical arrangement of data and control signals.

The knowledge-propagation functions which implement these heuristics can result in sensible groupings for the ports involved since signals and ports are often interrelated. These functions require that the connections collected have signals, origin ports and destination ports which have, respectively, compatible functionality values (the values are either the same or undefined). Knowledge propagation takes place in order to plan for interrelated undefined values. The heuristics used must necessarily include restrictions about the location of origin and destination ports in order to consider sensible groupings of connections.

The analysis of connectivity also offers new mechanisms for the extraction of knowledge plans for the interfaces of the cells. Groupings of ports are planned in section 4.4.2 by means of an analysis of their names. For very poorly described designs, the names of the objects may all be meaningless and groupings will not be formed if they are not specified in the description. Consequently, the matching with heuristic models may not be feasible even for cells with a moderate number of interfaces. Alternatively, plans for the interfaces of the cells which can be managed by the system can be generated by knowledge-extraction functions which purely deal with connectivity. These functions are the equivalent of the knowledge-propagation functions discussed above considering that the functionalities of all ports and signals are fully undefined. For this case, only the requirements for the location of origin and destination ports apply (such as grouping signals whose origin ports and destination ports respectively belong to the same nodes).

7.5 Example 2 — Data/Control Signal Flow

The relationships which are derived by means of the study of the connectivity in a situation are only established between adjacent objects, and they only interrelate knowledge about ports and signals. The study of the flow of data and control signals interrelates items of knowledge which correspond to nodes that are not directly connected. These items include, in addition to knowledge about signals and the interfaces of the nodes, knowledge about the types of each node (purpose and data flow types) and knowledge about data/control signal flow within each node. The interrelation of the generic purposes of the design objects allows an overall specification-level understanding of the situation to be gained. As an example, the design of figure 2.8 is represented in figure 7.6 in terms of the generic functionalities of nodes and signals. The heuristics which support the interrelation of these items are based on the generic functionalities of the nodes as follows:

1. *control*: a control (CTR) node sends signals to control a set of interrelated nodes or a data path (e.g. pipelined operator and storage nodes). The control node takes condition signals and, possibly, data signals (both through data input ports) to make data-dependent control and sequencing decisions. Condition signals come from nodes in the data-paths under control. A control node may take control signals (usually boundary signals) through control input ports and it always issues control signals through the control output ports. It is also possible to issue data signals through data output ports (e.g. constant values contained in the control node). A control node can store and generate control and data information.

2. *storage*: a storage (STO) node can store either data or control information (in the second case the information stored produces, possibly after decoding, control

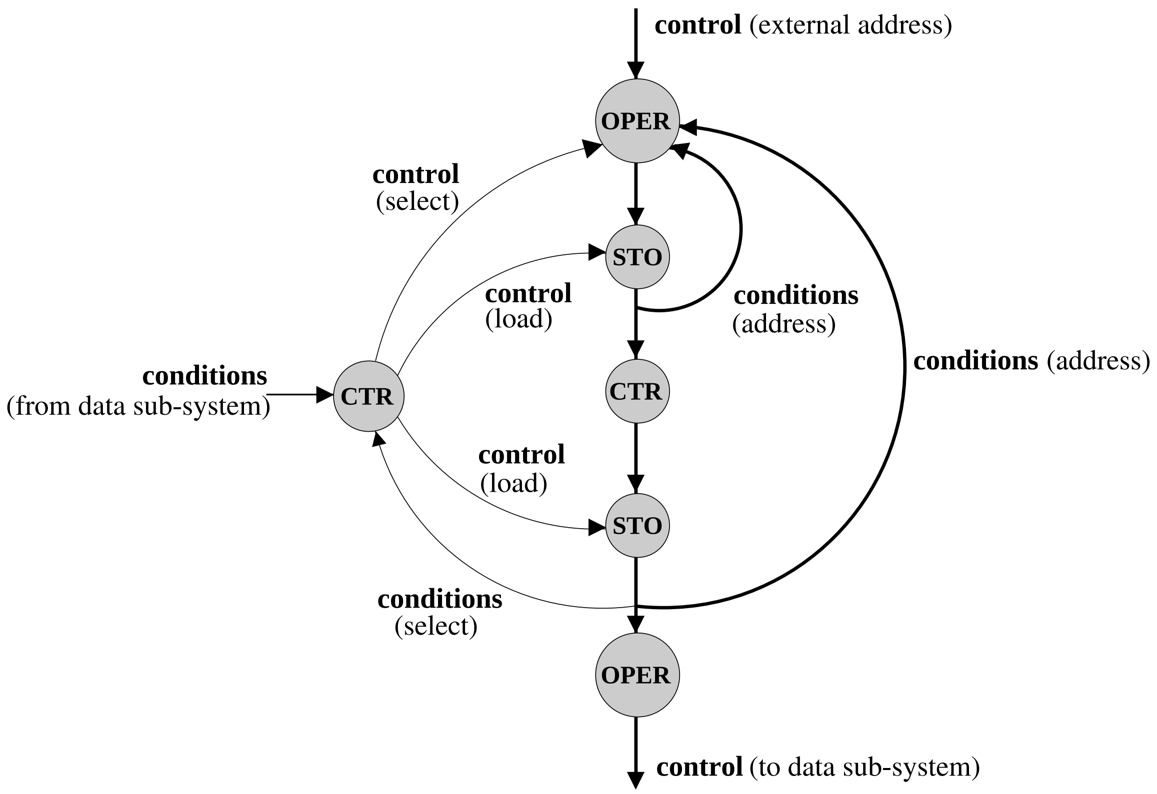


Figure 7.6: Data/Control Signal Flow Example

signals). A storage node must have data input and data output ports which usually have the same size (parallel input/output) and a data flow type of transporter (since the data cannot be modified). It is also possible to have all combinations of serial/parallel input and serial/parallel output (but see transducer nodes below) [4]. Storage nodes have control inputs which take boundary signals or internal signals from control nodes and, on some occasions, control outputs (e.g. to indicate that the device is full in a stack-type storage or to indicate completion of a read/write operation). Large storage nodes require addressing mechanisms to decode an address value provided through control input ports.

3. *operator*: an operator (OPER) node always takes a set of values to operate upon through its data input ports, and nearly always produces output values through its data output ports for the result of the operation. They often have control outputs to flag the result of an operation. In the case that there are no data outputs (e.g. a comparator) control output ports issue condition signals which can be used to make data-dependent decisions. Control nodes are mostly used for the control of operators (see figure 2.5 for alternative configurations).

[4]Serial input/output capabilities are often provided in storage nodes for testing purposes. This is often additional to the main parallel input/output use of the device.

4. *switch*: these nodes perform a controlled transfer of information from their data input ports to their data output ports. A data transfer path is selected by means of control input ports and they usually do not have control output ports. A set of interrelated storage, operator and switch nodes (a data-path) is often under the supervision of a single control node as shown in figure 7.6. Transfers of data in a data-path must be possible according to the data flow information in the nodes involved (see section 4.5.3).

5. *processor*: a processor node is a complex structure usually formed by a control unit which drives a data unit. It has all types of ports. For a processor, the hardware must be able to execute a sequence of instructions of a stored program: a firmware control implementation is required in the control unit.

6. *transducer*: this is a complex node for transmitting data into the system and out of the system. Data boundary signals are required. Often, the data can be stored in the node and code conversion, error detection and error correction may take place. Serial-to-parallel and parallel-to-serial transformations are often used for the communication of data into and out of the system, respectively. Data input, control input and data output ports are necessary and control output ports are often used (e.g. to indicate termination of a transfer of data).

The interrelations between nodes are imposed by means of the control of the data-paths. The data-path controlled by a control node can be identified by following the control signals generated by the control node and by establishing the flow of information between the nodes controlled. Control nodes interact with data-paths by means of control and condition signals (see figure 7.3(c)) defined as follows:

- an internal signal going from a control node to a non-control node (such as storage or operator nodes) involving control ports is of a control type.

- an internal signal going from a non-control node to a control node involving control ports is of a condition type.

An example of the kind of automatic reasoning that can be achieved for the example of figure 7.6 is as follows: an external address is passed to the system; an operator node operates upon the address, and possibly addresses of previous operations, to generate a new address; an operation is selected in the operator node by means of signals issued by a control node; a storage node keeps the address result under the supervision of the control node; the value stored in this node addresses a control node; this node does not control other nodes but produces output values, which after being stored in an intermediate storage node (always under supervision of the main control node), are used as conditions by the main control node and they are operated (decoded) by an operator (decoder) for the control of a data sub-system; the main control node controls all the flow of information and also makes use of condition signals issued by the data sub-system [5].

[5]The intermediate control node corresponds to a ROM memory. A ROM memory can never be a storage node since it is a combinational device used for the implementation of combinational networks (see

As a result, the whole circuit in figure 7.6 can be abstracted away as shown in figure 7.7. In figure 7.7(a), the data-path under control is identified as indicated above. The signals for the control of the data sub-system depart from the operator which is connected to the data-path. For this reason, it is possible to view the situation as shown in figure 7.7(b) which corresponds to a typical data/control structure (see figure 2.6). The circuit can be globally seen as shown in figure 7.7(c): the circuit takes condition signals from the data sub-system and, possibly, an external address, to control the next operation in the data sub-system.

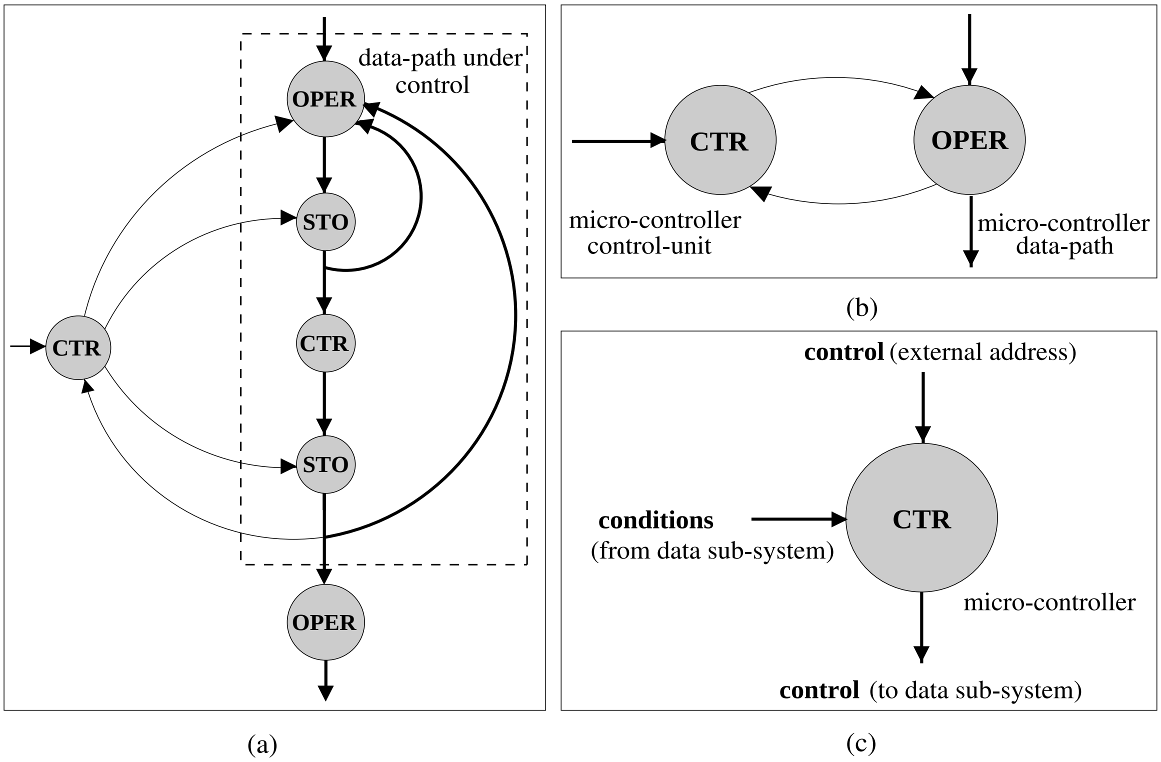


Figure 7.7: Data/Control Signal Flow Reasoning: (a) data-path under control, (b) identification of a data unit, and (c) overall view

7.6 Stereotypical Implementations

Designs are occasionally represented as a large network of transistors, gates or even vector level cells as the only level in the design hierarchy. The knowledge-propagation functions discussed above are not capable of understanding networks of gates or networks of transistors (considering a transistor as a switch) and they may become unworkable for

section 2.2.1). Data cannot be stored in a ROM (the information contained is pre-defined) but the device can be addressed, for example, for the generation of control signals (firmware control implementation) as discussed in section 2.3.2. In this case, a ROM is seen as a control node as opposed to an operator node.

the understanding of situations with a large number of nodes (even in the case that the nodes represent higher level cells). These problems, and mechanisms which attempt to overcome them, are discussed in this section. The mechanisms described are based on the observation that human experts exhibit an ability to view patterns of nodes as a whole for the understanding of a network.

Stereotypical implementations and problem-solving strategies are represented in the system as patterns and heuristic models, respectively. They represent the ability of combining electronic cells to implement more complicated functions in a way that reinforces the heuristic models described in chapter 5. The recognition of implementation patterns and heuristic problem-solving models corresponds to knowledge-generation functions which allow:

1. the interpretation of a group of nodes as a single larger node. The number of nodes that need to be considered for the understanding of a cell is reduced and a more meaningful design hierarchy may be generated.

2. the generation of new knowledge plans for existing cells. The fact that a number of nodes match an implementation pattern or a problem-solving strategy provides further positive evidence about the heuristic models of the interacting cells being right. The matching may result in the instantiation of undefined items for the heuristic models of some nodes.

Knowledge-extraction functions can also be used to determine repetitive patterns in a network. Identified patterns may be further grouped together to facilitate higher level abstractions.

7.6.1 Stereotypical Implementation Patterns

The heuristic models of cells at the gate and transistor levels do not contain much knowledge to clearly differentiate between them. For example, all gate level cells have data input ports and a single data output port and no control ports exist. It is difficult to gain decisive heuristic information from the analysis of a network of gates when all the ports are seen as having the same functionalities (and no generic purpose is defined for gate and transistor level cells). It is not even easy to differentiate, for example, an *and* gate from a *nand* gate since their interfaces and contents are very similar. On the other hand, low level cells are made of very few sub-cells and a small number of instances of these (e.g. a gate level cell usually contains a few interconnected transistors and a bit level cell usually contains a few interconnected gates). A higher level formulation of a network may be obtained by matching typical patterns of interconnections (*implementation patterns*) against the network (*contents pattern-matching*) as shown in the example of figure 7.8(a). The network is abstracted by matching patterns of typical bit level cells. For example, the flip-flop cell of figure 3.2(b) is used as a pattern. The shaded regions in the figure represent parts of the network which match the pattern. As a result, the design is abstracted into the network of figure 7.8(b).

Contents pattern-matching is based on the matching of typical transistor and gate level implementation patterns and patterns which can be obtained from the processing

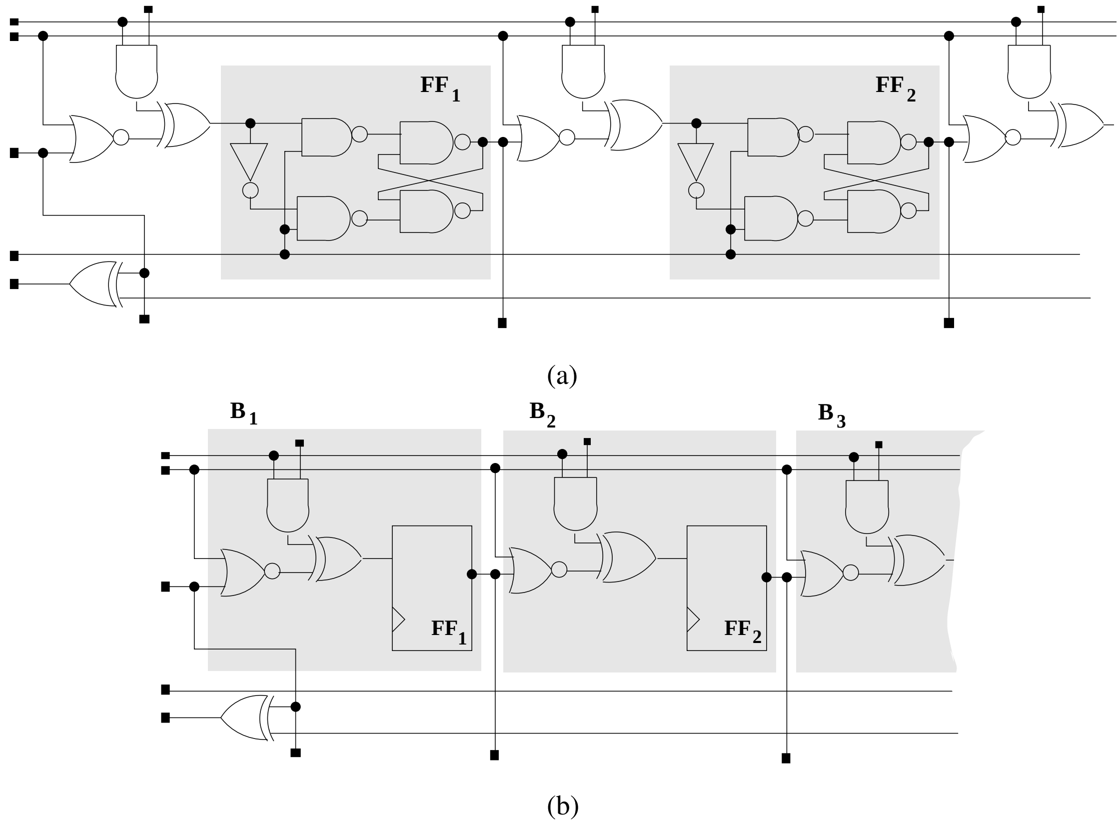


Figure 7.8: Contents Pattern-matching: (a) gate level network example, and (b) abstraction of the network.

of past designs (*logged implementation patterns*). An exact matching of these patterns is a feasible problem even for large networks of gate and transistor level cells [Mir89]. These cells have a very small number of interfaces and the functionalities of the ports can be ignored: for networks of gates, pattern-matching does not require to take into account to which one of the input ports of a gate another gate is connected since all ports are functionally the same (the important point is that the gates are interconnected). Transistor cells have a control interface (e.g. the gate of a MOS transistor) but pattern-matching may easily consider all possibilities if the functionalities of the interfaces are undefined since the number of interfaces is so small.

The conclusions of the heuristic recognition of an electronic cell as discussed in chapter 5 can be corroborated by pattern-matching on the contents of the cell (for cells at the bit level or lower): if a cell can be represented by the heuristic model of a class of cells and by the cell model of an instance cell of the class, the implementation pattern of the cell instance can be matched against the contents of the cell to determine if these cells are the same.

The search for symmetries and repetitive structures in a network also leads to ways for the abstraction of a design. For example, the shaded regions in figure 7.8(b) represent parts of the network with exactly the same structure. Each region corresponds to an instance of the same bit level unit (the set of bit level units forms a vector level cell). These methods of abstraction are not based on knowledge available about the entities (sub-cells and signals) but on topological features captured from the specification. The implementation of these methods corresponds, therefore, to knowledge-extraction functions.

7.6.2 Problem-solving Strategies

The matching of implementation patterns is impractical for networks of bit and higher level cells. The cells involved have more complicated interfaces which require the consideration of knowledge about the functionalities of the ports interconnected: a port of a cell cannot be connected to any port of another cell. Besides, high level cells are not so strictly typical as required by contents pattern-matching. For example, register cells all have a set of typical features but many different implementations exist which may differ in minor details. It is easier in general, at these levels of complexity, to search for heuristic features which may have been used in the implementation. An analysis based on decisive heuristics can take place since the models of the cells involved are more elaborated. This analysis is based on the recognition of key design strategies used for the implementation as opposed to a detailed analysis of the interconnections. These strategies are represented as *heuristic problem-solving models*.

An example of a typical design strategy is illustrated by means of figure 7.9. Figure 7.9(a) represents a comparator cell. This cell compares two bit-vectors $\underline{\mathbf{A}}$ and $\underline{\mathbf{B}}$ of width n and flags one of the three outputs to indicate if $\underline{\mathbf{A}}$ is greater (**G**), smaller (**S**) or equal (**E**) to $\underline{\mathbf{B}}$. The input interfaces of a comparator are data input ports and the output interfaces are control output ports (since the outputs are typically used to make data-dependent control and sequencing decisions).

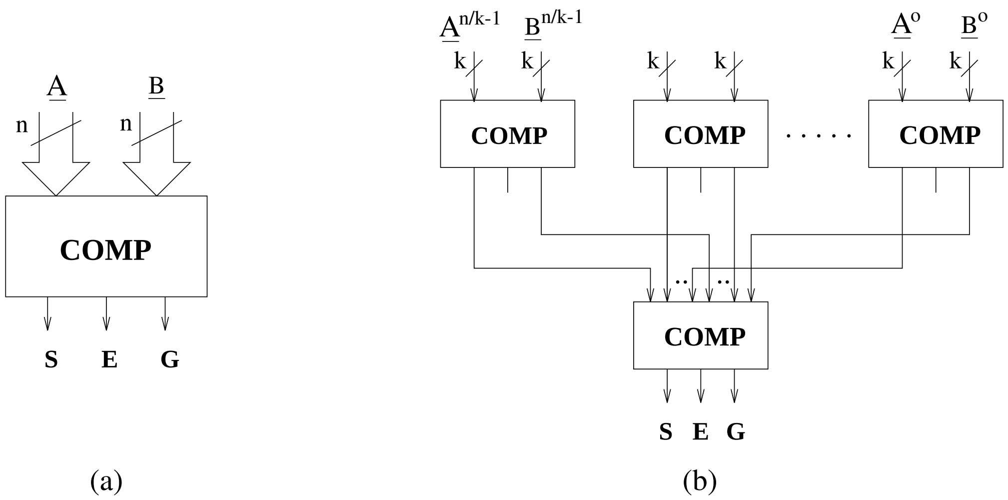


Figure 7.9: A Design Strategy: (a) a comparator cell, and (b) a tree construction of a comparator cell

A construction of the comparator of figure 7.9(a) is shown in figure 7.9(b). The construction corresponds to a tree organisation of instances of a smaller comparator sub-cell. Each sub-cell instance compares bit-vectors of width k. Each instance in the first level of the tree compares k bits of the input bit-vectors and the sub-cell instance of the second level performs a final comparison over the two bit-vectors of width k which come from the first level. This construction corresponds to a typical design strategy: a cluster of instances of a given cell often implements the same function as a single instance over a larger chunk of data (as discussed in section 2.2. Another example of this is the construction of the multiplexer of figure 5.2(c)). Of course, it is simpler to reason about a design that incorporates these circuits if there is a single node that represents the larger comparator (multiplexer) in place of a number of nodes each representing a smaller comparator (multiplexer).

Heuristic problem-solving models can be used for the representation of the high level architectural features of digital electronic systems discussed in chapter 2. The representation and matching of an heuristic model of a problem-solving implementation strategy can be done in three stages as shown in figure 7.10:

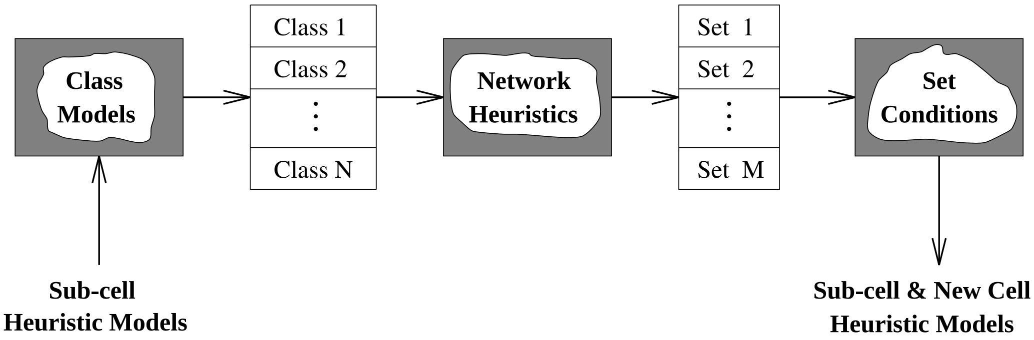


Figure 7.10: Heuristic Problem-solving Models

1. *node classification*: this stage defines the kinds of nodes (instantiated sub-cells) that can be involved in the strategy. A set of class models (described as in section 5.2) are considered for the strategy. The nodes corresponding to sub-cells which match a class model are candidate nodes for the strategy. A node may be a candidate for more than one class if the corresponding sub-cell matches more than one class model. For the example of figure 7.9, the class model of comparator cells is the only model required. All nodes in the situation which can be instances of a comparator sub-cell are considered.

2. *set definition*: this stage defines interconnection heuristics between nodes based on the knowledge represented in the corresponding class models. Candidate nodes which comply with these heuristics are grouped into mutually exclusive sets of nodes. In the example of figure 7.9(b), the interconnection heuristics include the connection of control output ports of a comparator with data input ports of another comparator. A set of nodes may contain mutually exclusive sub-sets. Each set or sub-set represents a typical grouping of sub-cells. For example, each collection of nodes in a

situation which can be grouped as shown in figure 7.9(b) forms a set (several groupings of comparators may exist in different parts of the situation). For each set, as many sub-sets as characteristic positions of a node in the strategy can be considered. For instance, in a tree network structure it is possible to consider as many characteristic positions as levels in the network. In the example of figure 7.9(b) there are two sub-sets (all the top nodes form the first sub-set and the bottom node forms the second sub-set).

3. *design reformulation*: this stage defines sets or sub-sets of nodes which can be viewed as single nodes. This results in the reformulation of the situation. For the example considered, each set can be viewed as a large comparator node. The reformulation can be made dependent on conditions imposed on the sets (e.g. the size or number of nodes in the sets).

7.7 Design Reformulation

The ways of reasoning described in section 7.5 and section 7.6 may result in the reformulation of the situation in order to facilitate its understanding. In general, the goals of design reformulation include the generation of a meaningful design hierarchy and the generation of alternative ways of looking at the design. The example of figure 7.11 is used to illustrated this. Figure 7.11(a) corresponds to the hierarchy of a design similar to the design in figure 2.7(a). The design consists of several instances of three different electronic cells as shown in the design hierarchy. An understanding of the operation of the design (see section 2.3.2) is based on the interrelation of the generic purposes of the nodes and the analysis of data and control signals between nodes.

A simplified way of looking at the design is the bit-slice approach given in figure 7.11(b). Each column of nodes in the design processes a slice of the overall block of data. This chunk is processed by three different nodes in cascade and a cell receives information from its right hand neighbour. An alternative way of looking at the design is the more functional approach given in figure 7.11(c). By grouping the nodes of each row new cells are obtained which perform the same function as any of its sub-cell instances but over the whole vector of data. The reformulation of the design generates an additional number of situations (cells in the design hierarchy) which will need to be considered in further reasoning cycles. However, it can significantly simplify the task of model-based reasoning (or even make it possible in the case of poorly organised designs) by allowing groups of nodes to be viewed as a whole.

In summary, model-based reasoning is applied during a reasoning cycle to each separate situation which forms the design (if the set of models which are used to represent the situation has not been considered in previous cycles). If a candidate set for a situation is considered valid, it can result in new knowledge plans for some cells of the design and in the reformulation of the situation. This chapter concludes the description of the automatic derivation of heuristic design knowledge. The current implementation is described in the next chapter.

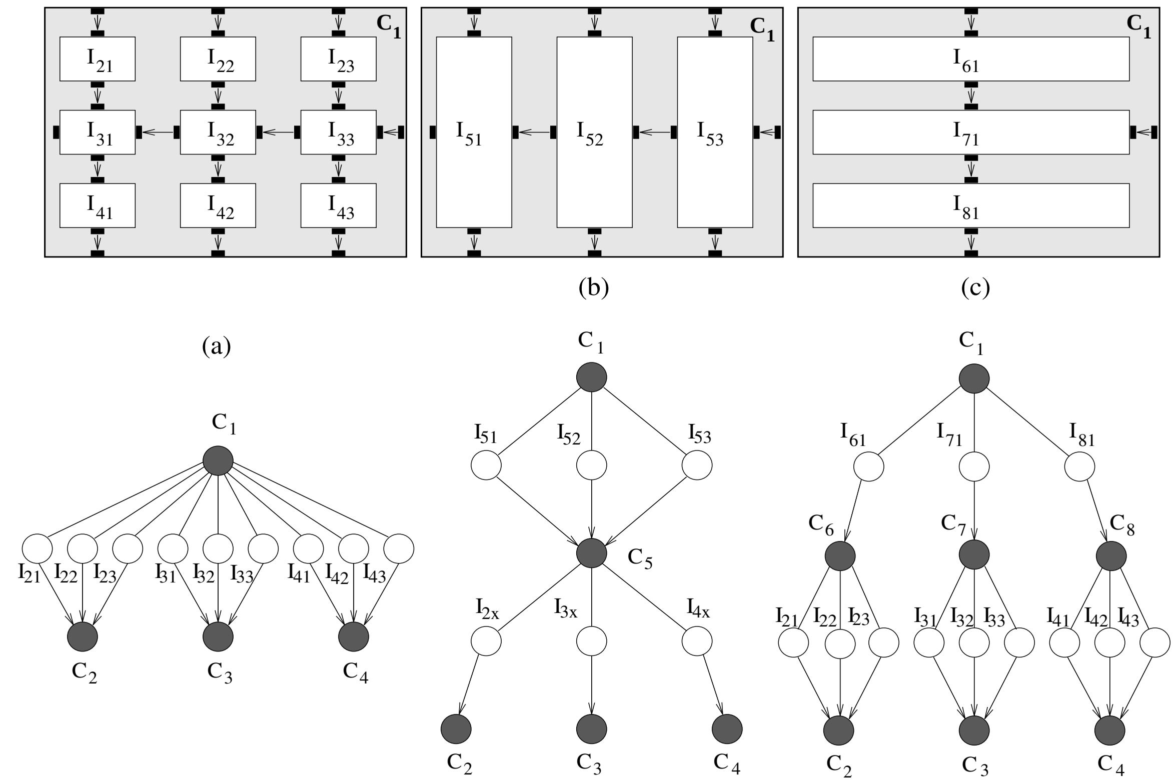


Figure 7.11: Design Reformulation: (a) example circuit, (b) bit-slice reformulation, and (c) reformulation based on functional blocks.

Part III

An Experimental Implementation, Applications and Conclusion

Chapter 8

Hercules: An Experimental Implementation

The experimental result of this research is a knowledge-based system aimed at providing an empirical demonstration of the viability of an automatic heuristic understanding of design specifications. The current implementation of the system is discussed. Indicators for measuring the performance of the system are presented. Results obtained with the system for the heuristic classification of electronic cells and signals of real designs are described. A discussion of the use of heuristic design knowledge for the planning and control of automatic ECAD tasks is given in chapter 9.

8.1 Hercules Overall Structure

The task of the KB system developed is the generation and selection of models for the cells (and signals) of a design by means of the heuristic analysis of its specification. The overall structure of the system is shown in figure 8.1. The input to the system consists of a specification of the design in the Electronic Design Interchange Format (EDIF). EDIF is a standard format designed to facilitate the interchange of electronic data between CAD systems [Ele87]. An EDIF input is parsed by means of the Manchester University EDIF parser [BK89] and a tokenised file is produced. This file is read by a *reader* program [KM87] which creates a data structure concerning structural information about the design and its objects (*netlist data*). The data structure is used to produce a representation of the structure and connectivity of the design in the logic programming language Prolog [SS86]. Only netlist EDIF views of the cells are considered for the representation of the design in Prolog.

The Prolog design representation contains factual information about the design objects: individual Prolog facts are used to represent cells, instances, ports and nets (signals). This representation is referred to as a net-oriented cell representation. A compact description of design cells is also required for the matching of implementation patterns with low level networks (see section 7.6.1). This representation, which is referred to as a part-oriented representation, is produced from the net-oriented representation of the cell. A unique clause or rule represents each cell. The head of the clause is the cell being defined and the body of the clause corresponds to the set of instantiated sub-cells used to represent the cell. The Prolog representation of design cells and the translation from

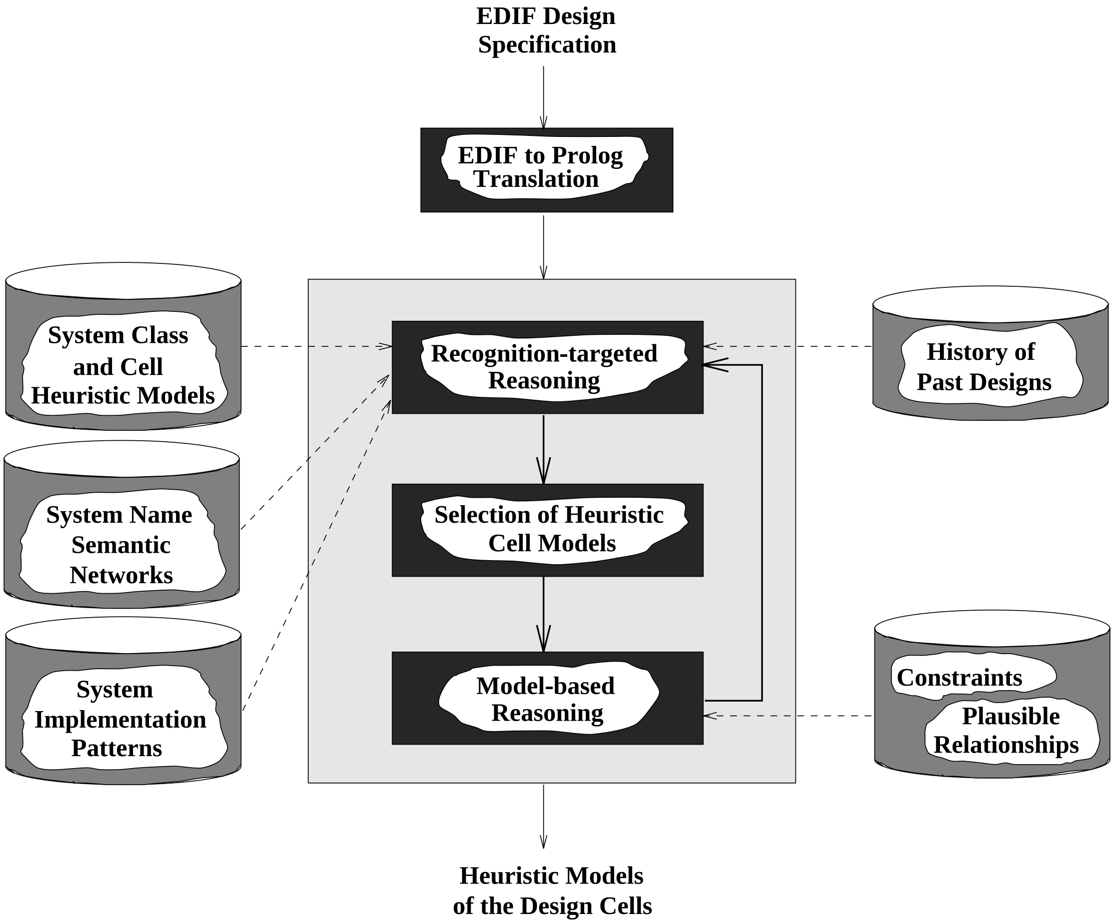


Figure 8.1: Overall Structure

EDIF to Prolog were developed during previous work and they are described in detail in [Mir89].

Each reasoning cycle of the system involves three steps as shown in figure 8.1:

1. a step of recognition-targeted reasoning for the generation of heuristic cell models from knowledge plans.

2. a step for the selection of heuristic models for the cells of the design.

3. a step of model-based reasoning about the representation of the design situations.

The level of expertise of the system and the quality of the results produced for the analysis of a design is obviously subject to the quality and amount of system knowledge. System knowledge which is used in the current implementation includes:

1. system knowledge for recognition-targeted reasoning which includes:

   (a) heuristic models of classes of electronic cells (21 class models which include classes of cells such as registers, multiplexers, logic gates, flip-flops, etc.).

(b) semantic networks and dictionaries for name processing. Two types of semantic networks are considered: networks for typical cell names (26 networks) and networks for typical port and signal names (18 networks). For each type of network a dictionary with the words involved in the network is used for name matching (see appendix B) (74 words exist in the dictionary of cell name words and 28 words in the dictionary of port name words).

(c) knowledge for the derivation of plausible cell types (see section 4.5) and for the interrelation of cell types and cell functionalities (see section 3.3).

(d) stereotypical implementation patterns of low level cells (see [Mir89]) (5 patterns) which are not used for obtaining the results presented in this chapter.

2. the evaluation function for the selection of cell models requires knowledge to estimate the complexity of a cell from a candidate model. This kind of knowledge is described in section 6.6.

3. system knowledge for model-based reasoning includes:

   (a) constraints and plausible relationships between the types of a cell and the types of its sub-cells (see section 4.5).

   (b) constraints about the electronic functionalities of interrelated ports and signals (e.g. a clock signal cannot drive a select port) (14 rules).

   (c) knowledge for the propagation of electronic functionality values and for the grouping of ports and signals as discussed in section 7.4 (see table 7.1 and table 7.2).

The system stores information about the processing of past designs. This history information can be added to the system knowledge for the processing of new designs. Heuristic cell models and implementation patterns of electronic cells formerly processed can be added to the system for the matching of new cells. Semantic networks and dictionaries obtained from the processing of past names can be merged to the system networks and dictionaries for the matching of new names. The system can run in an automatic or an interactive mode. In the automatic mode, system knowledge is used for the processing of a design and no user control is required. History knowledge is not automatically retrieved in the current implementation. In the interactive mode, the user can select history knowledge for processing a new design and browse knowledge available in the system and system results. Facilities for explaining the matching of class models and the matching of names are provided. The matching of implementation patterns is not yet included in the automatic mode but it can be considered in the interactive one. The case-studies discussed in this chapter are all processed in the automatic mode and no history knowledge is used.

8.2 System Strategy

The strategy of the system for the derivation of cell models is discussed in this section. The system executes knowledge-generation/knowledge-propagation cycles for the derivation of knowledge plans and cell models. Knowledge-extraction from the design specification takes place during the first reasoning cycle which forms initial knowledge plans for the cells. The formation of initial knowledge plans is carried out following a *bottom-up* strategy: the system starts with the cells which have the highest depth level in the hierarchy graph and works bottom-up towards the top cell of the graph. A cell is considered once its sub-cells have all been considered. The last cell to be considered is the top cell C_1 in the graph. This strategy can guarantee more detailed initial knowledge plans than other strategies (such as a top-down strategy): cells at lower levels of abstraction can in general be more easily and reliably classified than higher level ones. Knowledge about the sub-cells can then be passed to the cells for initially forming elaborate knowledge plans.

Considering that the n cells of the design are ordered from C_1 to C_n according to their depth level (where C_n is any of the primitive cells of the design which have the largest hierarchy depth level), the complete strategy of the system is as follows:

Step 1– **For** $i = n$ **to** 1 **step** -1 **do**
Form plans for cell C_i by knowledge-extraction and by propagation of the types of its n_i sub-cells.
Generate heuristic models for cell C_i by comparing these plans with heuristic class models in the system.
Select set of models for the C_j cells $(i \leq j \leq n)$ in the design which results in a higher evaluation value E_i for situation S_i.
EndFor

Step 2– **For** $i = 1$ **to** n **do**
Apply model-based reasoning to situation S_i if the set of models for S_i has not yet been considered. This may result in new knowledge plans if the representation of the situation is valid.
EndFor

Step 3– *Prompt termination condition: if the current representation for all situations has already been considered (no new models were selected) or no new more plans can be generated from the analysis of the selected solution set, stop.*

Step 4– **For** $i = 1$ **to** n **do**
Generate new heuristic models by comparing unconsidered plans with system heuristic models.
EndFor

Step 5– *Select best set of models for the overall design and go back to step 2.*

The prompt termination condition is required since it is not possible in general to consider all candidate solution sets. It is clear that reasoning about poorer candidate sets (sets which give lower evaluation function values) might result in new plans which could lead to more refined solution sets. The current prompt termination condition can be relaxed by considering other alternatives such as allowing the processing of a maximum number of candidate solution sets (for the overall design or for each individual situation) or by pruning from the search space those candidate solution sets whose evaluation falls below a threshold value (these alternative termination conditions are not considered for the results presented).

8.3 Current Status of the System and Limitations

The prototype of the system has been implemented using Sicstus Prolog [CW88]. The results presented in this chapter have been obtained by running the system on a Sun4/330 computer. The current implementation contains 3559 lines (15% comments) of C code and 26719 lines (24% comments) of Prolog code with 1500 Prolog predicates approximately. The knowledge-derivation functions which are currently used in the implementation include:

1. the k-extraction functions discussed in section 4.4 for deriving knowledge about the electronic functionality of the design objects from the analysis of their names, and in section 4.5 for inferring the types of a cell.
2. the k-generation functions discussed in section 5.5.1 for the comparison of knowledge plans with heuristic models of classes of cells, and in section 7.6.1 for the matching of stereotypical implementation patterns.
3. the k-propagation functions discussed in section 4.5 for the propagation of knowledge about the types of the cells, and in section 7.4 for propagating knowledge about the electronic functionalities of ports and signals according to the connectivity of the design.

The limitations of the current implementation include:

1. the knowledge-derivation functions based on the analysis of data/control signal flow and based on the matching of problem-solving strategies are not implemented.
2. plans and models are not hierarchically organised. Lists are used instead, which implies that the whole list of plans or models for a cell must be examined to determine if a plan or a model for a cell has already been considered.
3. the number of ports which can be ignored when matching the interfaces of a plan with the interfaces of an heuristic model (see section 5.5.2) must be lower than 25% of the total number of ports.
4. the number of sub-cells which can be ignored when matching the contents of a plan with the contents of an heuristic model (see section 5.5.2) must not represent more than 25% of the total number of estimated transistors for the cell.

5. the weighting factors for the slots of a plan or model are arbitrarily chosen according to the relative importance attributed to each slot ($0 < r_k < 1$). The choice of table 8.1 guarantees a value for the evaluation of a model or a plan close to 1 in equation 6.9. All the sub-slots of a slot have the same weighting factors which are also arbitrarily chosen for each type of compound slot.

Slot	**Weighting Factor**
Name	0.7
Types	0.2
Interface	0.6
Contents	0.6
DataFlow	0.2
Electronic Functionality	0
Constraints	0.5

Table 8.1: Slot Weighting Factors

6. the number of possible combinations for the matching of the interface slot of a plan with the interface slot of a system model cannot exceed 1000. If the number of combinations is larger, the matching of the plan is not considered.

7. a name can only successfully match one semantic network of names. A minimum evaluation value of 0.15 is required for considering that a name matches a network. A minimum evaluation value of 0.2 is also required for considering a word in a name close to a word in a semantic network, and a maximum number of 5 words which are close to a word of a name can be retrieved from the dictionaries.

8. the maximum number of failed sets which can be analysed for a situation of a design is set to 1000.

The value of these system input parameters is chosen as a result of the experimentation with the case-studies discussed later in the chapter.

8.4 Number of Plans/Models Derived

This section calculates the number of knowledge plans and models that can be derived for the cells of a design according to the k-derivation functions at present implemented. The calculation illustrates the way in which the system operates and defines indicators which are used for evaluating the performance of the system. The term fm_k indicates the number of plans that are available for matching in the k-th reasoning cycle and f_k the number of new solution plans (models) generated during the cycle.

The knowledge plans extracted during the initialisation cycle include information about the types and interfaces of the cells. A single plan for the types of a cell is initially derived according to the analysis discussed in section 4.5. If more than one type is possible for a type sub-slot, the possible types are defined in the slot for the constraints of the plan (see the example plan of table 4.1). The initial planning of the interface of a cell is done according to naming as indicated in section 4.4. The planning of the interface of cell C_i generates at most $I_i + 2$ plans: one plan from the analysis of the names of the ports of the cell, one plan from the analysis of the names of the signals which connect the ports of the cell and one plan for each one of the I_i instances of cell C_i from the analysis of the names of the signals which connect them (in the case that C_i is a primitive cell the number of plans is at most I_i +1). The single plan for the types of the cell is combined with each plan for the interfaces of the cell. Considering that some plans for each cell may be repeated (since some of the cell interface arrangements will be the same) the number of possible plans extracted for the n cells of a design is given by

$$fm_1 \leq \sum_{i=1}^{n}(I_i + 2) - p = 2 * n + \sum_{i=1}^{n} I_i - p \tag{8.1}$$

where p is the number of primitive cells in the design.

The *instantiation ratio* of a design is defined as the quotient between the number of instances in the design and the number of cells n. This ratio indicates the average number of instances per cell and it is defined as

$$Ir = \frac{\sum_{i=1}^{n} I_i}{n} \tag{8.2}$$

Considering the instantiation ratio of the design the inequality 8.1 becomes

$$fm_1 \leq n * (Ir + 2) - p$$

The actual number of knowledge plans derived is given by

$$fm_1 = (1 - Ff_1)\,[n\,(Ir + 2) - p] \tag{8.3}$$

where the *filtering factor* Ff_1 with

$$0 \leq Ff_1 \leq 1$$

indicates the proportion of knowledge plans which have been filtered out for all the cells in the design during the extraction of knowledge since they were repeated.

Each one of the plans extracted may match some of the h heuristic class models in the system by means of the knowledge-generation function described in section 5.5. The comparison of a plan with the q-th heuristic model H_q in the system may result in a large number of solution plans as asserted by equation 5.5. A restricted knowledge-generation function as defined by equation 4.4 is used which produces at most one solution plan or model from the comparison of a knowledge plan and a system heuristic model. The solution plan considered is the one which has the highest confidence evaluation value (see limitations of this in section 4.6). Therefore, if there are h heuristic models in the system and fm_1 plans to consider, the number of solution plans f_1 that can be generated during the first reasoning cycle must be

$$f_1 \leq h\,(1 - Ff_1)\,[n\,(Ir + 2) - p]$$

and the actual number of solution plans generated is given by

$$f_1 = \mathcal{E}m_1\,(1 - Ff_1)\,[n\,(Ir + 2) - p] \tag{8.4}$$

where $\mathcal{E}m_1$ is the average number of new models generated per plan available for matching during the first reasoning cycle.

During a step of model-based reasoning, the examination of the valid candidate set selected can result in I_i+1 plans for cell C_i: one plan from the analysis of the relationships between cell C_i and its sub-cells and one plan from the analysis of the relationships of each one of the I_i instances of the cell (in the case that cell C_i is flat, only the instances of the cell can be used). The number of plans which are generated by model-based reasoning for the cells of the design in the k-th reasoning cycle ($k > 1$) must be

$$fm_k \leq \sum_{i=1}^{n}(I_i + 1) - p$$

which, by means of the instantiation ratio of the design defined by equation 8.2, gives

$$fm_k \leq n * (Ir + 1) - p$$

The actual number of knowledge plans derived by propagation of knowledge in the k-th reasoning cycle is given by

$$fm_k = (1 - Ff_k)\,(1 - Fe_k)\,[n\,(Ir + 1) - p] \tag{8.5}$$

where the *filtering factors* Ff_k and Fe_k

$$0 \leq Ff_k, Fe_k \leq 1$$

indicate the proportion of knowledge plans which have been filtered out for all the cells in the design in the k-th reasoning cycle since they are repeated or they had already been generated in previous cycles, respectively. The value of these factors must in general increase when k increases. Each one of the new plans formed for the k-th reasoning cycle may match once some of the h heuristic models in the system. The number of solution plans f_k that can be generated during the k-th reasoning cycle must be

$$f_k = \mathcal{E}m_k\,(1 - Ff_k)\,(1 - Fe_k)\,[n\,(Ir + 1) - p] \tag{8.6}$$

where $\mathcal{E}m_k$ is the average number of new models generated per each plan available for matching in the k-th reasoning cycle. Given K reasoning cycles executed, the number of models derived is given by

$$f = \sum_{k=1}^{K} f_k \tag{8.7}$$

8.5 Case-Studies

The case-studies which are used to evaluate the performance of the current prototype of the system are introduced in this section. They correspond to real electronic designs obtained through the EDIF Technical Center at the University of Manchester. The designs are ordered in table 8.2 according to the size (in bytes) of the EDIF and Prolog files which represent the designs. The size of a Prolog file depends on the amount of netlist information available in the corresponding EDIF file.

Design Name	*Design Origin*	*EDIF size*	*Prolog size*
counter	CAD Lab. Univ. Manch.	2582	6226
h_bilbo	CAD Lab. Univ. Manch.	9565	13065
add	Rutherford Appleton Lab.	18103	21048
valid1	Rutherford Appleton Lab.	19800	24338
6g011a	CAD Lab. Univ. Manch.	77394	56072
multmilldesign	Industrial Company	139111	77857
designtop1	Rutherford Appleton Lab.	163756	182181
18ara700a	CAD Lab. Univ. Manch.	210007	249726
cwheel1_0	CAD Lab. Univ. Manch.	1176672	262927

Table 8.2: Case-Studies

The design hierarchy of each of these electronic designs is given in appendix F. The designs 'counter', 'h_bilbo', 'add' and 'valid1' are designs of cells at the vector level: the design 'counter' describes a counter cell which is composed of a few primitive vector level cells; the design 'h_bilbo' describes a register cell of type bilbo [WP82] which is composed of bit level cells which are decomposed down to the gate level; the design 'add' describes an adder cell in terms of gate level cells; and the design 'valid1' describes a logic function at the vector level in terms of vector level sub-cells which are decomposed down to the gate level. The designs '6g011a', 'multmilldesign' and 'designtop1' correspond to the design of small processor level cells which are decomposed down to the bit or gate level. Finally, the designs '18ara700a' and 'cwheel1_0' describe large processor level cells: the design '18ara700a' is decomposed down to the gate level and the design 'cwheel1_0' is mostly composed of vector level cells which are not further decomposed. The complexity of processing these designs and the results obtained are discussed in the next section.

8.6 System Evaluation Indicators

This section describes the indicators which are used for evaluating the performance of the system and the results obtained for the case-studies described in the previous section. These indicators measure the complication of the design being analysed, the complexity of the processing, the effectiveness of the knowledge-derivation functions and the quality of the knowledge derived.

8.6.1 Indicators of Design Complication

In order to compare the results obtained for the different designs, the following indicators are used as a measure of the degree of elaboration or complication of the designs analysed:

1. the *number of cells* n in the design.

2. the *number of primitive cells* p in the design.

3. the *dependability between cells* in the hierarchy graph. This indicator is defined as the quotient between the number of arcs in the hierarchy graph and the number of nodes or cells. It is calculated as

$$Dg = \frac{\sum_{i=1}^{n} n_i}{n} \tag{8.8}$$

   where n_i is the number of sub-cells (or down-dependencies) for cell C_i. For graphs in which each cell has at most one up-dependency (each cell is used in the definition of only one cell except for the top cell), such as in the example graph of figure 8.2(a), the value of Dg is always

$$Dg < 1$$

   and it tends to 1 when n increases. For graphs which have more than one path leading to a cell

$$Dg \geq 1$$

   such as the example graph of figure 8.2(b). The larger the value of Dg the more important is the interrelation between design cells (see figure 8.2(c)). As a result, the models of the cells must be consistent in a larger number of different situations.

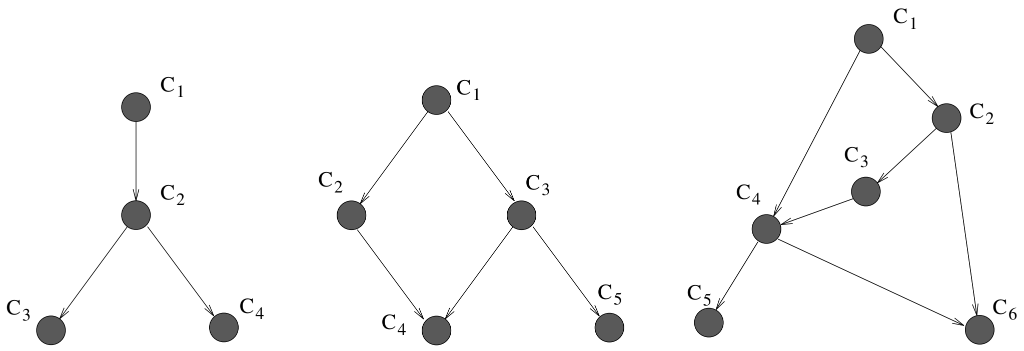
4. the *largest depth level* h_{max} of a cell in the design which is defined as

$$h_{max} = max\{h_1, \ldots, h_n\} \tag{8.9}$$

   The larger the value for h_{max} the larger the number of reasoning cycles which can be expected since knowledge is propagated level by level in each cycle.

5. the *instantiation ratio* of a design defined in equation 8.2. Considering that each instance of a cell offers new possibilities for the derivation of knowledge about the cell, the larger the instantiation rate of a design the larger the number of knowledge plans which can be expected.

6. the *average number of ports per cell* P_{av} in the design. This is an indication of the level of abstraction of the cells of the design (bidirectional ports count twice). The larger the number of ports the more complex is the matching of knowledge plans with class models.



Cell	h_i	n_i	ud_i
C_1	1	1	0
C_2	2	2	1
C_3	3	0	1
C_4	3	0	1

Dg = 0.75

(a)

Cell	h_i	n_i	ud_i
C_1	1	2	0
C_2	2	1	1
C_3	2	2	1
C_4	3	0	2
C_5	3	0	1

Dg = 1

(b)

Cell	h_i	n_i	ud_i
C_1	1	2	0
C_2	2	2	1
C_3	3	1	1
C_4	4	2	2
C_5	5	0	1
C_6	5	0	2

Dg = 1.16

(c)

Figure 8.2: Dependability between Cells

7. the *average number of paths per cell* L_{av} in the design. If L_i is the number of paths that lead from cell C_1 to cell C_i, this indicator is calculated as

$$L_{av} = \frac{\sum_{i=1}^{n} L_i}{n-1} \tag{8.10}$$

Similarly to Dg, the larger the value of L_{av} the more important is the interrelation between cells.

Table 8.3 contains the value of these indicators for the designs analysed. The design 'counter' has a significant number of ports per cell which affects the complexity of frame matching and it has few cells (n), few instances per cell (Ir) and just one situation (n-p). The design 'h_bilbo' includes a larger number of cells and situations and the cells are rather small (P_{av}) which facilitates the matching and identification of cells. The design 'add' has a significant number of cells and situations. As for the previous design, it has a substantial design hierarchy. The dependability between cells (Dg) is also significant. The cells are quite small ($P_{av} = 3.71$) and the number of instances per cell is also low. The discussion for the design 'valid1' is similar to the previous design. The dependability between cells is poorer for this design ($Dg < 1$) and the cells are on average larger.

Design Name	n	p	Dg	h_{max}	Ir	P_{av}	L_{av}
counter	4	3	0.75	2	0.75	12.50	1
h_bilbo	10	6	1	4	1.50	4.50	1.11
add	14	3	1.36	6	1.86	3.71	1.46
valid1	18	15	0.94	3	1.17	5.11	1
6g011a	18	6	2.06	4	10.72	4.94	2.18
multmilldesign	17	15	1.29	3	11.41	6.18	1.38
designtop1	35	17	2.43	4	12.97	8.51	2.5
18ara700a	40	8	3.5	7	15.15	14.43	6.9
cwheel1_0	32	31	0.97	2	6.50	26.62	1

Table 8.3: Indicators of Design Complication

The designs '6g011a' and 'multmilldesign' are larger designs. They have a similar number of cells which is slightly larger in the case of the design 'multmilldesign'. This design also has a slightly larger number of instances per cell. On the other hand, the first design has more situations ($18 - 6 = 12$) and the dependability between cells is substantially more important. The last three designs in the table, 'designtop1', '18ara700a' and 'cwheel1_0' have a large number of cells of a considerable size and a significant number of instances per cell. The design '18ara700a' has a deep design hierarchy and the dependability between cells is quite high. Finally, the design 'cwheel1_0' has no design hierarchy: all the cells are primitives of the design (except the top cell for which the interfaces are not defined in the specification as it happens for the design '18ara700a'). The design consists of a network of large vector level cells with a significant number of instances per cell.

8.6.2 Indicators of Processing Complexity

The complexity of processing a design is measured by means of four groups of indicators:

1. Indicators related to the processing time:
   - the total time for the processing of the design T.
   - the average processing time per cell T_{av}.
2. Indicators related to the number of reasoning cycles:
   - the number of reasoning cycles K required to reach the solution set.
   - the average number of productive cycles K_{av} per cell (a cycle is productive for a cell if new models are generated for a cell during the cycle).
   - the number of failed situation representations Nf which is defined as

$$Nf = \sum_{i=1}^{n-p} Nf_i \tag{8.11}$$

where Nf_i is the number of failed sets in the i-th situation, n the number of design cells and p the number of primitive cells in the design.

3. Indicators related to the number of models and plans:
   - the number of cell models generated Nm.
   - the average number of models generated per cell in the design Nm_{av}.
   - the number of cell knowledge plans generated Np.
   - the average number of plans generated per cell in the design Np_{av}.

4. Indicators related to the heuristic selection of cell models:
   - the number of candidate solution sets N calculated in equation 3.2 as

   $$N = \prod_{i=1}^{n} m_i \tag{8.12}$$

     where m_i is the number of models generated for the i-th cell.
   - the average number of candidate solution sets per situation N_{av} calculated as

   $$N_{av} = \frac{\sum_{i=1}^{n-p} N_i}{n-p} \tag{8.13}$$

     where N_i is the number of candidate sets for the i-th situation in the design calculated as

   $$N_i = m_i \prod_{j=1}^{n_i} m_{ij} \tag{8.14}$$

     with m_{ij} the number of models for the j-th sub-cell of cell C_i.
   - the number of failed candidate sets NF for the overall design. This considers the sets failed for the separate situations and the sets which failed for the overall design (e.g. if two cells in different situations have the same model and the system is forced to look for different models for each cell of the design).
   - the effectiveness of candidate set selection $\mathcal{E}$ defined as

   $$\mathcal{E} = 1 - \frac{NF}{N} \tag{8.15}$$

The indicators related to the processing time and the number of reasoning cycles for the designs analysed are shown in table 8.4. The processing time depends on the complexity of the design (as shown by the indicators of table 8.3) but also on the ability to form accurate knowledge plans with the knowledge available in the specification. If a small but significant amount of knowledge is initially extracted, a larger number of reasoning cycles can be generally expected since there will be more possibilities of adding knowledge to undefined items. For example, the designs 'counter' and 'h_bilbo' are accurately specified according to the knowledge in the system (see section 8.6.3) and only one productive reasoning cycle per cell (K_{av}) is used to reach the solution set. On the other hand, the system partially succeeds in extracting initial knowledge for the designs '18ara700a' and 'cwheel1_0' (see section 8.6.3) and several reasoning cycles (K) are executed before a solution is proposed. The processing time for the design 'cwheel1_0' is high since the number of reasoning cycles is higher than for previous designs and it has just one situation with a high number of instances of large cells (this significantly increases the time required for model-based reasoning).

Design Name	*Processing Time*		*Reasoning Cycles*		
	T (sec)	T_{av} (sec)	K	K_{av}	Nf_{av}
counter	7	1.8	2	1	0
h_bilbo	11	1.2	2	1	0
add	21	1.5	2	1	6
valid1	104	5.8	3	1.17	4
6g011a	84	4.7	5	1.44	2
multmilldesign	273	16	3	1.3	6
designtop1	563	16	4	1.1	11
18ara700a	476	11.9	5	1.4	2
cwheel1_0	10107	335	9	2.9	2

Table 8.4: Indicators of Processing Complexity – I

In general, the number of failed situation sets can be expected to be higher for designs which have high values for Dg and L_{av} since a cell must be consistent in a larger number of different situations. In the case of the designs 'counter' and 'h_bilbo' there are no failed situation sets since 'good' solution sets are initially generated. The average number of failed situation sets Nf_{av} is in general quite low for the rest of designs. Currently, a situation set can only fail because of three different reasons:

1. a constraint between the types of a cell and the types of its sub-cells is violated.
2. two or more cells in the situation are modelled with the same model (this test is optional when the system is run).
3. a connectivity rule is violated.

These three tests (together with the test for different models for the cells of the overall design) represent quite a limited ability to test for the consistency between models. The representation of knowledge about electronic design strategies is essential to strengthen the quality of the results and it is also important for the formation of highly accurate knowledge plans during model-based reasoning (see section 9.3).

The indicators which relate to the number of models and plans and to the heuristic selection of cell models are shown in table 8.5. The average number of plans generated per cell does not have any clear relationships with the indicators of design complication. This is correct since the ability to generate knowledge for a design should not depend on the structure of the design. The average number of plans generated per cell can critically depend on the ability to extract initial knowledge from the specification (see section 8.6.3). If only little knowledge can be extracted there will be few initial plans, the chances of matching system class models will decrease and so will the ability to form plans by knowledge-propagation. This is the case for the design '6g011a'. From table 8.5, the system appears to be quite successful in matching knowledge plans with system models for the generation of cell models. The average number of candidate sets per situation (N_{av}) is prohibitive for most designs which indicates that only with an effective mechanism for selecting models ($\mathcal{E}$ close to 1) is it possible to reach a solution. This is generally the case for the current implementation due to the prompt termination condition used and to the fact that model-based reasoning has quite limited knowledge for discarding combinations of models.

Design Name	*Number Models/Plans*				*Heuristic Model Selection*			
	Nm	Nm_{av}	Np	Np_{av}	N	N_{av}	NF	$\mathcal{E}$
counter	11	2.75	16	4	36	36	0	1
h_bilbo	50	5	68	6.8	6.1×10^5	7.3×10^2	0	1
add	72	5.14	89	6.36	2.1×10^9	2.8×10^2	2.8×10^8	0.87
valid1	127	7	166	9.22	1.1×10^{13}	3.4×10^8	1.0×10^9	1
6g011a	44	2.44	60	3.33	2.1×10^4	3.3×10^2	4.4×10^2	0.98
multmilldesign	133	7.8	189	11.12	1.4×10^{13}	7.0×10^{12}	1.8×10^8	1
designtop1	167	4.8	264	7.54	9.8×10^{18}	1.9×10^{11}	1.9×10^{14}	1
18ara700a	87	2.17	215	5.38	6.5×10^6	4.9×10^3	5.1×10^3	1
cwheel1_0	142	4.4	612	19.1	1.5×10^{18}	1.5×10^{18}	1	1

Table 8.5: Indicators of Processing Complexity – II

8.6.3 Effectiveness of the Knowledge-extraction Functions

The extraction of initial knowledge from a design specification clearly has a major contribution towards improving the quality of the results produced and towards reducing the complexity of processing the design. Currently, the most significant function for the extraction of knowledge is based on the analysis of names. Indicators of the effectiveness of name analysis are calculated as the proportion of names which matched with already existing semantic networks. For example, considering that the names of the ports of the electronic cells often refer to their electronic functionality the quotient

$$NM_p = \frac{n^{\underline{o}}\text{ port names matched with existing networks}}{n^{\underline{o}}\text{ of port names}} \tag{8.16}$$

is an indicator of the success in the identification of the electronic functionalities of cell ports by analysis of port names. Similarly, the indicators NM_c and NM_n are indicators of the success in the identification of the electronic functionalities of cells and signals by analysis of cell names and net names respectively.

The indicators of name analysis are correlated with a more general indicator $\mathcal{E}_{ext}$ which measures the effectiveness of the knowledge-extraction functions. This indicator is defined as the quotient

$$\mathcal{E}_{ext} = \frac{f_1}{\sum_{k=1}^{K} f_k} \tag{8.17}$$

where f_1 is the number of plans generated as a result of the initialisation cycle, f_k is the number of plans generated as a result of the k-th reasoning cycle and K is the number of reasoning cycles executed. The value of these indicators for the designs analysed is given in table 8.6.

Design Name	*NMc* (%)	*NMp* (%)	*NMn* (%)	$\mathcal{E}_{ext}$
counter	100	52	75	0.94
h_bilbo	90	94	31	0.95
add	50	0	9	1
valid1	61	0	0	0.94
6g011a	66	0	0	0.36
multmilldesign	94	20	4	0.84
designtop1	48	15	16	0.85
18ara700a	55	0	0	0.19
cwheel1_0	15	0	44	0.25

Table 8.6: Effectiveness of the Extraction of Knowledge

The first two designs in the table, 'counter' and 'h_bilbo', are described with names which are meaningful to the system (they match with semantic networks existing in the system and therefore with known electronic functionality values). As a result of this, highly accurate knowledge plans can be formed which implies that an important proportion of knowledge plans for the cells of the designs are generated during the initial cycle ($\mathcal{E}_{ext}$ is close to 1). The analysis of cell names is still significant for the designs in the middle of the table, but port names and net names carry little information which can be understood by the system (port names and net names are just numbers in some designs such as '6g011a' and '18ara700a'). For the last design in the table, 'cwheel1_0', the names of most cells are significant but the system has no knowledge about the names of the off-the-shelf cells used. In general, the value of $\mathcal{E}_{ext}$ tends to be high when enough knowledge can be extracted by name analysis and low otherwise.

8.6.4 Evaluation of the Knowledge Derived

The following indicators are considered for evaluating the knowledge derived:

1. the average confidence in the models selected e_{av}. This is defined as the average value of the confidence in the model of each cell in the design.

2. the confidence in the design representation E_1. This corresponds to the evaluation of the top situation of the design according to equation 6.12.

3. the average complexity deviation of the design cells r_{av} which is calculated as

$$r_{av} = \sum_{i=1}^{n} \frac{|1 - r_i|}{n} \tag{8.18}$$

   Large values for r_{av} indicate that the modelling of the cells deviates from typical representations which significantly affects the value of E_1.

4. the proportion of models in the selected set which are defined *Td*.

5. the proportion of defined models in the selected set which are correct *Tc*.

The results obtained for the designs analysed are given in table 8.7. The results are clearly better for the designs in the upper part of the table for which the electronic cells are smaller. Cells with a small number of inputs and outputs are easier to match and they do not differ significantly from typical classes of cells. On the other hand, large cells can only be matched if their interfaces can be adequately organised. The number of correct models generated is at times significantly low, specially for the designs which contain the largest cells. Two main reasons justify this:

Design Name	e_{av}	E_1	r_{av} (%)	*Td* (%)	*Tc* (%)
counter	0.67	0.59	21	75	100
h_bilbo	0.77	0.9	86	80	100
add	0.55	0.2	217	85	50
valid1	0.65	0	54	77	43
6g011a	0.25	0.14	270	66	8.3
multmilldesign	0.66	0.45	10	82	50
designtop1	0.40	0.07	78	60	43
18ara700a	0.16	0	112	20	25
cwheel1_0	0.59	0.24	0	75	20

Table 8.7: Evaluation of the Knowledge Derived

1. As described in section 8.1, a reduced set of class models is used for the analysis of these designs. This set only includes generic models of typical classes of cells. Models for specific electronic cells (such as typical off-the-shelf cells) can be easily added to the system. However, the use of only a generic set of models has been preferred in order to estimate the quality of the results produced in a more general environment.

2. The system requires more advanced model-based reasoning functions for testing the consistency between the cell models in a situation and for propagating knowledge. For example, the system is unaware of typical strategies of electronic design (see section 7.6.2).

The evaluation of the overall design representation is in general low. This is because the current complexity estimation function is only a rough approximation of the reality as discussed in section 6.6. For the case of the design 'cwheel1_0' the value of r_{av} is 0 since the top cell of the design is not defined (no interfaces are given) and the complexity of the cell can only be estimated from its contents. The evaluation of designs 'valid1' and '18ara700a' give a result of 0 since the models of the cells involved in the top situation are all undefined.

8.7 Discussion

This chapter has presented the current implementation of the system which presently contains a reduced example set of class models and semantic networks. The system is fairly successful in the identification of electronic cells in realistic designs but the knowledge of the system must be enhanced if high quality results are sought. The functions used for model-based reasoning are not restrictive enough and knowledge about electronic design strategies is required in order to improve this. The results of the system should largely improve if it is endowed with a significant amount of class and cell models (such as models describing the cells of typical families of circuits used in electronic design) since:

1. The complexity of recognition-targeted reasoning only grows linearly with the number of class and cell models in the system (and this should considerably improve if the models are organised hierarchically).

2. The estimated complexity of the selection of models is not exponential with the number of models available in the system.

3. The complexity of model-based reasoning does not depend on the number of class models in the system or in the number of models available per cell.

As a consequence, higher quality results should be generated if more system knowledge is available at the expense of an affordable increase in processing time.

Chapter 9

Conclusion

An heuristic classification of the cells and signals of a design symbolises a specification-level understanding of the design. The reasoning method used for reaching this level of understanding has without question limitations but it can in general, by exploiting knowledge implicit in the specification, allow the capabilities of ECAD frameworks and systems to be extended. Methods and strategies for the analysis, design and management of electronic data can be better planned and controlled by taking into account knowledge generated by means of an heuristic analysis of the specification. Examples of real and viable applications are discussed. Further work is necessary for a formal integration of design data and heuristic design knowledge, for enhancing the reasoning mechanisms upon which the heuristic classification of electronic data is based, and for the application of the knowledge generated to actual problems in the field.

9.1 Design Understanding — Limitations

A specification-level understanding of the description of a design is symbolised by means of a set of models which classify its cells and signals. The success or failure of the reasoning process for the generation of a consistent high quality solution set depends on the ability to extract initial knowledge from the specification, on the ability to form appropriate knowledge plans for the cells of the design, on the amount of knowledge available in the system and on the ability to select the most adequate models. The limitations of the reasoning process imply that the results produced must either be accepted with reservations or, hopefully not very often, partially or fully rejected. These limitations do not represent a critical drawback: human experts find themselves in a similar position when they attempt an heuristic understanding of a specification. The limitations which the system faces include:

1. the way in which knowledge plans are formed (by extraction, generation or propagation of knowledge) is heuristic: the plans are viewed with limited confidence and all knowledge plans which can be formed from the combination of all planning possibilities for the separate items of knowledge involved in a plan are not generated (see section 4.6).

2. the methods for evaluating the confidence in the items of knowledge generated and for evaluating and selecting solution sets are heuristic: they are not probabilistic.

The use of heuristic methods is consistent with the idea of attempting to simulate human reasoning for the evaluation and selection of solutions since human experts do not appear to be probabilistic reasoners only (see section 6.2).

3. the reasoning process may halt even though more refined solution sets could still be possible: a *prompt termination condition* is required since it is not feasible in general to consider all possible candidate solution sets. Typically, the reasoning is halted when it is not possible to add further knowledge for the formation of new knowledge plans to the best solution set generated. However, it may be possible that reasoning about poorer candidate sets leads to more refined solution sets after further reasoning cycles. The termination condition can be relaxed (for example by allowing the processing of a maximum number of candidate sets or by pruning from the search space those candidate sets whose evaluation falls below a threshold value) but it cannot be disregarded in most cases.

The limitations of the reasoning process imply that even in the case that a best solution exists for the understanding of a design the system may be unable to find it. For those cases which are beyond the computational capabilities of the system or the knowledge available to the reasoning process, the aims are reduced to an exploration of the architecture of the design (*architecture exploration*) and to the understanding of substantial parts of it as opposed to overall understanding. In any case, it is clear that the types of expert knowledge considered for the representation of an heuristic understanding of the design can be derived from the study of available behavioural data. At least in two cases detailed behavioural data must be used for the derivation of expert knowledge:

1. when the system is unable to derive enough knowledge for the understanding of the design, and this knowledge is judged convenient to carry out a task or support an hypothesis.

2. when a formal proof of the validity of the expert knowledge generated is needed in order to guarantee the quality of the results of a task or the legitimacy of an hypothesis which are based on this knowledge.

The derivation of expert knowledge from mathematical data can be achieved by simulation of the electrical and logical behaviours of the design (e.g. it is possible to consider stereotypical simulation plans to verify if a cell can actually behave according to a given electronic functionality). This is also a human expert activity which is often subordinated to an initial analysis of design semantics. As a last resort, human experts can be queried, with the queries being automatically addressed by the system according to estimates of the relative importance of the items of knowledge required (see appendix A).

The way of reasoning reflects expertise and common sense since it is based on knowledge of an heuristic nature rather than a systematic way of approaching the solution. The reasoning for the classification of electronic data comes close to exhibiting a convincing machine understanding ability as defined in section 1.4 for the following reasons:

1. the system has the power of representing knowledge about the domain and to reason about it effectively. Chapter 3 describes the kinds of expert knowledge that are used

in order to support the simulation of the expert understanding of digital electronic designs discussed in chapter 2. Chapters 4 to 7 present the mechanisms for capturing this knowledge and reasoning with it.

2. the ability of the system to heuristically classify electronic data is a powerful mechanism for perceiving equivalences or analogies between different representations of the same or similar situations. The knowledge-generation functions discussed in chapter 5 and section 7.6 attempt to perform these types of tasks.

3. the system has a limited though realistic ability to learn. It can use knowledge captured from the processing of cells of previous designs (logged cell models). The ability of the system to classify electronic cells, in combination with the ability for name matching and for learning about new names (see appendix B), may result in the formation of new categories for some types of knowledge (cell and port electronic functionalities) which may accurately describe design objects and which can be used for the understanding of new designs. The current system implementation can be enhanced in order to automatically generate classes of cells and discriminate between cell instances of a class as discussed in section 5.4.

9.2 Applications

Automatic systems can only process explicit electronic data which strictly meets the formalities of a hardware description language. They cannot exploit implicit knowledge which is provided by human designers in order to facilitate the understanding of the specification. By overlooking implicit knowledge, automatic systems are clearly at a disadvantage with respect to human experts for the processing of electronic data. The automatic heuristic understanding of design specifications presented in this work attempts to narrow this gap. General applications which can be derived from this understanding and applications to current research activities are discussed in this section.

9.2.1 General Issues — ECAD Frameworks

The management of the large amounts of data which are required for representing complex electronic circuits and the presentation of this data in a useful and efficient form to CAD tools, designers and manufacturing equipment has become a major issue in the electronics industry [HNSB90]. Large and dedicated software environments (ECAD frameworks) are required to address the complexity of the design process and of the data associated with the representation of the designs. An ECAD framework contains mechanisms and facilities (e.g. programming libraries, extension languages, data management and user interface facilities), at different levels of abstraction, which are used by the tool developers, tool integrators and often the end-users (electronic designers) for the configuration of a particular CAD system. This section examines the integration of a tool for the heuristic understanding of electronic specifications into an ECAD framework by giving examples of the ways in which different framework services can make use of this tool:

1. *project management and version services*: these services are considered for evaluating and displaying the progress of a design and managing its history or evolution. Versions and alternatives for the design objects, and specific configurations or collections of related versioned design objects, must be kept. This allows the exploration of alternative design implementations and provides an easy way to recover from bad design decisions. The right versioned objects must come together when the final design is assembled. This is not always trivial since the design of the cells of a system may be developed in parallel, often by different designers, on different computer systems or even at different locations. Besides, a cell in a given stage of the design process is often fed back to previous stages after modifications have been introduced. Designers need to constantly interact to understand the purpose of new or modified functionality. It is also desirable in large design projects to compare alternate views of a design which use different hierarchies. Most existing techniques require essentially identical hierarchies or must flatten to remove the differences. The capture of a small amount of design knowledge makes the comparison of design hierarchies much easier to solve [Spr90]. The ability to classify electronic data can be exploited for the comparison of different design versions, for the understanding of added functionality implicitly described in the specification and for estimating the complexity of design tasks required for the project (as discussed later in this section).

   The complexity of a design project is generally reduced if electronic cells which have been designed in the past (or are available in the system) can be reused or adapted for a new design. Automatic systems are more limited than human experts for performing these types of tasks since they generally lack methods for the recognition of similar electronic cells processed in the past. Matches between the entities compared must be exact. For human experts, the comparison among two different cells is less strict since they exploit their experience and common sense to only examine those aspects and data which appear convenient to judge the similarity between cells. A system cell or history cell can then be identified and reused or adapted for the purposes of the new design. The heuristic classification of data presents a clear opportunity for attempting to automate this type of task: an intelligent retrieval (or browsing) of system and history data can be performed by specifying heuristic knowledge which describes properties or categories of the electronic data required.

2. *data representation and management services*: they provide facilities for defining the data model and its particular database implementation, for managing the CAD data associated with the design and for coordinating access to the data by multiple users (human users or CAD tools). The data model defines a common representation for the information associated with the design so that two tools which read the same data interpret it the same way. The particular type of the data items supported in a particular database and the legal relationships that may exist between objects is usually referred to as the conceptual schema for the database which is often described by means of a data definition language (DDL). A user can extract specific sub-sets of information from the database via the database management system (DBMS). A query language, sometimes referred to as a data manipulation language (DML), is

used to specify requests to the database such as "find all the cells in the design which have two input ports and one output port". The fact that a cell can contain ports, and ports can be of various types such as input and output ports, must be part of the conceptual schema defined by the DDL. In engineering applications, many more queries are performed by CAD tools than directly by the end-users and a high level of integration in ECAD engineering frameworks is convenient. This is often achieved by merging the DML language with the language interfaces to other parts of the system, such as user interface, design flow management, and history management to form a common language interface or *extension language* to all the facilities in the framework [HNSB90].

A tool for the heuristic understanding of the electronic data can augment the capabilities for querying and browsing the database. For example, queries of the type "find all cells in the design which may be seen as a register" (a typical request as discussed below) are queries which designers can answer, even if detailed behavioural data is not available, and which normally are beyond automatic capabilities. The heuristic classification of electronic data can attempt to answer these types of queries. The types of knowledge used can be integrated in the data model and formally related to design knowledge after the heuristic analysis. The extension language must be enhanced to deal with this type of query. The extra amount of heuristic knowledge available can allow the system to move towards more human capabilities by answering queries which are often supposed to require intelligence, and by providing a high level ability to access, manipulate and classify/store design objects.

Another important feature of a DBMS is to provide for consistency checks to verify that the database is in a consistent state after modifications have been applied to it. In the engineering world, consistency checks are complex and time consuming. Often, full and exact data (as required, for example, for ensuring that a circuit performs according to its specification) is not available until the design reaches the final stages. Consistency of heuristic knowledge describing interrelated design objects can always be checked be means of heuristic knowledge-propagation functions. This type of consistency checking can be used to simulate human reasoning for those design phases in which full detailed information about a design and its parts is not available to check for design consistency.

3. *design methodology management* (DMM): these services view the design as a process which involves a sequence of operations (tools) each performed on design data. The user leaves some of the decision-making up to the design flow manager which invokes and controls CAD tools in order to meet some design goal which is beyond the scope of any individual tool. This facilitates the automation of tedious sequences of tool invocations. A DMM system is viewed as a 'metatool' in the CAD environment in the sense that it packages groups of tools into higher level entities which may be manipulated by the user as a single tool. It has an important part to play in synchronising the work of a team of designers and automating design cycles. A DMM system can be used to enforce designer's discipline (for example running design rule checking before approving layout changes).

A tool for the heuristic understanding of design specifications can help a DMM system to enforce high quality hardware descriptions. This can be achieved, for example, by insisting on designers making use of meaningful object names or by calling for a well structured design hierarchy. The tool can measure the quality of a specification from its ability to heuristically understand it. Heuristic knowledge about the functionality of the cells can be exploited by a design flow manager to automatically select the most adequate tools (e.g. selecting a PLA generator for the implementation of a piece of combinatorial hardware or a specific test vector generator for the testing of a piece of hardware identified as a memory cell). The relative importance and complexity of the different design objects (and collections of objects) can be estimated, even in the case that some pieces of hardware are not well understood, and estimates of the time required to perform design tasks may be calculated. These estimates can be exploited by the design flow manager to schedule design tasks, to select adequate tools or to implement load balancing on a single machine or a network of machines available to the system for performing the tasks (see example in section 9.2.2).

4. *user interface services*: these services provide high level facilities for constructing user interfaces and interacting with the user as needed. They must gather and present information efficiently and effectively. Explanation facilities based upon the methodology and the state of the design are increasingly important as the tools become more autonomous and several designers become involved in the same design. The heuristic classification of electronic data may provide guidance to design engineers and aid in displaying data in an efficient and meaningful form by virtue of its ability to abstract away functional heuristic information about design objects.

5. *communication and maintenance of electronic data*: a central issue in the electronics industry is the communication of design data. Support for industry-standard data formats such as EDIF and VHDL [IEE88] is relatively new and it is aimed at providing means of transferring design data via textual formats. In the absence of a single standard database or data model for electronic CAD, these formats have begun to make possible the communication of electronic data between CAD systems. However, a design high level specification or the design documentation may not be complete, reliable, or even available. Frequently, the only reliable sources of information are the hardware description code and the simulation of the design in a computer. Worst of all, the hardware description may be incomplete (e.g. behavioural data may not be available) and the description may rely on naming conventions or the attaching of properties to the design objects which are not in general understood by the system that receives the data. For example, a design can be created with a CAD system having an in-house library of cells, and the specification may only contain a partial description of the cells used. The names of the cells may be sufficient for some tools of the CAD system which generated the data, but this information will generally be meaningless for another CAD system receiving the data. Despite the use of standard formats, important design information may not be explicitly available to the CAD system, and an automatic heuristic understanding of

hardware descriptions may provide a valuable mechanism for the re-engineering and maintenance of hardware descriptions by facilitating the understanding of implicit knowledge.

In summary, a tool with the ability of heuristically classifying electronic data can extend the capabilities of ECAD framework services by attempting the simulation of activities which are normally considered to require human intervention. In addition, such a tool is useful for planning and controlling ECAD tools as discussed next.

9.2.2 Guidance and Control of ECAD Tools

Most ECAD tools process electronic data blindly according to the heuristics and algorithms they are programmed to use. The electronic data must meet the requirements of the ECAD tools and no intelligent machine interpretation of the data takes place. The tools are driven by data which is not deeply understood by the system, and they have to perform as dumb data-processors with little ability to plan and control the tasks performed according to the data processed. This section provides some examples of the use of heuristic design knowledge for the planning and control of design tasks:

1. *simulation*: heuristic knowledge about the operation of a cell can be used for planning the simulation of the cell. For example, a simulator tool may be simulating a multiplexer cell, but it does not *know*, at any time, that the cell performs a multiplexing function. Heuristic knowledge about the functionality of the cell and the functionality of its ports (e.g. select ports) can be used to plan and control the simulation. Stereotypical simulation plans can be selected to verify if a cell may behave according to the heuristic knowledge available (for example, its electronic functionality or its generic purpose) and the data transfer paths of the cell can be used as strategies for the simulation of the multiplexer. The simulator tool may be able to decide which signals to monitor and in which way [FM89]. Heuristic knowledge is useful for calculating estimates of the relative importance of the design parts, and these can be used for the generation of sensible simulation schedules. Estimates of the time required for the simulation of the overall design or parts of it could be calculated from available heuristic knowledge and used, for example, for load balancing. In the case that behavioural data is not available, heuristic knowledge about the functionality of the cell can be used to search in the database for a known cell whose behaviour may represent, approximate or aid in determining the behaviour of the cell in question.

2. *test*: the usefulness of heuristic design knowledge for controlling tasks such as hierarchical test generation has already been proved in successful systems such as Hitest [Ben84]. The designers of Hitest emphasise that high level design knowledge is crucial to effective test generation but Hitest only relies on human intervention for the derivation of this knowledge. Heuristic knowledge, which for example includes the intended use of cells, ports and signals, must be provided by the users. An heuristic analysis of the specification is specially adequate for generating the types of

heuristic knowledge which tools like Hitest require. Other testing tasks, such as the automatic insertion of design-for-testability (DFT) hardware [AB85, KTH88, JK86], can clearly exploit heuristic design knowledge. A structural description of the design and a vague understanding of the operation of some design objects is all that is required by these tools: register cells, data transporter cells and specific signals, such as control and clock signals, must be identified in the description. Special conventions are required for the recognition of these objects in systems which attempt the insertion of DFT hardware to circuits described in standard formats. For example, the insertion of DFT hardware in VHDL specifications as described in [KTH88] requires prefixes to be attached to the name of the objects to facilitate the recognition of register cells and signals. With the help of a tool for the heuristic analysis of design specifications, a DBMS system may be able to search for these objects. The types of knowledge generated by means of this analysis provide an adequate model for DFT applications and for planning and scheduling test plans.

3. *floorplanning*: heuristic design knowledge is of clear interest for tasks such as floorplanning. Traditionally, parameters such as chip area and wire length are of major importance for the automation of these tasks aimed at minimising area and communication costs. The generalised floorplanning problem appears computationally intractable for the great mass of general purpose chips, although specific floorplanners are very effective at laying out specific design families [Nix84]. Heuristic knowledge can be used to classify the cells of a design and choose the most adequate floorplanner. Expert knowledge, which includes estimates of the size and functionality of design objects and of the amount of connectivity which interconnects them, have been used in expert system approaches to floorplanning [JS89]. Knowledge about the functionality of design objects is required if the floorplanner attempts to optimise system performance instead of minimising area. For instance, the length of clock wires should be minimal in order to improve speed. As a second example, a register destined to store address values and connected to a memory cell, among other cells, should be placed close to the memory cell because of the frequent flow of information that can be expected between these cells (this increases circuit speed by reducing circuit capacitances in busy paths).

4. *routing*: the routing of a net is basically a geometric problem for the router, but this may differ depending on the signal carried by the net (e.g. clock signals should not be routed close to other clock signals in order to avoid cross-talk problems [RIXK91]). The type of layer used is also related to the functionality of a signal (e.g. power signals are usually done in metal layer). An understanding of the functionality of the signals of the design must then be captured from the specification. However, this type of knowledge is rarely explicitly provided. The routing of a design can be attempted by general purpose routers, or the design can be partitioned into separate routing areas for which restricted routers can be applied. Heuristic design knowledge can then be used to identify routing areas and select the most adequate routers.

5. *model-based diagnosis*: it is also interesting to note links of this research with model-based reasoning for the diagnosis of systems. Model-based reasoning for diagnosis has been studied in recent years as an alternative to empirical rule-based diagnosis [Dav84, Rei87]. An empirical diagnostic approach uses shallow knowledge to reason about devices (i.e. an expert experienced in diagnosing devices). On the other hand, model-based diagnosis assumes a complete and consistent theory or model of the behaviour of the device and its components. The research discussed in this thesis addresses issues of two of the major problems of model-based reasoning: how to obtain models for the devices and how to manage the exponential computational complexity of the reasoning methods.

9.3 Further Work

The research presented in this thesis has been used in the Esprit Special Project 5082 to emphasise the need to exploit heuristic design knowledge [FMW91]. Currently, it is been used in the Esprit Special Project 7064 (Jessi-Common-Frame) as an example of the way in which heuristic knowledge can affect the design of ECAD frameworks. Three main lines of future work related to this research are here considered necessary for arriving at a conclusive empirical demonstration of the convenience and viability of exploiting heuristic design knowledge:

1. a formal integration of the design data, the heuristic knowledge generated from the analysis of the specification, and the system information required for the derivation of heuristic design knowledge is necessary. A formal mapping between design data and heuristic knowledge provides an unified view of factual and heuristic design information, allowing a single procedural interface (see [YK90] for an example of these tools) to access and retrieve both kinds of information. It also provides a formal mechanism for the generation of explanations about the state of the reasoning process and about the reasoning path that the system followed to reach a conclusion. This is ongoing research in the CAD group at the University of Manchester using the information modelling language Express [Int91] under the support of the Esprit Special Project 7064. A formal Express representation of the heuristic models of electronic cells is being developed [BM93]. The aim is to move towards a formal integrated representation of electronic factual data and heuristic knowledge, covering all aspects and decisions in the design process. An intelligent automation of the design process may be envisaged in this way which can provide ECAD frameworks with high level facilities for giving advice to human designers.

2. a number of reasoning enhancements can be added to the system in order to improve the reasoning capabilities. These enhancements include:
   (a) the consideration of more advanced prompt termination conditions which can allow the investigation of a larger number of candidate sets.
   (b) a hierarchical organisation of models and plans in the system which can facilitate the processing of large numbers of them.

(c) the cell complexity estimation function considered in section 6.6 is useful for the illustration of the control mechanism but it is simplistic. A more accurate function must take into account more information about the cells instead of just considering their level of abstraction.

(d) the study of further k-derivation functions which can complement the existing ones in order to form highly accurate knowledge plans (the calculation of the confidence in individual items of knowledge for a plan by means of the current k-derivation functions does not have enough experimental support). Model-based reasoning must be enhanced providing the system with knowledge about problem-solving strategies typically used for electronic design (see section 7.6.2).

(e) the test-and-generate procedure for the matching of knowledge plans can be turned into a more efficient constraint-driven frame matching procedure.

3. further work is necessary for developing a wide range of applications of industrial interest based on the exploitation of heuristic design knowledge. The experimental tool presented in chapter 8 already demonstrates that valid heuristic design knowledge can be automatically generated, facilitating the understanding of a design and the re-engineering of hardware descriptions. Current state-of-the art tools (e.g. Hitest for hierarchical test generation) can definitely take advantage of this knowledge, but a wide range of successful applications aimed at improving the capabilities of framework services and CAD tools (see examples in section 9.2) are required to justify the integration of such a tool within advanced ECAD frameworks.

9.4 Afterword

An automatic system which is capable of generating heuristic design knowledge for the classification of electronic data is proposed in this work as a worthwhile tool for enhancing the capabilities of ECAD frameworks and systems. The method of reasoning of the system has undoubted limitations, but these are also faced by human experts when they attempt an heuristic understanding of electronic specifications. Further research can positively render more advanced ways of controlling the computational complexity of the reasoning method, but the philosophy of the system, which is based on the exploitation of data of an heuristic nature, makes the elimination of these limitations impossible for designs of arbitrary complexity. The current prototype of the system already indicates that correct results can be obtained and it shows methods for avoiding the critical computational complexity of the system which do not lead to the exclusion of attractive solutions. It is hoped that further work, which is still required for a complete implementation of the system proposed and for the development of applications based on the heuristic understanding of a specification, arrives at a conclusive empirical demonstration of the convenience and viability of the automatic heuristic understanding of logic electronic design specifications proposed in this work.

Bibliography

[AB85] M.S. Abadir and M.A. Breuer. A Knowledge-Based System for Designing Testable VLSI chips. *IEEE Design&Test*, 2(4):56–68, August 1985.

[AK86] J.S. Aude and H.J. Kahn. A design rule database system to support technology-adaptable applications. In *23rd Design Automation Conference*, pages 510–516, Las Vegas, Nevada, USA, June 1986.

[Bae80] J.L. Baer. *Computer Systems Architecture*. Computer Science Press, 1980.

[Ben84] M.J. Bending. Hitest: A knowledge-based test generation system. *IEEE Design&Test*, pages 83–92, May 1984.

[BK89] S. J. Bevan and H. J. Kahn. Objectively Parsing EDIF. In *Proceedings of the Third Europen EDIF Forum*, pages III–53–III–62, Bonn/Königswinter, Germany, October 1989.

[BM93] M. Brown and Z. Moosa. *A Heuristic Model of Design Cells*. Computer Science Department, University of Manchester, March 1993. Internal Report.

[BN71] C.G. Bell and A. Newell. *Computer Structures: Readings and Examples*. McGraw Hill, 1971.

[BS84] B.G. Buchanan and E.H. Shortliffe. *Rule-Based Expert Systems*. Addison-Wesley, 1984.

[Cla85] W.J. Clancey. Heuristic classification. *Artificial Intelligence*, 27(2):289–350, November 1985.

[CM85] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, 1985.

[CW88] M. Carlsson and J. Widen. *Sicstus Prolog User's Manual Version 0.6*. Swedish Institute of Computer Science, 1988.

[Dav84] R. Davis. Diagnostic reasoning based on structure and behaviour. *Artificial Intelligence*, 24(1-3):347–410, December 1984.

[Dav90] E. Davis. *Representations of Commonsense Knowledge*. Morgan Kaufmann Publishers,Inc., 1990.

[Doc88] Internal Document. *Manchester Simulation Engine (MANSE), Timewheel Board.* Computer Science Department, University of Manchester, 1988.

[EL85] M.D. Ercegovac and T. Lang. *Digital Systems and Hardware/Firmware Algorithms.* John Wiley&Sons, 1985.

[Ele87] Electronic Industries Association. *Electronic Design Interchange Format Version 2 0 0*, May 1987. EIA/ANSI Standard RS548.

[Fil88] N.P. Filer. *The Use of Knowledge Based Techniques for Electronic Computer Aided Design.* PhD thesis, Department of Computer Science, University of Manchester, January 1988.

[FK85] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, September 1985.

[FM89] N.P. Filer and R.A.J. Marshall. The design of a methodology for the intelligent control of simulation using the Manchester simulation engine. In *European Simulation Meeting*, pages 163–168, Rome, Italy, June 1989.

[FMW91] N.P. Filer, S. Mir, and D. Wray. Description of a prototype knowledge-based tool exploiting design semantics. Technical report, Jessi-CAD-Frame Deliverable D1.1, ESPRIT Special Project 5082, 1991.

[FS87] N. P. Filer and M. A. Spink. Knowledge Based Control For VLSI Layout. In *Proceedings of the IEEE International Workshop on AI-Applications to CAD-Systems for Electronics*, pages 119–136, Munich, Germany, October 1987.

[HB85] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985.

[Hec86] D. Heckerman. Probabilistic interpretations for Mycin's certainty factors. In L.N. Kanal and J.F. Lemmer, editors, *Uncertainty in Artificial Intelligence*, pages 167–196. Elsevier Science Publishers, 1986.

[HJ83] D.A. Hodges and H.G. Jackson. *Analysis and Design of Digital Integrated Circuits.* McGraw-Hill, 1983.

[HN88] M.T. Harandi and J.Q. Ning. PAT: A knowledge-based program analysis tool. In *IEEE Conference on Software Maintenance*, pages 312–318, Los Alamitos, California, October 1988.

[HNSB90] D.S. Harrison, A.R. Newton, R.L. Spickelmier, and T.J. Barnes. Electronic CAD frameworks. *Proc. of the IEEE*, 78(2):393–417, February 1990.

[HR85] F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, September 1985.

[IEE88] IEEE, New York, U.S.A. *IEEE Standard VHDL Language Reference Manual*, March 1988. IEEE standard No. 1076-1987.

[Int91] International Standard Organization STEP ISO/TC184/SC4/WG 5. *EXPRESS Language Reference Manual*, n14 edition, April 1991.

[Jac90] P. Jackson. *Introduction to Expert Systems*. Addison-Wesley, 1990.

[JK86] M.A. Jones and K.Baker. An intelligent knowledge-based system tool for high-level BIST design. In *IEEE International Test Conference*, pages 743–746, Philadelphia, Pennsylvania, 1986.

[JS89] M.A. Jabri and D.J. Skellern. PIAF: A knowledge-based/algorithmic top-down floorplanning system. In *26th Design Automation Conference*, pages 582–585, 1989.

[Kah85] H.J. Kahn. Environment for expert CAD software. *Silicon Design*, 2(9):17–18, September 1985.

[KF85] H. J. Kahn and N. P. Filer. An Application of Knowledge Based Techniques to VLSI Design. In Martin Merry, editor, *Expert Systems 85 – Proc. of the 5th Technical Conference of the BCS Specialist Group on Expert Systems*, pages 307–322. Cambridge University Press, 1985.

[KM87] H.J. Kahn and R.A.J. Marshall. Using EDIF with a hardware simulation engine. In *European EDIF Forum*, pages 2–20–2–25, Brussels, Belgium, 1987.

[KN89] W. Kozaczynski and J.Q. Ning. SRE: A knowledge-based environment for large-scale software re-engineering activities. In *11th International Conference on Software Engineering*, pages 113–122, Pittsburgh, Philadelphia, May 1989.

[KST82] D. Kahneman, P. Slovic, and A. Tversky. *Judgement under Uncertainty: Heuristics and Biases*. Cambridge University Press, 1982.

[KT72] D. Kahneman and A. Tversky. Subjective probability: a judgement of representativeness. *Cognitive Psychology*, 3:430–454, 1972.

[KTH88] K. Kim, J.G. Trout, and D.S. Ha. Automatic insertion of BIST hardware using VHDL. In *25th Design Automation Conference*, pages 9–15, Anaheim, California, 1988.

[Lai86] R. N. W. Laithwaite. An Expert System to Aid Placement of Gate Arrays. In *Proc. 3rd Silicon Design Conference*, Wembley, London, 1986.

[Lal85] P.K. Lala. *Fault Tolerant & Fault Testable Hardware Design*. Prentice-Hall, 1985.

[MC80] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[McC86] E.J. McCluskey. *Logic Design Principles*. Prentice-Hall, 1986.

[Mir89] S. Mir. *The Use of Knowledge Based Techniques for Simulation and Test of VLSI Digital Circuits*. Master's thesis, University of Manchester, October 1989.

[Mor90] T. Morgan. Five Years of Deep Knowledge. In 7^{th} *UK Deep Knowledge Based Systems Workshop*, Gregynog, Wales, United Kingdom, April 1990.

[Nix84] I.M. Nixon. *An Idiomatic Floorplanner*. University of Edinburgh, October 1984. Internal Report.

[Pen89] R. Penrose. *The Emperor's New Mind*. Oxford University Press, 1989.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, April 1987.

[RIXK91] A. Rubio, N. Itazaki, X. Xu, and K. Kinoshita. An approach to the analysis and test of crosstalk faults in digital VLSI circuits. In *The European Conference on Design Automation*, pages 72–79, Amsterdam, Holland, February 1991.

[RKCM85] G. Russell, D.J. Kinniment, E.G. Chester, and M.R. McLauchlan. *CAD for VLSI*. Van Nostrand Reinhold (UK), 1985.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Sha76] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

[Sho76] E.H. Shortliffe. *Computer-Based Medical Consultations: Mycin*. Elsevier, 1976.

[Sin87] N. Singh. *An Artificial Intelligence Approach to Test Generation*. Kluwer Academic Publishers, 1987.

[Spr90] M. Spreitzer. Comparing structurally different views of a VLSI design. In *27th Design Automation Conference*, pages 200–206, 1990.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.

[Sto90] H.S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, 1990.

[WE85] N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design. A Systems Perspective*. Addison-Wesley, 1985.

[Woo75] W.A. Woods. What's in a link: foundations for semantic nets. In D. Bobrow and A. Collins, editors, *Representation and Understanding*. Academic Press, 1975.

[WP82] T.W. Williams and K.P. Parker. Design for testability — A survey. *IEEE Transactions on Computers*, C-31(1):2–15, January 1982.

[YK90] T.C.O. Young and H.J. Kahn. A procedural interface to CAD data Based on EDIF. In *The European Design Automation Conference*, pages 496–500, Glasgow, Scotland, March 1990.

[Zad75] L.A. Zadeh. Fuzzy logic and approximate reasoning. *Synthese*, 30:407–428, 1975.

Appendix A

Decision Factors

This appendix estimates the effect that changes in the items of knowledge for the model of a cell can cause in the evaluation of the modelling of the overall design (an item of knowledge corresponds to a slot or a sub-slot of an heuristic model of a cell). Assuming a set of possible changes for some items of knowledge (for example, adding knowledge for the instantiation of undefined items, increasing the confidence in existing knowledge or changing knowledge about defined items) the changes which result in the highest values for the evaluation of the design are relatively more important. The effect of a change in an item of knowledge on the evaluation of the overall design is called the *decision factor* of that change. Decision factors can be used, for example, for automatically scheduling queries (to the users or other expert tools) for obtaining heuristic design knowledge about undefined items of knowledge which the system may have not been able to derive.

A.1 Decision Factor of a Change

A *change decision factor* calculates the rate of change of the evaluation of the overall design given a change in the k-th slot of the heuristic model M_i of the i-th cell in the design. For simplicity, the calculation considers that each cell C_i of a design with n cells has a model with an evaluation value of $e_i \geq 0$ (this is the case in the current implementation described in chapter 8). In this case, the evaluation of situation S_i which represents cell C_i and its contents is given by the evaluation function 6.12 as

$$E_i = 1 - (1 - e_i) \prod_{k=1}^{n_i} (1 - w_{i,k}) = 1 - U_i \tag{A.1}$$

where from equation 6.10 the weighted contribution of the sub-cells are calculated as

$$w_{i,k} = r_{i,k}\, E_{i,k} \tag{A.2}$$

The term $r_{i,k}$ corresponds to the weighting factor for the weighted contribution of sub-cell $C_{i,k}$ to the understanding of cell C_i. The term $E_{i,k}$ corresponds to the evaluation of the situation representing sub-cell $C_{i,k}$. The value E_1 for the evaluation of situation S_1 evaluates the overall design. Those individual changes which can result in the highest values for E_1 are relatively more important.

A change in the k-th slot of the heuristic model M_i of the i-th cell in the design results in a change in the value e_i which evaluates model M_i. A change in the value e_i results in a change in the value E_i which evaluates the i-th situation in the design. A change in the value E_i results in a change in the evaluation of all situations which make use of cell C_i. If v_k denotes the evaluation of the k-th slot of model M_i, the change decision factor which estimates the effect of a change in this slot is given by

$$\frac{\partial E_1}{\partial v_k} = \frac{\partial E_1}{\partial E_i}\frac{\partial E_i}{\partial e_i}\frac{\partial e_i}{\partial v_k} \tag{A.3}$$

In general, the rate of change of the value E_i which evaluates the i-th situation in the design when the value E_j which evaluates the j-th situation in a design changes is calculated by means of the chain rule for partial differentiation as

$$\frac{\partial E_i}{\partial E_j} = \sum_{k=1}^{n_i}\left(\frac{\partial E_i}{\partial E_{i,k}}\frac{\partial E_{i,k}}{\partial E_j}\right) \tag{A.4}$$

In this equation, the calculation of $\frac{\partial E_i}{\partial E_{i,k}}$ considers the value of the $E_{i,r}$ with $r \neq k$ constant when the value of $E_{i,k}$ changes. The k-th sub-cell $C_{i,k}$ of cell C_i corresponds to a cell C_q $(q \neq i)$ in the design, and the calculation of

$$\frac{\partial E_{i,k}}{\partial E_j} = \frac{\partial E_q}{\partial E_j}$$

is done again by means of the same equation A.4. The term

$$F_{i,k} = \frac{\partial E_i}{\partial E_{i,k}}$$

represents the *sensitivity* of the arc between cell C_i and its k-th sub-cell in the hierarchy graph to the change. Equation A.4 is written as

$$\frac{\partial E_i}{\partial E_j} = \sum_{k=1}^{n_i}\left(F_{i,k}\frac{\partial E_{i,k}}{\partial E_j}\right) \tag{A.5}$$

The *sensitivity* of a path in the hierarchy graph to the change is defined as the product of the sensitivities of the arcs which form the path. The differentiation $\frac{\partial E_i}{\partial E_j}$ is obtained from the sum of the sensitivities of all paths in the hierarchy graph which lead from cell C_i to cell C_j (this is easily derived from equation A.5). For example, in the hierarchy graph of figure A.1, the effect of changes in situation S_6 for the evaluation of the overall design is given by

$$\frac{\partial E_1}{\partial E_6} = F_{14}F_{46} + F_{12}\left(F_{23}F_{34}F_{46} + F_{26}\right)$$

which considers the three paths in the hierarchy graph which can be traced from cell C_1 to cell C_6 [1].

[1] As a matter of notation, it must be noted that in equation A.5, the arc decision factor $F_{i,k}$ denotes the arc between cell C_i and its k-th sub-cell. For convenience, in the example design the arc decision factors denote the arc between two cells: for example, F_{14} denotes the arc between cell C_1 and cell C_4 in the hierarchy graph.

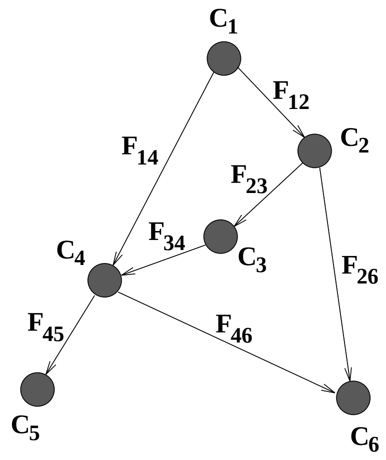


Figure A.1: An Example Design

A.2 Decision Factors for a Situation

This section calculates the sensitivity of an arc in the hierarchy graph to a change. The arc sensitivity $F_{i,k}$ determines the rate of change of the evaluation of situation S_i when the value $E_{i,k}$ which evaluates the situation representing its k-th sub-cell changes. As required in equation A.4, this calculation considers the rate of change of the evaluation of situation S_i when $E_{i,k}$ changes considering that the evaluation of the rest of sub-cell situations does not change. For the calculation, it is assumed that the changes do not affect the relative importance $r_{i,k}$ of sub-cell $C_{i,k}$ in the design of cell C_i. That is, the $r_{i,k}$ remain constant with the changes introduced. This assumption is relaxed in section A.4.

The arc sensitivity $F_{i,k}$ is calculated from equation A.1 as

$$F_{i,k} = \frac{\partial E_i}{\partial E_{i,k}} = -(1-e_i)\,\frac{\prod_{k=1}^{n_i}(1-w_{i,k})}{1-w_{i,k}}\,\frac{\partial(1-w_{i,k})}{\partial E_{i,k}}$$

From equation A.2

$$\frac{\partial(1-w_{i,k})}{\partial E_{i,k}} = -r_{i,k}$$

and, by considering equation 6.13, the arc sensitivity is given by

$$F_{i,k} = \frac{r_{i,k}}{1-w_{i,k}}\,U_i \tag{A.6}$$

It can easily be seen that

$$\frac{\partial^2 E_i}{\partial^2 E_{i,k}} = 0$$

and the evaluation of situation S_i grows linearly with the evaluation of the k-th sub-cell. The rate of change depends on the relative importance of the k-th sub-cell (as expressed by means of the weighting factor $r_{i,k}$) and the term $\frac{U_i}{1-w_{i,k}}$ which represents the uncertainty of the modelling of the cells involved in the situation (except the sub-cell considered). Equation A.6 can be written as

$$F_{i,k} = \frac{\partial E_i}{\partial E_{i,k}} = D_{i,k}\, U_i \tag{A.7}$$

with

$$D_{i,k} = \frac{r_{i,k}}{1 - w_{i,k}} \tag{A.8}$$

The increment (decrement) ΔE_i in the evaluation of situation S_i given an increment $\Delta E_{i,k}$ in the evaluation of the situation which represents its k-th sub-cell is calculated by integration of equation A.7 as

$$\Delta E_i = D_{i,k}\, U_i\, \Delta E_{i,k}$$

In the case that the $\Delta E_{i,k}$ are known for several possible changes in the evaluation of several sub-cell situations, this equation determines which change results in the highest value for E_i. In the case that the $\Delta E_{i,k}$ cannot be calculated (e.g. the system only estimates which queries about undefined items of knowledge are to be scheduled first) the $D_{i,k}$ factors determine which changes in the situation more rapidly affect the value of E_i (the value of U_i is the same for all changes). Because of this, the $D_{i,k}$ are called *situation decision factors.*

A.3 Decision Factors for a Cell Heuristic Model

A change in the k-th slot of the heuristic model M_i results in a change for the evaluation of the model e_i and, therefore, in a change in the value E_i which evaluates situation S_i. This section calculates these changes. For simplicity, it is possible to assume that each slot in the heuristic model M_i has an evaluation value of $v_k \geq 0$ (this is the case in the current implementation described in chapter 8). In this case, the value e_i which evaluates the model can be calculated by means of the evaluation function 6.5 as

$$e_i = 1 - \prod_{k=1}^{7} (1 - w_k) \qquad w_k > 0$$

where

$$w_k = r_k\, v_k$$

with r_k being the weighting factor (relative importance) of the k-th slot in the heuristic model (a model has seven slots in the current implementation). A change in the k-th slot of the heuristic model M_i which represents cell C_i results in a change for the evaluation of the model given by

$$\frac{\partial e_i}{\partial v_k} = \frac{r_k}{1 - w_k}\, u_i \tag{A.9}$$

where

$$u_i = \prod_{k=1}^{7} (1 - w_k)$$

represents the level of uncertainty in the heuristic model M_i. Equation A.9 can be written as

$$\frac{\partial e_i}{\partial v_k} = d_k\, u_i \tag{A.10}$$

with

$$d_k = \frac{r_k}{1 - w_k} \tag{A.11}$$

The d_k factors determine which changes in the model more rapidly affect the value of e_i (the value of u_i is the same for all changes). For this reason, the d_k are called *slot decision factors* [2].

The rate of change of the evaluation of situation S_i when the evaluation of the model of C_i changes is calculated from equation A.1 as

$$\frac{\partial E_i}{\partial e_i} = \frac{U_i}{1 - e_i} \tag{A.12}$$

It is easy to observe that

$$\frac{\partial^2 E_i}{\partial^2 e_i} = 0$$

and the evaluation of situation S_i grows linearly with the evaluation of the model of cell C_i. Equation A.12 implies that the rate of change only depends on the uncertainty associated with the modelling of the sub-cells. This equation can be written as

$$\frac{\partial E_i}{\partial e_i} = D_i\, U_i \tag{A.13}$$

with

$$D_i = \frac{1}{1 - e_i} \tag{A.14}$$

The factors D_i are called *model decision factors*. Comparing equation A.13 with equation A.7, it is clear that the model and situation decision factors determine which changes can more rapidly affect the evaluation of the whole situation.

The effect of a change in the k-th slot of the heuristic model M_i upon the evaluation of the overall design is obtained as the product of the corresponding slot decision factor, model decision factor and the sum of the sensitivities of the paths which lead to cell C_i as indicated in equation A.3. As an example, a change in the fourth slot of the model of cell C_6 in the design of figure A.1 results in a rate of change of the evaluation of the overall design given by

$$\frac{\partial E_1}{\partial v_4} = [F_{14}F_{46} + F_{12}\,(F_{23}F_{34}F_{46} + F_{26})]\, D_6 U_6 d_4 u_6$$

where v_4 and d_4 refer to the fourth slot of the model of cell C_6.

[2] Decision factors for the sub-slots of a slot can be calculated in the same way since the evaluation of a compound slot is obtained from the combination of the evaluation of its sub-slots.

A.4 Changes of Estimated Complexity

The relative importance and weighting factor of a sub-cell in the design of a cell is calculated from estimates of the complexities of the cells involved in the design of the cell. In the above analysis, the weighting factor $r_{i,k}$ of a sub-cell $C_{i,k}$ which is used in the design of cell C_i has been assumed to be constant for all changes in the models of the cells. However, the estimated complexities of the cells depend on heuristic knowledge, and changes of the knowledge in the models may trigger changes for the estimated complexities. Therefore, when the knowledge about a cell or a sub-cell of a situation changes, the weighting factors of the sub-cells may change. This section takes into account these changes.

If $CO^*_{i,j}$ is the estimated complexity of the j-th sub-cell of cell C_i, a change in the value E_i which evaluates situation S_i given a change in the complexity of sub-cell $C_{i,j}$ is calculated by differentiation of equation A.1 as

$$\frac{\partial E_i}{\partial CO^*_{i,j}} = -\frac{\partial U_i}{\partial CO^*_{i,j}} = -\frac{\partial[(1-e_i)\prod_{k=1}^{n_i}(1-w_{i,k})]}{\partial CO^*_{i,j}}$$

which gives

$$\frac{\partial E_i}{\partial CO^*_{i,j}} = -U_i \sum_{k=1}^{n_i}\left(\frac{1}{1-w_{i,k}}\,\frac{\partial(1-w_{i,k})}{\partial CO^*_{i,j}}\right) = U_i \sum_{k=1}^{n_i}\left(\frac{E_{i,k}}{1-w_{i,k}}\,\frac{\partial r_{i,k}}{\partial CO^*_{i,j}}\right) \tag{A.15}$$

The weighting factor $r_{i,k}$ is given by equation 6.28 as

$$r_{i,k} = \frac{R^*_{i,k}}{R^*_{i,k} + |1-r_i|} \tag{A.16}$$

where $R^*_{i,k}$ is the relative importance of sub-cell $C_{i,k}$ in the design of cell C_i and r_i is the complexity deviation factor for cell C_i. The differentiation of this equation with respect to $CO^*_{i,j}$ gives

$$\frac{\partial r_{i,k}}{\partial CO^*_{i,j}} = \frac{r^2_{i,k}}{R^{*2}_{i,k}}\left(|1-r_i|\,\frac{\partial R^*_{i,k}}{\partial CO^*_{i,j}} - R^*_{i,k}\,\frac{\partial|1-r_i|}{\partial CO^*_{i,j}}\right) \tag{A.17}$$

The relative importance $R^*_{i,k}$ is given by equation 6.20 as

$$R^*_{i,k} = I_{i,k}\,\frac{CO^*_{i,k}}{CO^c_i} \tag{A.18}$$

where $I_{i,k}$ is the number of instances of sub-cell $C_{i,k}$ in the design of cell C_i and CO^c_i corresponds to the complexity of C_i which is calculated by means of the estimated complexities of the sub-cells according to equation 6.19 as

$$CO^c_i = \sum_{k=1}^{n_i} I_{i,k}\,CO^*_{i,k}$$

The differentiation of equation A.18 with respect to $CO^*_{i,j}$ gives

$$\frac{\partial R^*_{i,k}}{\partial CO^*_{i,j}} = R^*_{i,k} \left(\frac{1}{CO^*_{i,k}} \frac{\partial CO^*_{i,k}}{\partial CO^*_{i,j}} - \frac{R^*_{i,j}}{CO^*_{i,j}} \right) \tag{A.19}$$

The complexity deviation factor r_i is defined in equation 6.27 as the quotient

$$r_i = \frac{CO^*_i}{CO^c_i} \tag{A.20}$$

and the differentiation of this equation with respect to $CO^*_{i,j}$ gives

$$\frac{\partial r_i}{\partial CO^*_{i,j}} = -I_{i,j} \frac{CO^*_i}{CO^{c2}_i}$$

and

$$\frac{\partial |1 - r_i|}{\partial CO^*_{i,j}} = \begin{cases} \frac{\partial r_i}{\partial CO^*_{i,j}} & r_i < 1 \\ 0 & r_i = 0 \\ -\frac{\partial r_i}{\partial CO^*_{i,j}} & r_i > 1 \end{cases} \tag{A.21}$$

The substitution of equation A.19 and equation A.21 in equation A.17 gives

$$\frac{\partial r_{i,k}}{\partial CO^*_{i,j}} = \frac{r^2_{i,k}}{R^*_{i,k}} \left(\frac{|1 - r_i|}{CO^*_{i,k}} \frac{\partial CO^*_{i,k}}{\partial CO^*_{i,j}} - K \frac{R^*_{i,j}}{CO^*_{i,j}} \right)$$

with

$$\left.\begin{array}{ll} r_i < 1 & K = 1 \\ r_i = 0 & K = 0 \\ r_i > 1 & K = -1 \end{array}\right\}$$

and the substitution of this equation in equation A.15 gives

$$\frac{\partial E_i}{\partial CO^*_{i,j}} = \frac{U_i}{CO^*_{i,j}} \left[\frac{E_{i,j}}{1 - w_{i,j}} \frac{r^2_{i,j}}{R^*_{i,j}} |1 - r_i| - K\, R^*_{i,j} \sum_{k=1}^{n_i} \left(\frac{E_{i,k}}{1 - w_{i,k}} \frac{r^2_{i,k}}{R^*_{i,k}} \right) \right] \tag{A.22}$$

Considering now that the estimated complexity CO^*_i of cell C_i changes, the rate of change of E_i which evaluates situation S_i is calculated by differentiation of equation A.1 as

$$\frac{\partial E_i}{\partial CO^*_i} = U_i \sum_{k=1}^{n_i} \left(\frac{E_{i,k}}{1 - w_{i,k}} \frac{\partial r_{i,k}}{\partial CO^*_i} \right) \tag{A.23}$$

The differentiation of equation A.16 with respect to CO^*_i gives

$$\frac{\partial r_{i,k}}{\partial CO^*_i} = -\frac{r^2_{i,k}}{R^{*2}_{i,k}} \frac{\partial |1 - r_i|}{\partial CO^*_i}$$

and the differentiation of equation A.20 with respect to CO^*_i gives

$$\frac{\partial r_i}{\partial CO^*_i} = \frac{1}{CO^c_i}$$

and, by considering equation A.21 (differentiating with respect to CO^*_i in this case), equation A.23 becomes

$$\frac{\partial E_i}{\partial CO^*_i} = \frac{K\,U_i}{CO^c_i} \sum_{k=1}^{n_i} \left(\frac{E_{i,k}}{1 - w_{i,k}} \frac{r^2_{i,k}}{R^*_{i,k}} \right) \tag{A.24}$$

Given a change in the k-th slot of the heuristic model of cell C_i which also affects its complexity, the change in the evaluation of situation S_i is given by

$$dE_i = \frac{\partial E_i}{\partial e_i} \frac{\partial e_i}{\partial v_k} dv_k + \frac{\partial E_i}{\partial CO^*_i} dCO^*_i \tag{A.25}$$

where $\frac{\partial E_i}{\partial e_i}$ is given by equation A.13, $\frac{\partial e_i}{\partial v_k}$ is given by equation A.10 and $\frac{\partial E_i}{\partial CO^*_i}$ is given by equation A.24. Similarly, given a change in the heuristic model of sub-cell $C_{i,j}$ which also affects its estimated complexity, the change in the evaluation of situation S_i is given by

$$dE_i = \frac{\partial E_i}{\partial E_{i,j}} dE_{i,j} + \frac{\partial E_i}{\partial CO^*_{i,j}} dCO^*_{i,j} \tag{A.26}$$

where $\frac{\partial E_i}{\partial E_{i,j}}$ is given by equation A.7 and $\frac{\partial E_i}{\partial CO^*_{i,j}}$ is given by equation A.22.

The complexity of a cell is estimated from heuristic knowledge about the cell by means of a complexity estimation function. The complexity estimation function discussed in section 6.6 only considers the level of abstraction EH_i for evaluating the complexity of cell C_i. According to equation 6.33 and equation 6.32

$$\begin{aligned} CO^*_i &= HR^{EH_i} \\ CO^*_{i,j} &= HR^{EH_{i,j}} \end{aligned}$$

where HR is the typical interlevel complexity ratio. Considering this complexity estimation function, a change in the evaluation of a situation depends on a change of the evaluation value for an item of knowledge and on a change on the level of abstraction. That is, given a change in the level of abstraction of cell C_i (which also affects the confidence in it)

$$dE_i = \frac{\partial E_i}{\partial e_i} \frac{\partial e_i}{\partial v_k} dv_k + \frac{\partial E_i}{\partial CO^*_i} \frac{\partial CO^*_i}{\partial EH_i} dEH_i \tag{A.27}$$

where

$$\frac{\partial CO^*_i}{\partial EH_i} = HR^{EH_i} \ln HR = CO^*_i \ln HR$$

and given a change in the level of abstraction in the model of sub-cell $C_{i,j}$

$$dE_i = \frac{\partial E_i}{\partial E_{i,j}} dE_{i,j} + \frac{\partial E_i}{\partial CO^*_{i,j}} \frac{\partial CO^*_{i,j}}{\partial EH_{i,j}} dEH_{i,j} \tag{A.28}$$

where

$$\frac{\partial CO^*_{i,j}}{\partial EH_{i,j}} = CO^*_{i,j} \ln HR$$

Appendix B

Semantic Networks for Name Analysis

A key technique for the identification of the electronic functionality of cells and signals is based on the examination of the names associated with the objects in a design. Although the name given to an object is arbitrary, the use of meaningful names is usual practice to produce accurate design descriptions. This appendix describes a way of representing knowledge about names, and automating the comparison of names, based on the use of semantic networks. Semantic networks provide a natural way of jointly storing information about object names and electronic functionality values. Name semantic networks are used to provide guidance for the matching with heuristic models as discussed in section 5.5.1. The main uses of the comparison of names in the current implementation include:

1. a name is compared with typical values for the electronic functionality of the type of objects to which the name refers to (cells, ports and signals).

2. a name is compared with electronic functionality values and names generated from the processing of previous designs which are represented in the semantic networks of the system. Semantic networks provide an automatic way of generating possible electronic functionality values and storing information about new names.

3. names are compared between them, regardless of the actual meaning of these names. This is useful since objects with similar names (mostly ports and signals) usually have similar functionalities.

B.1 Meanings of a Word

Names are usually composed of a number of basic entities or *words*. A word in a name is formed by sequences of letters or sequences of digits. Words are not case sensitive (a word in lower case or upper case generally carries the same semantic value). Words in a name are usually separated by delimiter characters (e.g. _ - / \ | .). A name is decomposed into words by looking for sequences of letters, sequences of digits and delimiter characters. For example, the name 'ctnand4' is decomposed into two entities 'ctnand' and '4' and the name 'shift_register_latch' is decomposed into the words 'shift', 'register' and 'latch'. Semantic networks are classified according to the type of design objects to which the names represented apply. Two types of networks are used in the current implementation: networks which apply to names given to the cells of a design and networks which apply to names given to ports and signals. Table B.1 illustrates some examples of names given to cells and signals in the electronic designs analysed in chapter 8.

Words form the atomic units for the representation of names with semantic networks. The system associates an evaluation (confidence) value with each word in a network. When forming the words of a name, it may be necessary to throw away odd characters. In the case that some characters are discarded during the formation of the words of a name, the confidence values of the words which are next to the characters are initially decreased (considering the ratio between the number of characters ignored and the number of characters in the word). The confidence in a word is calculated as a result of the matching with an existing semantic network or as a result of forming a new network.

Cell Names	**Port Names**
ctnand4	data_input
shift_register_latch	read/write
jkff	scan_data_in
2_input_mux	clock
ROM	CK
read_only_memory	test_port
inverter	ctrl
multiplexer	SDI
positive_edge_triggered_flip_flop	1D
half_adder	scan_data_output
and_or_invert	CTRDIN_256
SN7400	VME_ADR
D_flipflop	P_greater_Q
bilbo_register	cwheel_sel_data_out

Table B.1: Examples of Object Names

Figure B.1 illustrates some examples of semantic networks. Figure B.1(a) is a network for the word 'latch' and figure B.1(b) is a network for the word 'setresetlatch'. These are words of kind 'class' since they have no links to other words. A class word represents an electronic functionality value and it has a semantic network of words associated with it which represents names which refer to this electronic functionality. Any word in a semantic network has only one meaning and one link to another word according to this meaning. A word can have links coming from a number of words. The possible meanings of a word are classified as follows:

1. *class word*: this word corresponds to a typical value for the electronic functionality of a cell or a port (signal). A class word has no links to other words.

2. *mnemonic word*: this word is used to mean another word. It is linked in a network

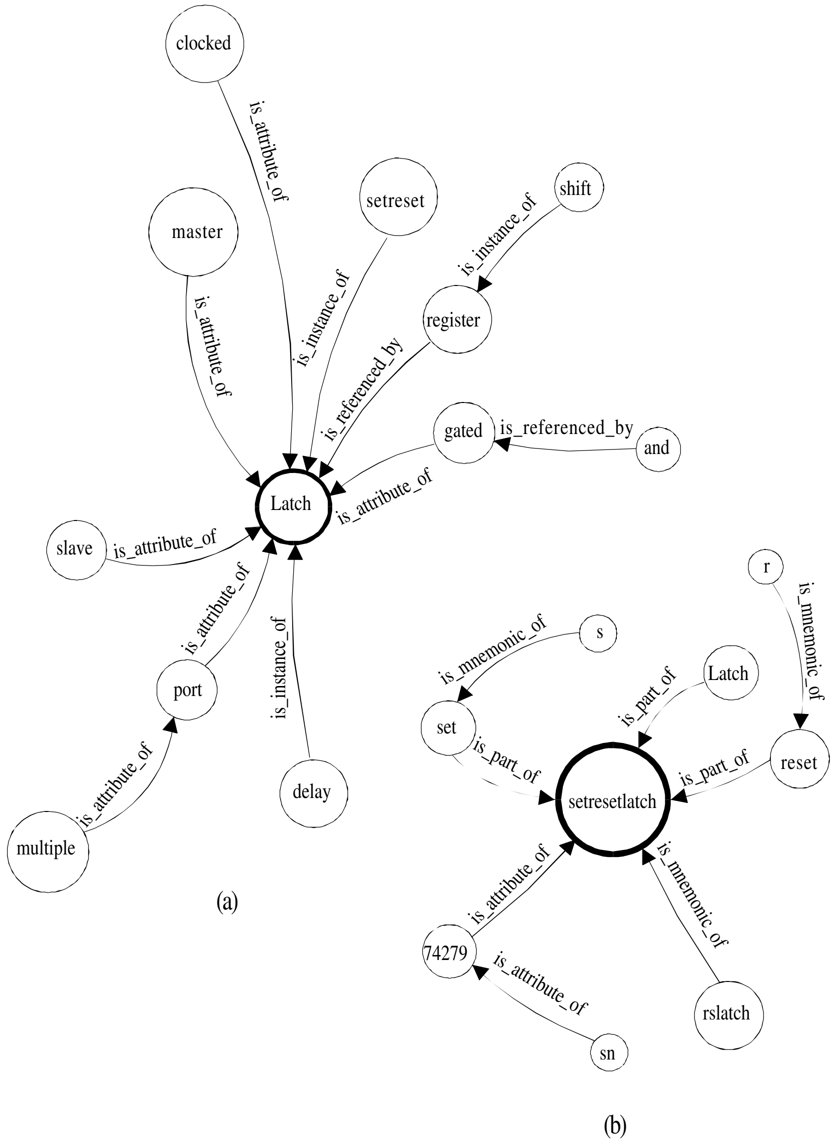


Figure B.1: Semantic Networks: a) class latch, and b) class set-reset latch.

using the link *is_mnemonic_of*. For example, 's' is often used to mean the word 'set' and 'ff' usually means the word 'flipflop'.

3. *part_of word*: this word is a part of another word. It is linked with an *is_part_of* link. For instance, the word 'flip' is a part of a word 'flipflop' and a word 'set' is a part of a word 'setresetlatch'.

4. *reference word*: this word is used in a network to reference another class and it is related in a network by means of an *is_referenced_by* link. For instance, the word 'register' in the semantic network of the class word 'latch' allows the use of the semantic network of the class 'word' register when matching names with the network of the class word 'latch'.

5. *part_reference word*: this word is a part of another word and it references a class word. It is linked with a link *is_part_of*. For example, the word 'latch' in the network of the class word 'setresetlatch'. The referenced network can be used when matching names with a network.

6. *attribute word*: this word is used to qualify another word. It is linked in a network with a link *is_attribute_of*. For instance, the word 'master' in the network of class word 'latch'.

7. *instance word*: this word indicates that names which can be formed in the network using an instance word are represented in another network (called a sub-class network). The word is linked with an *is_instance_of* link. For example, the word 'setreset' in the network of the class word 'latch' is an instance word. The network with class word 'setresetlatch' is a sub-class network of the network with class word 'latch' (the electronic functionality value 'setresetlatch' is a sub-class of the class of electronic functionality values 'latch'). Operationally, if the matching with a network makes use of an instance word, the matching is overlooked since the matching with a sub-class network must be considered.

A confidence value is associated with each word in a network which evaluates the belief that the meaning of the word in the network is correct. For example, the word 'clk' is used as a mnemonic for the word 'clock' and both words are related by means of an *'is_mnemonic_of'* link. The confidence in the word 'clk' being a mnemonic of word 'clock' (which is in fact the confidence in the link) is the confidence associated with the word 'clk'.

B.2 Meaning of a Semantic Network

A word can have only one link to another word, but it can have links coming from a number of words. The valid links to each kind of word are defined by considering the valid *sub-networks of a word.* A sub-network is defined as follows:

> *"a sub-network of a word* **W1** *is a sub-set of a semantic network of words which includes a link from another word* **W2** *to word* **W1** *and all sub-networks of word* **W2***".*

A semantic network is then defined as follows:

> *"the semantic network of a class word* **W** *includes the word* **W** *and all the sub-networks of this word".*

The different kinds of sub-networks are called mnemonic, part_of, reference, attribute and instance sub-networks. For instance, the class word 'latch' in figure B.1(a) has two instance sub-networks, one reference sub-network and six attribute sub-networks. Class, part_of and attribute words can have any kind of sub-networks. Mnemonic and instance words can not have any sub-networks (they are not further linked in a network). Reference and part_reference words can have attribute, instance, and reference sub-networks. Since reference and part_reference words refer to a class word, all sub-networks of the referenced class word are meaningful in the network of the referencing word (they are also sub-networks of the referencing word).

The meaning of a semantic network of words is then defined as follows:

> *"the meaning of a semantic network of words corresponds to all the names which can be formed with the words linked in the semantic network. Names are formed by following the relationships in the network from a word towards a class word. Instance words cannot be used. A mnemonic word can be used to substitute the word it applies to. All or some of the parts of a word can be used in place of this word. A referenced semantic network can be used in place of the referencing word".*

For example, a number of names can be formed in the networks of figure B.1 by following the links towards the class word. The names 'setreset_latch', 'master_latch' or 'and_gated_latch' are possible names for the class 'latch'. Mnemonic words can be used in place of the words they apply to. Thus, for class 'setresetlatch', the names 'rslatch', 'sr_latch' or 'rs_latch' are possible names to mean this class. Because the sub-networks of referencing words are also meaningful in the network where they appear, names like 'sr_gated_latch' can be formed.

B.3 Control of Name Matching

The matching of the words of a name with a semantic network implies that the name can be derived from the network (the name is included in the *meaning* of the semantic network). The matching must first identify the class word of the name. For this, some of the words of a name must match with the class word of the network and the matching will proceed with the rest of words in the name. The rest of words either match the sub-networks of the class word or the semantic network is modified to include the words of the matching name.

A value in the range $[-1, 1]$ evaluates the matching of a name which is obtained by combining the evaluation of the matching of each word and the confidence in the meaning of each word in the network using the formulae described in section 6.2. A successful matching of a name must have a positive evaluation value in the current implementation. That is, positive evidence that the name suggests the electronic functionality value represented by the class word is required.

Heuristic rules are used for the matching of a name. These rules, which govern the flow of control and the creation of new words, are:

1. *mnemonisation*: a word of a name has not been matched in a network, but it is a close word of an existing word in the network. The evaluation of the word being a close word or mnemonic of an existing one is done by means of a function which calculates the lexical distance between words. The words in the semantic networks are arranged in dictionaries (see [Mir89]). Separate dictionaries exist for the words

included in the networks of cell names and port (signal) names. If perfect matches for a word of a name do not exist in the networks, a set of the closest words (if any) to a word can be obtained from a dictionary. A word from this set is then used for the matching. This rule can create mnemonic words. The confidence in a new mnemonic word is calculated from the lexical distance between the words.

2. *decomposition*: a word is not matched in a network but it is a sub-word of an existing one (it is contained within another word). The existing word is seen as a compound word which can be split into two or three parts (depending if the sub-word is in the middle or at an end of the compound word). The splitting of the compound word can take place if at least one of the other component words are meaningful (they match existing words or they are close to existing words). Similarly, if a word of a network is a sub-word of a word of the name being matched, this last word is also decomposed into shorter words. This rule can create part_of and part_reference words.

3. *referencing*: the network of a referenced word is used for matching a name with a network which includes a reference word (the sub-networks of the reference word in the current network are investigated first and the sub-networks of the referenced class word afterwards).

4. *importation*: a word and a sub-network of it can be matched in a network which is different to the network currently investigated. The sub-network can be copied to the current network.

5. *generalisation*: some words of a name can be ignored since they do not provide any additional meaning to the name (for example, integers which are added to the end of a name two make it a unique identifier such as in the case of 'top_free_list_1'). A new name is obtained which is seen as a generalisation of the old name.

6. *classification*: new class words can be created if there is no successful matching with the existing networks. This rule can create class, part_reference and reference words. The creation of a new class word may result in the transformation of words in other networks into reference and part_reference words.

7. *instantiation*: a name matches two networks and the class word of a network is assumed to be a sub-class of the class word of another network. An instance word must be added to the class network which corresponds to a part_of word of the class word of the sub-class network.

8. *descending*: the matching of a branch of a semantic network cannot continue. The matching may proceed by following a different branch (the matching with the same network may continue from a different point).

9. *ascending*: the matching of a branch of a semantic network can proceed by jumping over words in the branch. This rule is applied before the rule for descending. The

rule of ascending results in a higher value for the evaluation of the matching of a name than the rule of descending since the same branch in the network is still considered.

10. *extension*: this rule applies when further matches in the current network are not possible and new words must be added to it. This rule can add new attribute and reference words.

Appendix C

Combination of Evaluation-function Values

This appendix describes the system of equations 6.3. These equations allow to update the evaluation of an hypothesis when more knowledge to support the hypothesis is presented. These equations are

$$e_{k+1} = (1 - w_{k+1})\, e_k + w_{k+1} \qquad e_k, w_{k+1} > 0 \tag{C.1}$$

$$e_{k+1} = (1 + w_{k+1})\, e_k + w_{k+1} \qquad e_k, w_{k+1} < 0 \tag{C.2}$$

$$e_{k+1} = \frac{e_k + w_{k+1}}{1 - min\{|e_k|, |w_{k+1}|\}} \qquad otherwise \tag{C.3}$$

with

$$w_{k+1} = r_{k+1}\, v_{k+1}$$

The current value for the evaluation of the hypothesis is e_k. The value v_{k+1} represents the evaluation of a new item of knowledge which supports the hypothesis and r_{k+1} represents the relative importance of this $(k+1)$-th item of knowledge with respect to the hypothesis. The value w_{k+1} is the weighted contribution of the $(k+1)$-th item of knowledge to the evaluation of the hypothesis investigated. The value e_k is combined with the value w_{k+1} according to the above equations to determine the new value e_{k+1} for the evaluation of the hypothesis. Figure C.1 describes these equations for different values of e_k. The figure is interpreted as follows:

1. e_k and w_{k+1} are both positive: in this case e_k linearly rises towards 1 by an amount that depends on w_{k+1} as represented by equation C.1. This can be seen in the first quadrant of figure C.1 for the cases in which e_k is positive. For example, for the case $e_k = 0.9$ if w_{k+1} had a value of 0 then e_{k+1} would have the same value as e_k of 0.9. If w_{k+1} had a value of 1 e_{k+1} would reach the maximum value of 1. Generally, this cannot happen since the relative importance of the $(k+1)$-th item of knowledge r_{k+1} will be lower than 1 and so will be the value w_{k+1}.

2. e_k and w_{k+1} are both negative: in this case e_k linearly slides towards -1 by an amount that depends on w_{k+1} as represented by equation C.2. This can be seen in the third quadrant of figure C.1 for the cases in which e_k is negative. The analysis is similar as in the previous case.

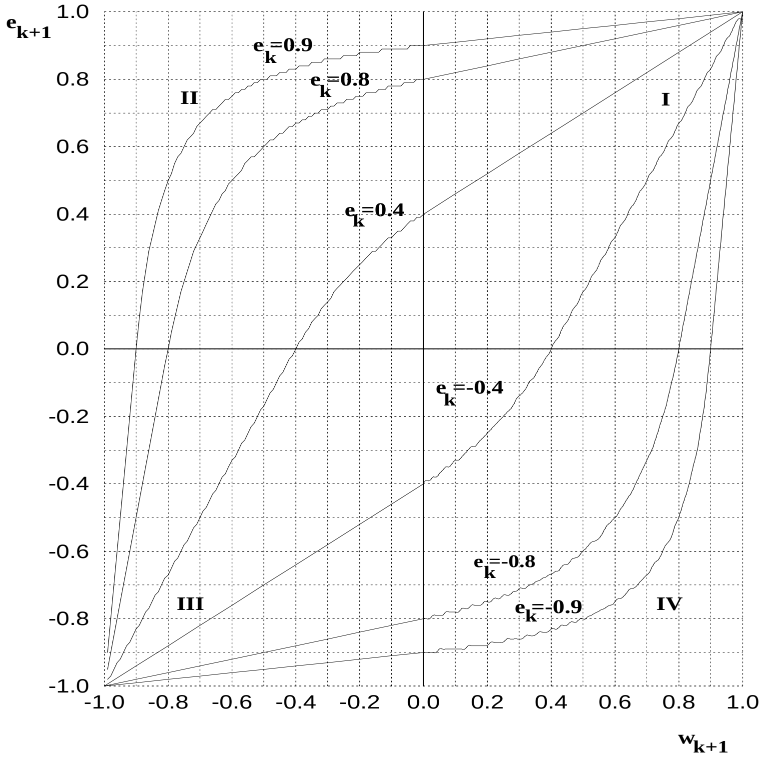


Figure C.1: Combination of Two Evaluation-function Values

3. equation C.3 represents all the other cases. The values e_k and w_{k+1} are of a different sign. If e_k is positive and w_{k+1} is negative the value of e_{k+1} must be lower than e_k (this can be seen in the second and third quadrants for the cases in which e_k is positive). If e_k is negative and w_{k+1} is positive the value of e_{k+1} must be higher than e_k (see the first and fourth quadrants for the cases in which e_k is negative). If e_k and w_{k+1} have the same absolute value e_{k+1} is zero. If w_{k+1} has a higher absolute value than e_k the value e_{k+1} represents a linear increase/decrease towards 1/-1 with respect to 0 which depends on w_{k+1} (see the first quadrant for the cases in which e_k is negative and the third quadrant for the cases in which e_k is positive). If w_{k+1} has a lower absolute value than e_k the value e_{k+1} tends to zero when the absolute value of w_{k+1} increases (see the second and fourth quadrants). This tendency is more significant for absolute values of w_{k+1} that are closer to the absolute value of e_k. Absolute values of w_{k+1} which are low with respect to the absolute value of e_k find it difficult to alter the existing evidence which is of a different sign. But if the absolute value of w_{k+1} is high with respect to the absolute value of e_k the resulting evaluation tends rapidly to zero.

Appendix D

Architecture of Computer Systems

The organisation of computer systems can be divided into two groups: those which are built around a single processor (uniprocessor systems) usually for conventional applications and those which include special techniques for high-performance computer applications.

D.1 Uniprocessor Systems

A typical uniprocessor system contains three major components: the main memory, the central-processor-unit (CPU) and the input-output sub-system. The organisation is shown in figure D.1. The data sub-system and the control sub-system form the processor (address values can be calculated in the data sub-system and passed to the control unit, although special operators for this purpose are usually provided in the control unit).

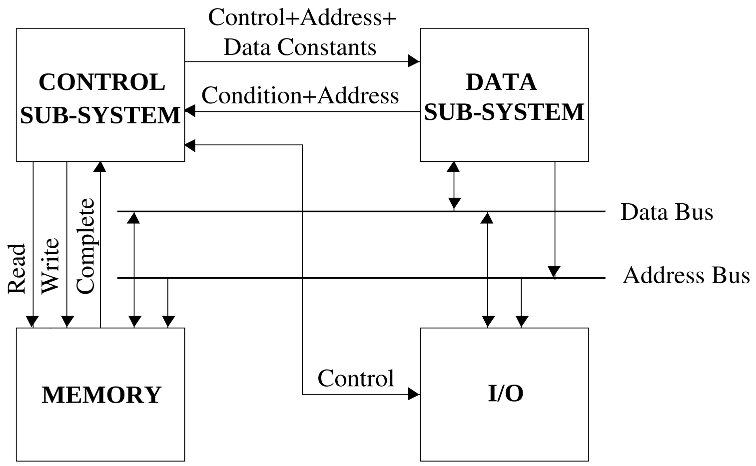


Figure D.1: Structure of a Uniprocessor System

The memory stores the data and the instructions to execute. The architecture must provide for concurrent demands on memory for data to process, instructions to execute and input/output transfer between memory and the environment. The operation of the system consists of periods of algorithm execution separated by transfers of blocks of data between levels of the memory hierarchy. For example, blocks of memory words are moved from the main memory into the cache so that immediate instructions/data can be available most of the time from the cache which operates as a data/instruction buffer.

The input-output sub-system connects the processor with the environment to obtain data and commands and to return the results of the computation. The interface of the I/O devices with the processor takes place by means of *device controllers* or *channels* as shown in figure D.2. The devices usually operate asynchronously with respect to the processor. The controllers generate timing and control signals for the communication with the processor, perform data format conversions if required, and detect and correct data transmission errors. A direct-memory-access (DMA) channel is usually used to provide direct information transfer between the I/O devices and the main memory.

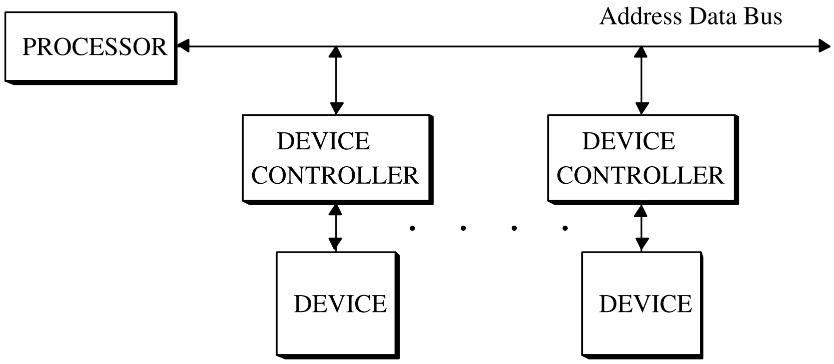


Figure D.2: Organisation of the I/O System

The interface processor/controller is viewed as a set of registers that are read and written by the processor and by the controller. These registers can be data registers (used to transfer the data), control registers (used to send control information from the processor to the device controller), and status registers used to send status information from the controller to the processor. With this organisation, the processors perform the I/O function just by reading and writing into the registers.

D.2 High-Performance Computer Systems

Advanced computer architectures are centered around the concept of parallel processing. Parallel processing was initiated by promoting concurrency in uniprocessor systems. A number of parallel processing mechanisms have been developed in uniprocessor computers including multiplicity of functional units, parallelism and pipelining within the CPU, overlapped CPU and I/O operations, use of a hierarchical memory system, multiprogramming and time-sharing [HB85].

Computer performance can be further upgraded by using a set of interactive processors (since semiconductor technology limits the speed of any single processor) [1]. Parallel

[1] In some critical applications, the main purpose for using several processors is for reliability rather

processing machines or *parallel computers* can be divided into three architectural configurations: *pipeline computers*, *array processors* and *multiprocessor systems*. Most existing computers are now pipelined, and some of them assume an array or a multiprocessor structure.

D.2.1 Pipeline computers

In a non-pipelined computer, the different steps of the instruction cycle (typically instruction fetch (IF), instruction decoding (ID), operand fetch (OF) and execution (EX)) must be completed before the next instruction can be issued. A pipelined computer has different stages for every step of the instruction cycle which are connected in cascade as shown in figure D.3. High performance is achieved by placing the several stages of the pipeline in operation simultaneously. Successive instructions are executed in an overlapped fashion. The flow of data (input operands, intermediate results and output results) is triggered by a common clock of the pipeline. Interface latches are used between adjacent segments to hold the intermediate results. In the example of figure D.3, it takes four pipeline cycles to complete one instruction, but an output value can be produced from the pipeline on each cycle once the pipeline is filled up. Some of the stages of the pipeline can be further partitioned. For example, the execution stage can be partitioned into a multi-stage arithmetic-logic pipeline. This is the case for sophisticated floating-point operations.

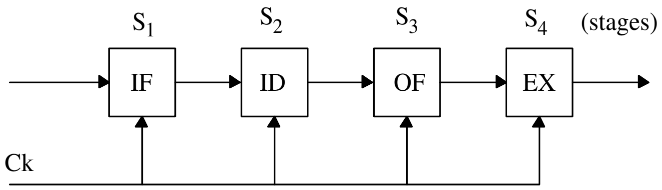


Figure D.3: Pipelined Processor

D.2.2 Array Computers

An *array processor* is a synchronous parallel computer with multiple processing elements (PEs) that operate in parallel under the control of a single processor. A typical array processor is shown in figure D.4. Scalar and control type instructions are directly executed in the control unit. The PEs execute a single stream of vector instructions broadcast to them by a single processor. Instruction fetch (from local memories or from the control memory) and decode is done by the control unit. The PEs are passive devices without instruction decoding capabilities. Each PE consists of a data processing unit and a local memory. The PEs are synchronised to perform the same instruction over a different operand fetched directly from the local memories. An appropriate data-routing mechanism must be established among the PEs. The interconnection pattern for a given application is under program control from the control unit.

than high performance. This is based on the idea that if any single processor fails, its task can be

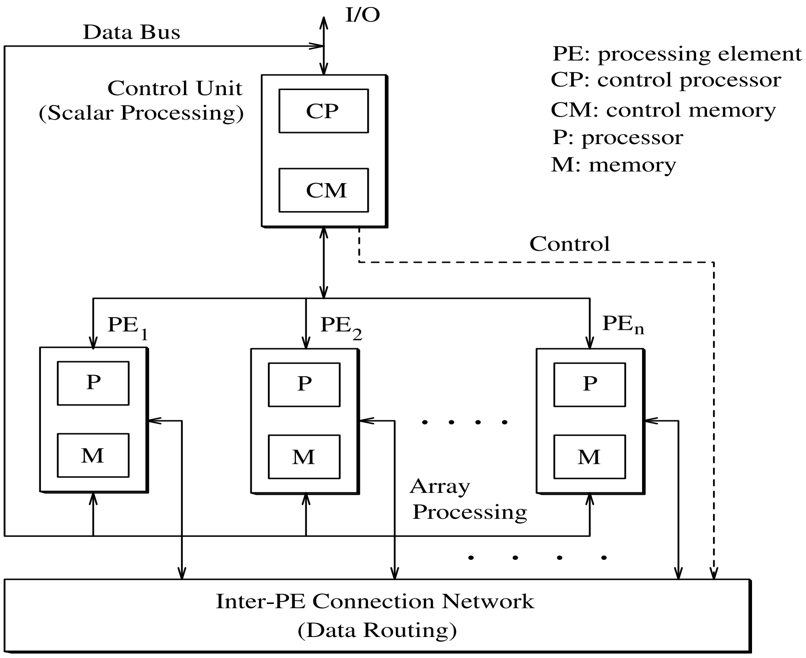


Figure D.4: Functional Structure of a Conventional Array Processor

D.2.3 Multiprocessor Systems

A multiprocessor is a computer system composed of several independent processors, each capable of executing its own program. The parallelism is achieved asynchronously through the set of interactive processors that share access to common sets of memory cells, I/O channels and peripheral devices. Interprocessor communications can be done through the shared memories or through an interrupt network [Sto90].

The organisation of a multiprocessor system is determined basically by the interconnection structure of memories and processors as shown in D.5 (and between memories and I/O channels if needed). Multiprocessors can have any reasonable combination of shared global memory and private global memory. In one extreme of the design space, all the memory and I/O sub-systems are shared among all the processors as shown in figure D.5(a). The shared memory provides a convenient means to exchange data and to synchronise activities since any pair of processors can communicate through a shared location. In the other extreme of the design space, memory and I/O units are attached to individual processors and no sharing of memory and I/O is permitted (communication is supported through a point-to-point exchange of information).

The simplest way to construct a multiprocessor is to use a bus interconnection as shown in figure D.5(b). All the processors are connected to a shared bus that provides access to a global memory. This memory is a resource for all processors. Each processor has a local memory and a cache memory. By using the local and cache memory the need of using the shared bus is reduced and less bus interferences occur. As a final example, figure D.5(c)

performed by a spare processor [Lal85].

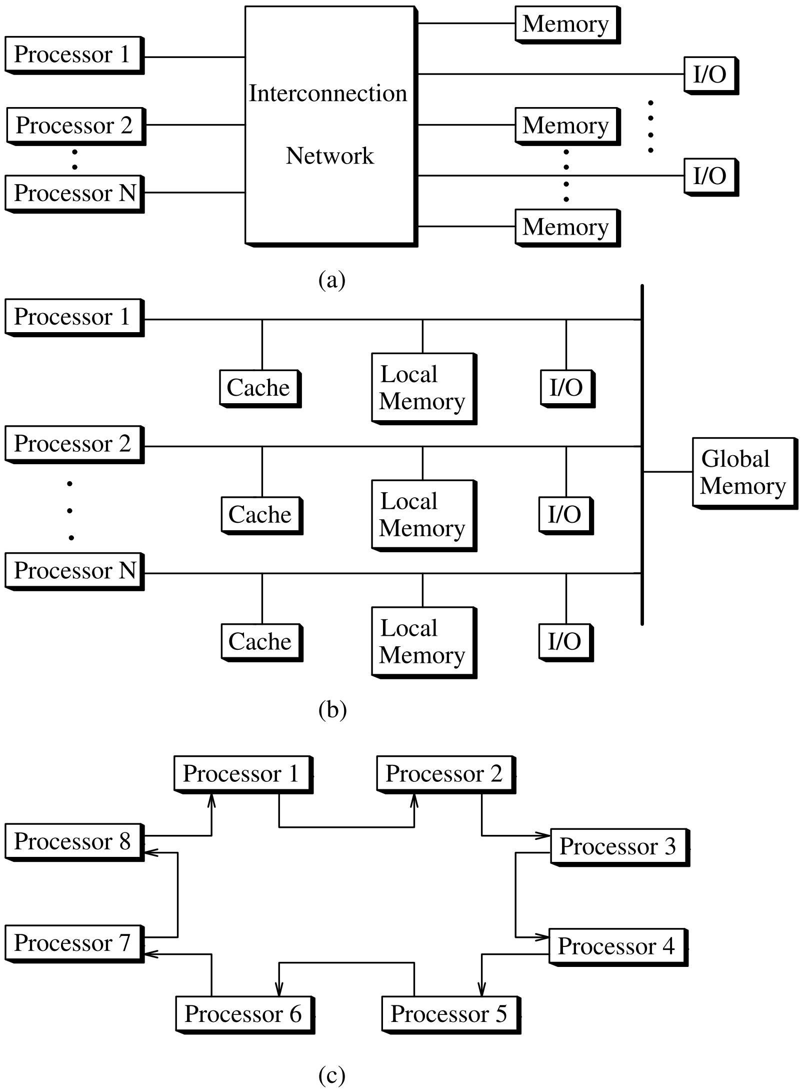


Figure D.5: Multiprocessor Structures: a) memory and I/O remote and shared, b) bus connected multiprocessor, and c) multiprocessor based on loop interconnection.

shows a ring interconnection of processors which supports point-to-point connections between processors. The efficiency of a multiprocessor system depends on the overheads included due to the cost of scheduling operations over the different processors, contention for shared resources, synchronisation, and processor-to-processor communications.

Appendix E

Proofs

E.1 Proof I

This section proofs equations 6.5 and 6.6. These equations are respectively

$$e_n = 1 - \prod_{k=1}^{n}(1 - w_k) \qquad (w_k > 0)$$
$$e_n = -1 + \prod_{k=1}^{n}(1 + w_k) \qquad (w_k < 0)$$

and they are obtained as a solution for the sequence 6.4. The negative sign in this sequence is used for the case that $w_k > 0$ and the positive sign for the case that $w_k < 0$. For the first case the sequence is

$$\left\{\begin{array}{lcl} e_1 & = & w_1 \\ e_2 & = & (1 - w_2)\, e_1 + w_2 \\ e_3 & = & (1 - w_3)\, e_2 + w_3 \\ & \vdots & \\ e_n & = & (1 - w_n)\, e_{n-1} + w_n \end{array}\right. \tag{E.1}$$

A proof that equation 6.5 is the solution for this sequence is obtained by induction on n. Equation 6.6 is derived in the same manner considering the positive signs in the sequence. By substitution of the equation for e_1 in the equation for e_2

$$e_2 = (1 - w_2)\, w_1 + w_2 = w_1 - w_2\, w_1 + w_2 = 1 - (1 - w_1)\,(1 - w_2)$$

By substitution of this result for e_2 in the equation for e_3

$$\begin{array}{lcl} e_3 & = & (1 - w_3)\,[1 - (1 - w_1)\,(1 - w_2)] + w_3 \\ & = & 1 - w_3 - (1 - w_1)\,(1 - w_2)\,(1 - w_3) + w_3 \\ & = & 1 - (1 - w_1)\,(1 - w_2)\,(1 - w_3) \end{array}$$

and therefore by induction on n the equation 6.5 is obtained.

E.2 Proof II

This section proves that:

1. The equation 6.7

$$e_n = 1 - (1-w)^n \qquad (0<w<1,\, n>1)$$

is monotonically increasing and convex with respect to n and w. The function is monotonically increasing with respect to n if

$$\frac{\partial e_n}{\partial n} > 0 \qquad (n>1) \tag{E.2}$$

By differentiating the equation with respect to n

$$\frac{\partial e_n}{\partial n} = -\frac{\partial[(1-w)^n]}{\partial n}$$

Considering that

$$\frac{d(a^x)}{dx} = a^x \ln a$$

the differentiation gives

$$\frac{\partial e_n}{\partial n} = -(1-w)^n \ln(1-w) \tag{E.3}$$

Since $0<w<1$ it is true that

$$\ln(1-w) < 0$$

and equation E.2 is always true.

The function is convex with respect to n if

$$\frac{\partial^2 e_n}{\partial^2 n} < 0 \qquad (n>1) \tag{E.4}$$

and by differentiation of equation E.3

$$\frac{\partial^2 e_n}{\partial^2 n} = -[\ln(1-w)]^2(1-w)^n < 0 \qquad (n>1) \tag{E.5}$$

Similarly, the function is monotonically increasing with respect to w since

$$\frac{\partial e_n}{\partial w} = -n(1-w)^{n-1}(-1) = n(1-w)^{n-1} > 0 \qquad (0<w<1) \tag{E.6}$$

and it is convex with respect to w since

$$\frac{\partial^2 e_n}{\partial^2 w} = -(n^2-n)(1-w)^{n-2} < 0 \qquad (0<w<1) \tag{E.7}$$

2. For each pair of values $0 < w < 1$ and $n > 1$ the function $e_n = f(w)$ always rises faster than $e_n = f(n)$. The proof of this requires

$$\frac{\partial e_n}{\partial w} > \frac{\partial e_n}{\partial n}$$

From equation E.6 and equation E.3

$$n(1-w)^{n-1} > -(1-w)^n \ln(1-w)$$

and therefore

$$n > -(1-w)\ln(1-w) \qquad (0 < w < 1,\ n > 1)$$

Figure E.1 shows that the second hand of this inequality is always smaller than 1 for the range of values for w considered.

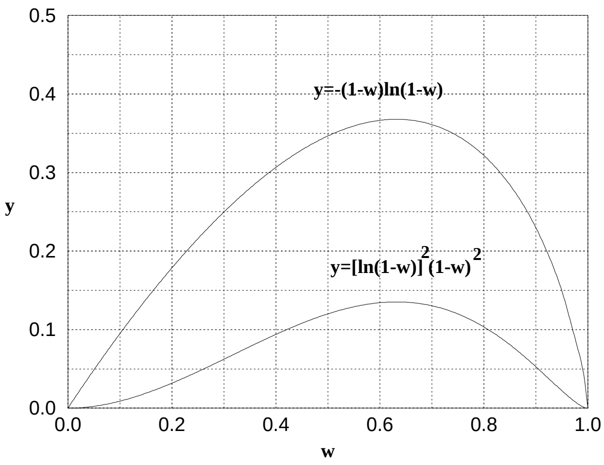


Figure E.1: Graph I

3. For each pair of values $0 < w < 1$ and $n > 1$ the rate of increase decreases less rapidly for $e_n = f(w)$ than for $e_n = f(n)$. The first function is more convex than the second. As a consequence, a few 'good' items of knowledge will have a higher contribution than more items of a lesser quality. The proof of this requires that

$$\frac{\partial^2 e_n}{\partial^2 w} < \frac{\partial^2 e_n}{\partial^2 n}$$

and by means of equation E.7 and equation E.5

$$-(n^2 - n)(1-w)^{n-2} < -[\ln(1-w)]^2(1-w)^n$$

which gives

$$(n^2 - n) > [\ln(1-w)]^2(1-w)^2$$

For $n > 1$ the left hand side of the inequality is always greater than 1 and for $0 < w < 1$ the right hand side is always smaller than 1 as shown in figure E.1.

E.3 Proof III

This section proves that in the system 6.31

$$\left.\begin{array}{rl} E_i = & 1-(1-r\,e)^{n_i} \\ r = & \frac{1}{n_i\,|1-r_i|+1} \end{array}\right\} \tag{E.8}$$

the total contribution of the sub-cells E_i increases always when n_i increases despite the fact that r decreases with n_i. This implies

$$\frac{\partial E_i}{\partial n_i} > 0 \qquad (n_i > 1) \tag{E.9}$$

From the first equation of the system E.8

$$1 - E_i = (1 - r\,e)^{n_i}$$

and by taking logarithms in both sides

$$\ln(1 - E_i) = n_i\,\ln(1 - r\,e)$$

Differentiating both sides with respect to n_i

$$\frac{1}{1-E_i}\,(-\frac{\partial E_i}{\partial n_i}) = \frac{\partial}{\partial n_i}[n_i\,\ln(1-r\,e)]$$

which gives

$$\frac{\partial E_i}{\partial n_i} = (E_i - 1)\,[\ln(1-r\,e) + n_i\,\frac{1}{1-r\,e}(-e\,\frac{\partial r}{\partial n_i})]$$

From the second equation of the system E.8

$$\frac{\partial r}{\partial n_i} = \frac{-|1-r_i|}{(n_i\,|1-r_i|+1)^2}$$

and then

$$\frac{\partial E_i}{\partial n_i} = (1-E_i)\,[-\ln(1-\frac{e}{n_i\,|1-r_i|+1}) - \frac{n_i\,e\,|1-r_i|}{(n_i\,|1-r_i|+1-e)\,(n_i\,|1-r_i|+1)}]$$

By taking

$$\begin{array}{ll} x = \frac{e}{n_i\,|1-r_i|+1} & \qquad 0 < x \leq 1 \\ k = \frac{n_i\,|1-r_i|}{n_i\,|1-r_i|+1-e} & \qquad 0 \leq k \leq 1 \end{array}$$

the above equation becomes

$$\frac{\partial E_i}{\partial n_i} = (1-E_i)\,[-\ln(1-x) - k\,x]$$

Since $(1-E_i) \geq 0$, $\frac{\partial E_i}{\partial n_i} > 0$ if

$$k\,x < -ln(1-x)$$

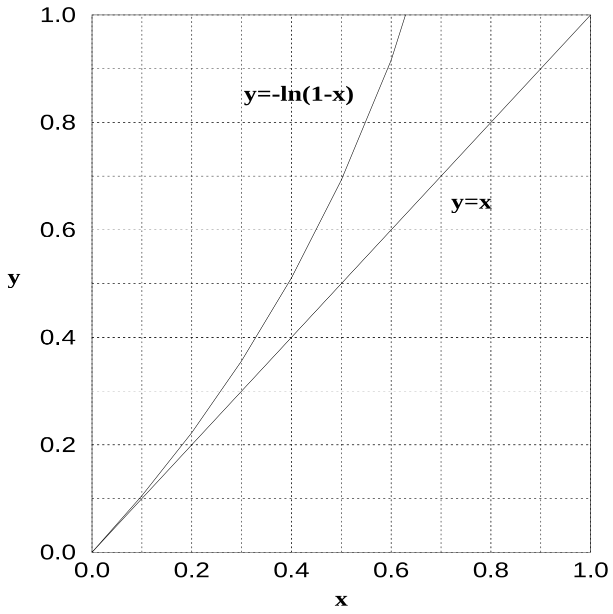


Figure E.2: Graph II

The functions $y = x$ and $y = -\ln(1-x)$ are depicted in figure E.2. From this figure, $y = x$ is always smaller than $y = -\ln(1-x)$ in the interval $0 < x \leq 1$. Since $k \leq 1$, this inequality must always hold and equation E.9 is true.

From figure E.2, the largest values of $\frac{\partial E_i}{\partial n_i}$ are obtained for values of x close to 1. For this to happen, $r_i \to 1$. When $r_i \to 1$, $x \to e$ and $k \to 0$ thus widening the gap between the functions in figure E.2. The fact that r_i is near 1 implies that the situation is well represented. A well-represented situation with a larger number of sub-cells is, of course, more difficult to achieve and therefore it is more significant. For $r_i \gg 1$, $x \to 0$ and $k \to 1$ with the increase of n_i. Then, the increase of E_i with n_i tends to be very small. For $r_i \ll 1$ the derivative is more sensible to n_i but it decreases rapidly with n_i. Finally, larger values of e will give smaller values for the derivative since $k \to 1$ when $e \to 1$ and this narrows the gap between the functions in figure E.2.

Appendix F

Design Hierarchy for the Case-Studies

F.1 Counter

```
(1) Top Cell: counter [Lib: counter]
    3 instances
     1 instance(s) of cell ‘mux’
     1 instance(s) of cell ‘adder’
     1 instance(s) of cell ‘register’
      (2) Cell: mux [Lib: counter]
      (2) Cell: adder [Lib: counter]
      (2) Cell: register [Lib: counter]
```

F.2 H_bilbo

```
(1) Top Cell: h_bilbo_register [Lib: h_design]
    5 instances
     1 instance(s) of cell ‘serial_logic’
     4 instance(s) of cell ‘bilbo_stage’
      (2) Cell: serial_logic [Lib: h_design]
          2 instances
           1 instance(s) of cell ‘2_to_1_multiplexer’
           1 instance(s) of cell ‘exor’
            (3) Cell: 2_to_1_multiplexer [Lib: h_design]
                4 instances
                 3 instance(s) of cell ‘nand’
                 1 instance(s) of cell ‘inverter’
                  (4) Cell: nand [Lib: h_devices]
                  (4) Cell: inverter [Lib: h_devices]
            (3) Cell: exor [Lib: h_devices]
      (2) Cell: bilbo_stage [Lib: h_design]
          4 instances
           1 instance(s) of cell ‘nor’
           1 instance(s) of cell ‘and’
           1 instance(s) of cell ‘d_flipflop’
           1 instance(s) of cell ‘exor’
            (3) Cell: nor [Lib: h_devices]
            (3) Cell: and [Lib: h_devices]
```

```
        (3) Cell: d_flipflop [Lib: h_devices]
        (3) Cell: exor [Lib: h_devices]
```

F.3 Add

```
(1) Top Cell: add [Lib: adder]
    12 instances
     3 instance(s) of cell `and2'
     3 instance(s) of cell `not'
     4 instance(s) of cell `and3'
     1 instance(s) of cell `or4'
     1 instance(s) of cell `or3'
      (2) Cell: and2 [Lib: adder]
      (2) Cell: not [Lib: adder]
      (2) Cell: and3 [Lib: adder]
          1 instances
           1 instance(s) of cell `xxxfunct2'
            (3) Cell: xxxfunct2 [Lib: adder]
                2 instances
                 1 instance(s) of cell `and2'
                 1 instance(s) of cell `xxxfunct1'
                  (4) Cell: and2 [Lib: adder]
                  (4) Cell: xxxfunct1 [Lib: adder]
                      1 instances
                       1 instance(s) of cell `and2'
                        (5) Cell: and2 [Lib: adder]
      (2) Cell: or4 [Lib: adder]
          1 instances
           1 instance(s) of cell `xxxfunct8'
            (3) Cell: xxxfunct8 [Lib: adder]
                2 instances
                 1 instance(s) of cell `or2'
                 1 instance(s) of cell `xxxfunct7'
                  (4) Cell: or2 [Lib: adder]
                  (4) Cell: xxxfunct7 [Lib: adder]
                      2 instances
                       1 instance(s) of cell `or2'
                       1 instance(s) of cell `xxxfunct6'
                        (5) Cell: or2 [Lib: adder]
                        (5) Cell: xxxfunct6 [Lib: adder]
                            1 instances
                             1 instance(s) of cell `or2'
                              (6) Cell: or2 [Lib: adder]
      (2) Cell: or3 [Lib: adder]
          1 instances
           1 instance(s) of cell `xxxfunct4'
            (3) Cell: xxxfunct4 [Lib: adder]
                2 instances
                 1 instance(s) of cell `or2'
                 1 instance(s) of cell `xxxfunct3'
                  (4) Cell: or2 [Lib: adder]
                  (4) Cell: xxxfunct3 [Lib: adder]
```

```
                  1 instances
                   1 instance(s) of cell ‘or2’
                    (5) Cell: or2 [Lib: adder]
```

F.4 Valid1

```
(1) Top Cell: valid1 [Lib: design]
    3 instances
     1 instance(s) of cell ‘subb’
     2 instance(s) of cell ‘suba’
      (2) Cell: subb [Lib: design]
          12 instances
           1 instance(s) of cell ‘ctnand3’
           1 instance(s) of cell ‘ctaoi’
           1 instance(s) of cell ‘ctoai’
           1 instance(s) of cell ‘ctexnor’
           1 instance(s) of cell ‘ctnor4’
           2 instance(s) of cell ‘ctnand4’
           1 instance(s) of cell ‘cthadd’
           1 instance(s) of cell ‘ctexor’
           1 instance(s) of cell ‘ctinv4’
           2 instance(s) of cell ‘ctfadd’
            (3) Cell: ctnand3 [Lib: ellacad042]
            (3) Cell: ctaoi [Lib: ellacad042]
            (3) Cell: ctoai [Lib: ellacad042]
            (3) Cell: ctexnor [Lib: ellacad042]
            (3) Cell: ctnor4 [Lib: ellacad042]
            (3) Cell: ctnand4 [Lib: ellacad042]
            (3) Cell: cthadd [Lib: ellacad042]
            (3) Cell: ctexor [Lib: ellacad042]
            (3) Cell: ctinv4 [Lib: ellacad042]
            (3) Cell: ctfadd [Lib: ellacad042]
      (2) Cell: suba [Lib: design]
          6 instances
           1 instance(s) of cell ‘ctnor3’
           1 instance(s) of cell ‘ct2anor’
           1 instance(s) of cell ‘ctnand2’
           1 instance(s) of cell ‘ctinv1’
           2 instance(s) of cell ‘ctnor2’
            (3) Cell: ctnor3 [Lib: ellacad042]
            (3) Cell: ct2anor [Lib: ellacad042]
            (3) Cell: ctnand2 [Lib: ellacad042]
            (3) Cell: ctinv1 [Lib: ellacad042]
            (3) Cell: ctnor2 [Lib: ellacad042]
```

F.5 6g011a

```
(1) Top Cell: chip [Lib: user]ist]
    96 instances
     10 instance(s) of cell ‘lg2’
     18 instance(s) of cell ‘lg31’
```

```
58 instance(s) of cell ‘nor1’
1 instance(s) of cell ‘nor2’
1 instance(s) of cell ‘reg8’
1 instance(s) of cell ‘shift8’
1 instance(s) of cell ‘fsm’
1 instance(s) of cell ‘mux4to1’
2 instance(s) of cell ‘mux2to1’
1 instance(s) of cell ‘dmux1to2’
1 instance(s) of cell ‘dmux1to4’
1 instance(s) of cell ‘eqv’
 (2) Cell: lg2 [Lib: peripherials]
 (2) Cell: lg31 [Lib: peripherials]
 (2) Cell: nor1 [Lib: primitives]
 (2) Cell: nor2 [Lib: primitives]
 (2) Cell: reg8 [Lib: user]
     10 instances
      8 instance(s) of cell ‘dl1q’
      2 instance(s) of cell ‘nor1’
       (3) Cell: dl1q [Lib: user]
           4 instances
            4 instance(s) of cell ‘nor2’
             (4) Cell: nor2 [Lib: primitives]
       (3) Cell: nor1 [Lib: primitives]
 (2) Cell: shift8 [Lib: user]
     13 instances
      8 instance(s) of cell ‘dt1q’
      4 instance(s) of cell ‘nor1’
      1 instance(s) of cell ‘nor2’
       (3) Cell: dt1q [Lib: user]
           6 instances
            1 instance(s) of cell ‘nor3’
            5 instance(s) of cell ‘nor2’
             (4) Cell: nor3 [Lib: primitives]
             (4) Cell: nor2 [Lib: primitives]
       (3) Cell: nor1 [Lib: primitives]
       (3) Cell: nor2 [Lib: primitives]
 (2) Cell: fsm [Lib: user]
     32 instances
      7 instance(s) of cell ‘nor3’
      9 instance(s) of cell ‘nor4’
      8 instance(s) of cell ‘nor1’
      4 instance(s) of cell ‘nor2’
      4 instance(s) of cell ‘dt1rqn’
       (3) Cell: nor3 [Lib: primitives]
       (3) Cell: nor4 [Lib: primitives]
       (3) Cell: nor1 [Lib: primitives]
       (3) Cell: nor2 [Lib: primitives]
       (3) Cell: dt1rqn [Lib: user]
           6 instances
            3 instance(s) of cell ‘nor3’
            3 instance(s) of cell ‘nor2’
             (4) Cell: nor3 [Lib: primitives]
             (4) Cell: nor2 [Lib: primitives]
```

```
(2) Cell: mux4to1 [Lib: user]
    7 instances
     4 instance(s) of cell 'nor3'
     1 instance(s) of cell 'nor4'
     2 instance(s) of cell 'nor1'
      (3) Cell: nor3 [Lib: primitives]
      (3) Cell: nor4 [Lib: primitives]
      (3) Cell: nor1 [Lib: primitives]
(2) Cell: mux2to1 [Lib: user]
    4 instances
     3 instance(s) of cell 'nor2'
     1 instance(s) of cell 'nor1'
      (3) Cell: nor2 [Lib: primitives]
      (3) Cell: nor1 [Lib: primitives]
(2) Cell: dmux1to2 [Lib: user]
    4 instances
     2 instance(s) of cell 'nor2'
     2 instance(s) of cell 'nor1'
      (3) Cell: nor2 [Lib: primitives]
      (3) Cell: nor1 [Lib: primitives]
(2) Cell: dmux1to4 [Lib: user]
    7 instances
     4 instance(s) of cell 'nor3'
     3 instance(s) of cell 'nor1'
      (3) Cell: nor3 [Lib: primitives]
      (3) Cell: nor1 [Lib: primitives]
(2) Cell: eqv [Lib: user]
    4 instances
     4 instance(s) of cell 'nor2'
      (3) Cell: nor2 [Lib: primitives]
```

F.6 Multmilldesign

```
(1) Top Cell: multmill [Lib: netlibrary]
    178 instances
     13 instance(s) of cell 'ctinv4'
     30 instance(s) of cell 'ctinv1'
     31 instance(s) of cell 'split'
     11 instance(s) of cell 'ct2anor'
     10 instance(s) of cell 'ctnand2'
     32 instance(s) of cell 'ctnor2'
     1 instance(s) of cell 'ctaoi'
     4 instance(s) of cell 'ctnor3'
     3 instance(s) of cell 'ctoai'
     8 instance(s) of cell 'ctnand3'
     11 instance(s) of cell 'ct2onand'
     3 instance(s) of cell 'ctexor'
     2 instance(s) of cell 'ctnor4'
     3 instance(s) of cell 'ctnand4'
     4 instance(s) of cell 'multadd'
     12 instance(s) of cell 'ctfadd'
      (2) Cell: ctinv4 [Lib: cad042]
```

```
      (2) Cell: ctinv1 [Lib: cad042]
      (2) Cell: split [Lib: netlibrary]
      (2) Cell: ct2anor [Lib: cad042]
      (2) Cell: ctnand2 [Lib: cad042]
      (2) Cell: ctnor2 [Lib: cad042]
      (2) Cell: ctaoi [Lib: cad042]
      (2) Cell: ctnor3 [Lib: cad042]
      (2) Cell: ctoai [Lib: cad042]
      (2) Cell: ctnand3 [Lib: cad042]
      (2) Cell: ct2onand [Lib: cad042]
      (2) Cell: ctexor [Lib: cad042]
      (2) Cell: ctnor4 [Lib: cad042]
      (2) Cell: ctnand4 [Lib: cad042]
      (2) Cell: multadd [Lib: netlibrary]
          16 instances
           3 instance(s) of cell `split'
           3 instance(s) of cell `ctnand4'
           2 instance(s) of cell `ctexor'
           2 instance(s) of cell `ctnand2'
           2 instance(s) of cell `ctinv1'
           4 instance(s) of cell `ctnand3'
            (3) Cell: split [Lib: netlibrary]
            (3) Cell: ctnand4 [Lib: cad042]
            (3) Cell: ctexor [Lib: cad042]
            (3) Cell: ctnand2 [Lib: cad042]
            (3) Cell: ctinv1 [Lib: cad042]
            (3) Cell: ctnand3 [Lib: cad042]
      (2) Cell: ctfadd [Lib: cad042]
```

F.7 Designtop1

```
(1) Top Cell: decodedmill [Lib: design]
    61 instances
     1 instance(s) of cell `xxxfunct6'
     1 instance(s) of cell `xxxfunct1'
     6 instance(s) of cell `invsel4'
     1 instance(s) of cell `reg'
     1 instance(s) of cell `xxxfunct7'
     1 instance(s) of cell `xxxfunct8'
     1 instance(s) of cell `ctnor2'
     1 instance(s) of cell `xxxfunct9'
     1 instance(s) of cell `xxxfunct10'
     1 instance(s) of cell `eq4'
     1 instance(s) of cell `xxxfunct4'
     1 instance(s) of cell `xxxfunct2'
     1 instance(s) of cell `xxxfunct11'
     1 instance(s) of cell `xxxfunct3'
     1 instance(s) of cell `xxxfunct5'
     28 instance(s) of cell `ctihm'
     9 instance(s) of cell `ctopa'
     2 instance(s) of cell `multmill'
     2 instance(s) of cell `millc'
```

```
(2) Cell: xxxfunct6 [Lib: design]
    16 instances
     8 instance(s) of cell ‘ctinv1’
     4 instance(s) of cell ‘ct2anor’
     4 instance(s) of cell ‘ctdf’
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: xxxfunct1 [Lib: design]
    13 instances
     7 instance(s) of cell ‘ctinv1’
     3 instance(s) of cell ‘ct2anor’
     3 instance(s) of cell ‘ctdf’
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: invsel4 [Lib: design]
    4 instances
     4 instance(s) of cell ‘ctexor’
      (3) Cell: ctexor [Lib: ellacad042]
(2) Cell: reg [Lib: design]
    16 instances
     8 instance(s) of cell ‘ctinv1’
     4 instance(s) of cell ‘ct2anor’
     4 instance(s) of cell ‘ctdf’
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: xxxfunct7 [Lib: design]
    16 instances
     8 instance(s) of cell ‘ctinv1’
     4 instance(s) of cell ‘ct2anor’
     4 instance(s) of cell ‘ctdf’
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: xxxfunct8 [Lib: design]
    16 instances
     8 instance(s) of cell ‘ctinv1’
     4 instance(s) of cell ‘ct2anor’
     4 instance(s) of cell ‘ctdf’
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: ctnor2 [Lib: ellacad042]
(2) Cell: xxxfunct9 [Lib: design]
    16 instances
     8 instance(s) of cell ‘ctinv1’
     4 instance(s) of cell ‘ct2anor’
     4 instance(s) of cell ‘ctdf’
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
```

```
(2) Cell: xxxfunct10 [Lib: design]
    16 instances
     8 instance(s) of cell 'ctinv1'
     4 instance(s) of cell 'ct2anor'
     4 instance(s) of cell 'ctdf'
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: eq4 [Lib: design]
    6 instances
     4 instance(s) of cell 'ctexor'
     1 instance(s) of cell 'ctinv1'
     1 instance(s) of cell 'ctnor4'
      (3) Cell: ctexor [Lib: ellacad042]
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ctnor4 [Lib: ellacad042]
(2) Cell: xxxfunct4 [Lib: design]
    16 instances
     8 instance(s) of cell 'ctinv1'
     4 instance(s) of cell 'ct2anor'
     4 instance(s) of cell 'ctdf'
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: xxxfunct2 [Lib: design]
    10 instances
     6 instance(s) of cell 'ctinv1'
     2 instance(s) of cell 'ct2anor'
     2 instance(s) of cell 'ctdf'
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: xxxfunct11 [Lib: design]
    6 instances
     4 instance(s) of cell 'ctexor'
     1 instance(s) of cell 'ctinv1'
     1 instance(s) of cell 'ctnor4'
      (3) Cell: ctexor [Lib: ellacad042]
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ctnor4 [Lib: ellacad042]
(2) Cell: xxxfunct3 [Lib: design]
    16 instances
     8 instance(s) of cell 'ctinv1'
     4 instance(s) of cell 'ct2anor'
     4 instance(s) of cell 'ctdf'
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: xxxfunct5 [Lib: design]
    16 instances
     8 instance(s) of cell 'ctinv1'
     4 instance(s) of cell 'ct2anor'
     4 instance(s) of cell 'ctdf'
```

```
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctdf [Lib: ellacad042]
(2) Cell: ctihm [Lib: ellacad042]
(2) Cell: ctopa [Lib: ellacad042]
(2) Cell: multmill [Lib: design]
    147 instances
     11 instance(s) of cell ‘ct2anor’
     30 instance(s) of cell ‘ctinv1’
     12 instance(s) of cell ‘ctfadd’
     33 instance(s) of cell ‘ctnor2’
     11 instance(s) of cell ‘ct2onand’
     4 instance(s) of cell ‘multadd’
     2 instance(s) of cell ‘ctnor4’
     9 instance(s) of cell ‘ctnand2’
     3 instance(s) of cell ‘ctnand4’
     5 instance(s) of cell ‘ctnand3’
     3 instance(s) of cell ‘ctexor’
     3 instance(s) of cell ‘ctoai’
     7 instance(s) of cell ‘ctnor3’
     13 instance(s) of cell ‘ctinv4’
     1 instance(s) of cell ‘ctaoi’
      (3) Cell: ct2anor [Lib: ellacad042]
      (3) Cell: ctinv1 [Lib: ellacad042]
      (3) Cell: ctfadd [Lib: ellacad042]
      (3) Cell: ctnor2 [Lib: ellacad042]
      (3) Cell: ct2onand [Lib: ellacad042]
      (3) Cell: multadd [Lib: design]
          13 instances
           2 instance(s) of cell ‘ctnand2’
           4 instance(s) of cell ‘ctnand3’
           2 instance(s) of cell ‘ctinv1’
           3 instance(s) of cell ‘ctnand4’
           2 instance(s) of cell ‘ctexor’
            (4) Cell: ctnand2 [Lib: ellacad042]
            (4) Cell: ctnand3 [Lib: ellacad042]
            (4) Cell: ctinv1 [Lib: ellacad042]
            (4) Cell: ctnand4 [Lib: ellacad042]
            (4) Cell: ctexor [Lib: ellacad042]
      (3) Cell: ctnor4 [Lib: ellacad042]
      (3) Cell: ctnand2 [Lib: ellacad042]
      (3) Cell: ctnand4 [Lib: ellacad042]
      (3) Cell: ctnand3 [Lib: ellacad042]
      (3) Cell: ctexor [Lib: ellacad042]
      (3) Cell: ctoai [Lib: ellacad042]
      (3) Cell: ctnor3 [Lib: ellacad042]
      (3) Cell: ctinv4 [Lib: ellacad042]
      (3) Cell: ctaoi [Lib: ellacad042]
(2) Cell: millc [Lib: design]
    50 instances
     9 instance(s) of cell ‘ctnand4’
     4 instance(s) of cell ‘ctnand3’
     24 instance(s) of cell ‘ctinv4’
```

```
            3 instance(s) of cell 'ctnand2'
            5 instance(s) of cell 'ctnor4'
            5 instance(s) of cell 'ctnor3'
             (3) Cell: ctnand4 [Lib: ellacad042]
             (3) Cell: ctnand3 [Lib: ellacad042]
             (3) Cell: ctinv4 [Lib: ellacad042]
             (3) Cell: ctnand2 [Lib: ellacad042]
             (3) Cell: ctnor4 [Lib: ellacad042]
             (3) Cell: ctnor3 [Lib: ellacad042]
```

F.8 18ara700a

```
(1) Top Cell: chip [Lib: user]ist]
    47 instances
     13 instance(s) of cell 'ar38'
     23 instance(s) of cell 'ar2'
     1 instance(s) of cell 'tbsely'
     1 instance(s) of cell 'rngsel'
     1 instance(s) of cell 'counter10'
     1 instance(s) of cell 'bitshift'
     1 instance(s) of cell 'tbsel'
     3 instance(s) of cell 'rnor1'
     1 instance(s) of cell 'rexor'
     1 instance(s) of cell 'resout'
     1 instance(s) of cell 'valcomp4'
      (2) Cell: ar38 [Lib: peripherials]
      (2) Cell: ar2 [Lib: peripherials]
      (2) Cell: tbsely [Lib: user]
          20 instances
           8 instance(s) of cell 'rnor2'
           11 instance(s) of cell 'rnor1'
           1 instance(s) of cell 'rnor4'
            (3) Cell: rnor2 [Lib: primitives]
            (3) Cell: rnor1 [Lib: primitives]
            (3) Cell: rnor4 [Lib: primitives]
      (2) Cell: rngsel [Lib: user]
          31 instances
           1 instance(s) of cell 'rnor7'
           18 instance(s) of cell 'rnor1'
           6 instance(s) of cell 'rnor2'
           1 instance(s) of cell 'tbsel11'
           2 instance(s) of cell 'valcomp7'
           1 instance(s) of cell 'tbselx'
           2 instance(s) of cell 'counter10'
            (3) Cell: rnor7 [Lib: primitives]
            (3) Cell: rnor1 [Lib: primitives]
            (3) Cell: rnor2 [Lib: primitives]
            (3) Cell: tbsel11 [Lib: user]
                25 instances
                 10 instance(s) of cell 'rnor2'
                 14 instance(s) of cell 'rnor1'
                 1 instance(s) of cell 'rnor4'
```

```
      (4) Cell: rnor2 [Lib: primitives]
      (4) Cell: rnor1 [Lib: primitives]
      (4) Cell: rnor4 [Lib: primitives]
(3) Cell: valcomp7 [Lib: user]
    10 instances
     7 instance(s) of cell 'rexor'
     1 instance(s) of cell 'rnor5'
     2 instance(s) of cell 'rnor2'
      (4) Cell: rexor [Lib: user]
          3 instances
           3 instance(s) of cell 'rnor2'
            (5) Cell: rnor2 [Lib: primitives]
      (4) Cell: rnor5 [Lib: primitives]
      (4) Cell: rnor2 [Lib: primitives]
(3) Cell: tbselx [Lib: user]
    24 instances
     10 instance(s) of cell 'rnor2'
     13 instance(s) of cell 'rnor1'
     1 instance(s) of cell 'rnor4'
      (4) Cell: rnor2 [Lib: primitives]
      (4) Cell: rnor1 [Lib: primitives]
      (4) Cell: rnor4 [Lib: primitives]
(3) Cell: counter10 [Lib: user]
    3 instances
     1 instance(s) of cell 'halcou'
     2 instance(s) of cell 'cou4'
      (4) Cell: halcou [Lib: user]
          13 instances
           4 instance(s) of cell 'rnor1'
           4 instance(s) of cell 'rnor3'
           3 instance(s) of cell 'rnor2'
           2 instance(s) of cell 'dt1sqn'
            (5) Cell: rnor1 [Lib: primitives]
            (5) Cell: rnor3 [Lib: primitives]
            (5) Cell: rnor2 [Lib: primitives]
            (5) Cell: dt1sqn [Lib: user]
                6 instances
                 2 instance(s) of cell 'rnor1'
                 2 instance(s) of cell 'rnor2'
                 2 instance(s) of cell 'rnor3'
                  (6) Cell: rnor1 [Lib: primitives]
                  (6) Cell: rnor2 [Lib: primitives]
                  (6) Cell: rnor3 [Lib: primitives]
      (4) Cell: cou4 [Lib: user]
          23 instances
           5 instance(s) of cell 'rnor2'
           6 instance(s) of cell 'rnor3'
           5 instance(s) of cell 'rnor1'
           1 instance(s) of cell 'rnor4'
           4 instance(s) of cell 'dt1sqn'
           2 instance(s) of cell 'rnor5'
            (5) Cell: rnor2 [Lib: primitives]
            (5) Cell: rnor3 [Lib: primitives]
```

```
                    (5) Cell: rnor1 [Lib: primitives]
                    (5) Cell: rnor4 [Lib: primitives]
                    (5) Cell: dt1sqn [Lib: user]
                        6 instances
                         2 instance(s) of cell 'rnor1'
                         2 instance(s) of cell 'rnor2'
                         2 instance(s) of cell 'rnor3'
                          (6) Cell: rnor1 [Lib: primitives]
                          (6) Cell: rnor2 [Lib: primitives]
                          (6) Cell: rnor3 [Lib: primitives]
                    (5) Cell: rnor5 [Lib: primitives]
  (2) Cell: counter10 [Lib: user]
      3 instances
       1 instance(s) of cell 'halcou'
       2 instance(s) of cell 'cou4'
        (3) Cell: halcou [Lib: user]
            13 instances
             4 instance(s) of cell 'rnor1'
             4 instance(s) of cell 'rnor3'
             3 instance(s) of cell 'rnor2'
             2 instance(s) of cell 'dt1sqn'
              (4) Cell: rnor1 [Lib: primitives]
              (4) Cell: rnor3 [Lib: primitives]
              (4) Cell: rnor2 [Lib: primitives]
              (4) Cell: dt1sqn [Lib: user]
                  6 instances
                   2 instance(s) of cell 'rnor1'
                   2 instance(s) of cell 'rnor2'
                   2 instance(s) of cell 'rnor3'
                    (5) Cell: rnor1 [Lib: primitives]
                    (5) Cell: rnor2 [Lib: primitives]
                    (5) Cell: rnor3 [Lib: primitives]
        (3) Cell: cou4 [Lib: user]
            23 instances
             5 instance(s) of cell 'rnor2'
             6 instance(s) of cell 'rnor3'
             5 instance(s) of cell 'rnor1'
             1 instance(s) of cell 'rnor4'
             4 instance(s) of cell 'dt1sqn'
             2 instance(s) of cell 'rnor5'
              (4) Cell: rnor2 [Lib: primitives]
              (4) Cell: rnor3 [Lib: primitives]
              (4) Cell: rnor1 [Lib: primitives]
              (4) Cell: rnor4 [Lib: primitives]
              (4) Cell: dt1sqn [Lib: user]
                  6 instances
                   2 instance(s) of cell 'rnor1'
                   2 instance(s) of cell 'rnor2'
                   2 instance(s) of cell 'rnor3'
                    (5) Cell: rnor1 [Lib: primitives]
                    (5) Cell: rnor2 [Lib: primitives]
                    (5) Cell: rnor3 [Lib: primitives]
              (4) Cell: rnor5 [Lib: primitives]
```

```
(2) Cell: bitshift [Lib: user]
    23 instances
     10 instance(s) of cell ‘rnor1’
     1 instance(s) of cell ‘rnor4’
     1 instance(s) of cell ‘rnor2’
     1 instance(s) of cell ‘selunit4’
     1 instance(s) of cell ‘selunit6’
     1 instance(s) of cell ‘selunit7’
     1 instance(s) of cell ‘selunit5’
     1 instance(s) of cell ‘selunit3’
     1 instance(s) of cell ‘encoder’
     1 instance(s) of cell ‘sftlogic’
     1 instance(s) of cell ‘adder’
     1 instance(s) of cell ‘selunit1’
     1 instance(s) of cell ‘selunit2’
     1 instance(s) of cell ‘selunit0’
      (3) Cell: rnor1 [Lib: primitives]
      (3) Cell: rnor4 [Lib: primitives]
      (3) Cell: rnor2 [Lib: primitives]
      (3) Cell: selunit4 [Lib: user]
          13 instances
           7 instance(s) of cell ‘rnor1’
           3 instance(s) of cell ‘rnor2’
           1 instance(s) of cell ‘rnor4’
           1 instance(s) of cell ‘valuecomp’
           1 instance(s) of cell ‘counteru’
            (4) Cell: rnor1 [Lib: primitives]
            (4) Cell: rnor2 [Lib: primitives]
            (4) Cell: rnor4 [Lib: primitives]
            (4) Cell: valuecomp [Lib: user]
                7 instances
                 1 instance(s) of cell ‘rnor3’
                 1 instance(s) of cell ‘rnor2’
                 2 instance(s) of cell ‘rnor1’
                 3 instance(s) of cell ‘rexor’
                  (5) Cell: rnor3 [Lib: primitives]
                  (5) Cell: rnor2 [Lib: primitives]
                  (5) Cell: rnor1 [Lib: primitives]
                  (5) Cell: rexor [Lib: user]
                      3 instances
                       3 instance(s) of cell ‘rnor2’
                        (6) Cell: rnor2 [Lib: primitives]
            (4) Cell: counteru [Lib: user]
                2 instances
                 1 instance(s) of cell ‘rnor1’
                 1 instance(s) of cell ‘cou’
                  (5) Cell: rnor1 [Lib: primitives]
                  (5) Cell: cou [Lib: user]
                      17 instances
                       5 instance(s) of cell ‘rnor3’
                       4 instance(s) of cell ‘rnor2’
                       1 instance(s) of cell ‘rnor4’
                       4 instance(s) of cell ‘rnor1’
```

```
                  3 instance(s) of cell 'dt1sqn'
                   (6) Cell: rnor3 [Lib: primitives]
                   (6) Cell: rnor2 [Lib: primitives]
                   (6) Cell: rnor4 [Lib: primitives]
                   (6) Cell: rnor1 [Lib: primitives]
                   (6) Cell: dt1sqn [Lib: user]
                       6 instances
                        2 instance(s) of cell 'rnor1'
                        2 instance(s) of cell 'rnor2'
                        2 instance(s) of cell 'rnor3'
                         (7) Cell: rnor1 [Lib: primitives]
                         (7) Cell: rnor2 [Lib: primitives]
                         (7) Cell: rnor3 [Lib: primitives]
(3) Cell: selunit6 [Lib: user]
    14 instances
     1 instance(s) of cell 'rnor4'
     8 instance(s) of cell 'rnor1'
     3 instance(s) of cell 'rnor2'
     1 instance(s) of cell 'valuecomp'
     1 instance(s) of cell 'counteru'
      (4) Cell: rnor4 [Lib: primitives]
      (4) Cell: rnor1 [Lib: primitives]
      (4) Cell: rnor2 [Lib: primitives]
      (4) Cell: valuecomp [Lib: user]
          7 instances
           1 instance(s) of cell 'rnor3'
           1 instance(s) of cell 'rnor2'
           2 instance(s) of cell 'rnor1'
           3 instance(s) of cell 'rexor'
            (5) Cell: rnor3 [Lib: primitives]
            (5) Cell: rnor2 [Lib: primitives]
            (5) Cell: rnor1 [Lib: primitives]
            (5) Cell: rexor [Lib: user]
                3 instances
                 3 instance(s) of cell 'rnor2'
                  (6) Cell: rnor2 [Lib: primitives]
      (4) Cell: counteru [Lib: user]
          2 instances
           1 instance(s) of cell 'rnor1'
           1 instance(s) of cell 'cou'
            (5) Cell: rnor1 [Lib: primitives]
            (5) Cell: cou [Lib: user]
                17 instances
                 5 instance(s) of cell 'rnor3'
                 4 instance(s) of cell 'rnor2'
                 1 instance(s) of cell 'rnor4'
                 4 instance(s) of cell 'rnor1'
                 3 instance(s) of cell 'dt1sqn'
                  (6) Cell: rnor3 [Lib: primitives]
                  (6) Cell: rnor2 [Lib: primitives]
                  (6) Cell: rnor4 [Lib: primitives]
                  (6) Cell: rnor1 [Lib: primitives]
                  (6) Cell: dt1sqn [Lib: user]
```

```
                      6 instances
                       2 instance(s) of cell 'rnor1'
                       2 instance(s) of cell 'rnor2'
                       2 instance(s) of cell 'rnor3'
                        (7) Cell: rnor1 [Lib: primitives]
                        (7) Cell: rnor2 [Lib: primitives]
                        (7) Cell: rnor3 [Lib: primitives]
(3) Cell: selunit7 [Lib: user]
    15 instances
     9 instance(s) of cell 'rnor1'
     3 instance(s) of cell 'rnor2'
     1 instance(s) of cell 'rnor4'
     1 instance(s) of cell 'valuecomp'
     1 instance(s) of cell 'counteru'
      (4) Cell: rnor1 [Lib: primitives]
      (4) Cell: rnor2 [Lib: primitives]
      (4) Cell: rnor4 [Lib: primitives]
      (4) Cell: valuecomp [Lib: user]
          7 instances
           1 instance(s) of cell 'rnor3'
           1 instance(s) of cell 'rnor2'
           2 instance(s) of cell 'rnor1'
           3 instance(s) of cell 'rexor'
            (5) Cell: rnor3 [Lib: primitives]
            (5) Cell: rnor2 [Lib: primitives]
            (5) Cell: rnor1 [Lib: primitives]
            (5) Cell: rexor [Lib: user]
                3 instances
                 3 instance(s) of cell 'rnor2'
                  (6) Cell: rnor2 [Lib: primitives]
      (4) Cell: counteru [Lib: user]
          2 instances
           1 instance(s) of cell 'rnor1'
           1 instance(s) of cell 'cou'
            (5) Cell: rnor1 [Lib: primitives]
            (5) Cell: cou [Lib: user]
                17 instances
                 5 instance(s) of cell 'rnor3'
                 4 instance(s) of cell 'rnor2'
                 1 instance(s) of cell 'rnor4'
                 4 instance(s) of cell 'rnor1'
                 3 instance(s) of cell 'dt1sqn'
                  (6) Cell: rnor3 [Lib: primitives]
                  (6) Cell: rnor2 [Lib: primitives]
                  (6) Cell: rnor4 [Lib: primitives]
                  (6) Cell: rnor1 [Lib: primitives]
                  (6) Cell: dt1sqn [Lib: user]
                      6 instances
                       2 instance(s) of cell 'rnor1'
                       2 instance(s) of cell 'rnor2'
                       2 instance(s) of cell 'rnor3'
                        (7) Cell: rnor1 [Lib: primitives]
                        (7) Cell: rnor2 [Lib: primitives]
```

```
                        (7) Cell: rnor3 [Lib: primitives]
(3) Cell: selunit5 [Lib: user]
    14 instances
     1 instance(s) of cell ‘rnor4’
     8 instance(s) of cell ‘rnor1’
     3 instance(s) of cell ‘rnor2’
     1 instance(s) of cell ‘valuecomp’
     1 instance(s) of cell ‘counteru’
      (4) Cell: rnor4 [Lib: primitives]
      (4) Cell: rnor1 [Lib: primitives]
      (4) Cell: rnor2 [Lib: primitives]
      (4) Cell: valuecomp [Lib: user]
          7 instances
           1 instance(s) of cell ‘rnor3’
           1 instance(s) of cell ‘rnor2’
           2 instance(s) of cell ‘rnor1’
           3 instance(s) of cell ‘rexor’
            (5) Cell: rnor3 [Lib: primitives]
            (5) Cell: rnor2 [Lib: primitives]
            (5) Cell: rnor1 [Lib: primitives]
            (5) Cell: rexor [Lib: user]
                3 instances
                 3 instance(s) of cell ‘rnor2’
                  (6) Cell: rnor2 [Lib: primitives]
      (4) Cell: counteru [Lib: user]
          2 instances
           1 instance(s) of cell ‘rnor1’
           1 instance(s) of cell ‘cou’
            (5) Cell: rnor1 [Lib: primitives]
            (5) Cell: cou [Lib: user]
                17 instances
                 5 instance(s) of cell ‘rnor3’
                 4 instance(s) of cell ‘rnor2’
                 1 instance(s) of cell ‘rnor4’
                 4 instance(s) of cell ‘rnor1’
                 3 instance(s) of cell ‘dt1sqn’
                  (6) Cell: rnor3 [Lib: primitives]
                  (6) Cell: rnor2 [Lib: primitives]
                  (6) Cell: rnor4 [Lib: primitives]
                  (6) Cell: rnor1 [Lib: primitives]
                  (6) Cell: dt1sqn [Lib: user]
                      6 instances
                       2 instance(s) of cell ‘rnor1’
                       2 instance(s) of cell ‘rnor2’
                       2 instance(s) of cell ‘rnor3’
                        (7) Cell: rnor1 [Lib: primitives]
                        (7) Cell: rnor2 [Lib: primitives]
                        (7) Cell: rnor3 [Lib: primitives]
(3) Cell: selunit3 [Lib: user]
    14 instances
     8 instance(s) of cell ‘rnor1’
     3 instance(s) of cell ‘rnor2’
     1 instance(s) of cell ‘rnor4’
```

```
      1 instance(s) of cell 'valuecomp'
      1 instance(s) of cell 'counter'
       (4) Cell: rnor1 [Lib: primitives]
       (4) Cell: rnor2 [Lib: primitives]
       (4) Cell: rnor4 [Lib: primitives]
       (4) Cell: valuecomp [Lib: user]
           7 instances
            1 instance(s) of cell 'rnor3'
            1 instance(s) of cell 'rnor2'
            2 instance(s) of cell 'rnor1'
            3 instance(s) of cell 'rexor'
             (5) Cell: rnor3 [Lib: primitives]
             (5) Cell: rnor2 [Lib: primitives]
             (5) Cell: rnor1 [Lib: primitives]
             (5) Cell: rexor [Lib: user]
                 3 instances
                  3 instance(s) of cell 'rnor2'
                   (6) Cell: rnor2 [Lib: primitives]
       (4) Cell: counter [Lib: user]
           1 instances
            1 instance(s) of cell 'cou'
             (5) Cell: cou [Lib: user]
                 17 instances
                  5 instance(s) of cell 'rnor3'
                  4 instance(s) of cell 'rnor2'
                  1 instance(s) of cell 'rnor4'
                  4 instance(s) of cell 'rnor1'
                  3 instance(s) of cell 'dt1sqn'
                   (6) Cell: rnor3 [Lib: primitives]
                   (6) Cell: rnor2 [Lib: primitives]
                   (6) Cell: rnor4 [Lib: primitives]
                   (6) Cell: rnor1 [Lib: primitives]
                   (6) Cell: dt1sqn [Lib: user]
                       6 instances
                        2 instance(s) of cell 'rnor1'
                        2 instance(s) of cell 'rnor2'
                        2 instance(s) of cell 'rnor3'
                         (7) Cell: rnor1 [Lib: primitives]
                         (7) Cell: rnor2 [Lib: primitives]
                         (7) Cell: rnor3 [Lib: primitives]
(3) Cell: encoder [Lib: user]
     30 instances
      9 instance(s) of cell 'rnor2'
      13 instance(s) of cell 'rnor1'
      6 instance(s) of cell 'rnor4'
      1 instance(s) of cell 'rnor3'
      1 instance(s) of cell 'rnor5'
       (4) Cell: rnor2 [Lib: primitives]
       (4) Cell: rnor1 [Lib: primitives]
       (4) Cell: rnor4 [Lib: primitives]
       (4) Cell: rnor3 [Lib: primitives]
       (4) Cell: rnor5 [Lib: primitives]
(3) Cell: sftlogic [Lib: user]
```

```
        62 instances
         35 instance(s) of cell 'rnor2'
         19 instance(s) of cell 'rnor1'
         8 instance(s) of cell 'dt1rq'
          (4) Cell: rnor2 [Lib: primitives]
          (4) Cell: rnor1 [Lib: primitives]
          (4) Cell: dt1rq [Lib: user]
              6 instances
               4 instance(s) of cell 'rnor2'
               1 instance(s) of cell 'rnor1'
               1 instance(s) of cell 'rnor3'
                (5) Cell: rnor2 [Lib: primitives]
                (5) Cell: rnor1 [Lib: primitives]
                (5) Cell: rnor3 [Lib: primitives]
    (3) Cell: adder [Lib: user]
        16 instances
         4 instance(s) of cell 'rnor1'
         8 instance(s) of cell 'rnor2'
         2 instance(s) of cell 'rexor'
         2 instance(s) of cell 'subadder'
          (4) Cell: rnor1 [Lib: primitives]
          (4) Cell: rnor2 [Lib: primitives]
          (4) Cell: rexor [Lib: user]
              3 instances
               3 instance(s) of cell 'rnor2'
                (5) Cell: rnor2 [Lib: primitives]
          (4) Cell: subadder [Lib: user]
              50 instances
               17 instance(s) of cell 'rnor2'
               3 instance(s) of cell 'rnor4'
               4 instance(s) of cell 'rnor3'
               20 instance(s) of cell 'rnor1'
               4 instance(s) of cell 'rexor'
               2 instance(s) of cell 'rnor5'
                (5) Cell: rnor2 [Lib: primitives]
                (5) Cell: rnor4 [Lib: primitives]
                (5) Cell: rnor3 [Lib: primitives]
                (5) Cell: rnor1 [Lib: primitives]
                (5) Cell: rexor [Lib: user]
                    3 instances
                     3 instance(s) of cell 'rnor2'
                      (6) Cell: rnor2 [Lib: primitives]
                (5) Cell: rnor5 [Lib: primitives]
    (3) Cell: selunit1 [Lib: user]
        13 instances
         1 instance(s) of cell 'rnor4'
         7 instance(s) of cell 'rnor1'
         3 instance(s) of cell 'rnor2'
         1 instance(s) of cell 'valuecomp'
         1 instance(s) of cell 'counter'
          (4) Cell: rnor4 [Lib: primitives]
          (4) Cell: rnor1 [Lib: primitives]
          (4) Cell: rnor2 [Lib: primitives]
```

```
        (4) Cell: valuecomp [Lib: user]
            7 instances
             1 instance(s) of cell 'rnor3'
             1 instance(s) of cell 'rnor2'
             2 instance(s) of cell 'rnor1'
             3 instance(s) of cell 'rexor'
              (5) Cell: rnor3 [Lib: primitives]
              (5) Cell: rnor2 [Lib: primitives]
              (5) Cell: rnor1 [Lib: primitives]
              (5) Cell: rexor [Lib: user]
                  3 instances
                   3 instance(s) of cell 'rnor2'
                    (6) Cell: rnor2 [Lib: primitives]
        (4) Cell: counter [Lib: user]
            1 instances
             1 instance(s) of cell 'cou'
              (5) Cell: cou [Lib: user]
                  17 instances
                   5 instance(s) of cell 'rnor3'
                   4 instance(s) of cell 'rnor2'
                   1 instance(s) of cell 'rnor4'
                   4 instance(s) of cell 'rnor1'
                   3 instance(s) of cell 'dt1sqn'
                    (6) Cell: rnor3 [Lib: primitives]
                    (6) Cell: rnor2 [Lib: primitives]
                    (6) Cell: rnor4 [Lib: primitives]
                    (6) Cell: rnor1 [Lib: primitives]
                    (6) Cell: dt1sqn [Lib: user]
                        6 instances
                         2 instance(s) of cell 'rnor1'
                         2 instance(s) of cell 'rnor2'
                         2 instance(s) of cell 'rnor3'
                          (7) Cell: rnor1 [Lib: primitives]
                          (7) Cell: rnor2 [Lib: primitives]
                          (7) Cell: rnor3 [Lib: primitives]
  (3) Cell: selunit2 [Lib: user]
      13 instances
       1 instance(s) of cell 'rnor4'
       7 instance(s) of cell 'rnor1'
       3 instance(s) of cell 'rnor2'
       1 instance(s) of cell 'valuecomp'
       1 instance(s) of cell 'counter'
        (4) Cell: rnor4 [Lib: primitives]
        (4) Cell: rnor1 [Lib: primitives]
        (4) Cell: rnor2 [Lib: primitives]
        (4) Cell: valuecomp [Lib: user]
            7 instances
             1 instance(s) of cell 'rnor3'
             1 instance(s) of cell 'rnor2'
             2 instance(s) of cell 'rnor1'
             3 instance(s) of cell 'rexor'
              (5) Cell: rnor3 [Lib: primitives]
              (5) Cell: rnor2 [Lib: primitives]
```

```
            (5) Cell: rnor1 [Lib: primitives]
            (5) Cell: rexor [Lib: user]
                3 instances
                 3 instance(s) of cell 'rnor2'
                  (6) Cell: rnor2 [Lib: primitives]
        (4) Cell: counter [Lib: user]
            1 instances
             1 instance(s) of cell 'cou'
              (5) Cell: cou [Lib: user]
                  17 instances
                   5 instance(s) of cell 'rnor3'
                   4 instance(s) of cell 'rnor2'
                   1 instance(s) of cell 'rnor4'
                   4 instance(s) of cell 'rnor1'
                   3 instance(s) of cell 'dt1sqn'
                    (6) Cell: rnor3 [Lib: primitives]
                    (6) Cell: rnor2 [Lib: primitives]
                    (6) Cell: rnor4 [Lib: primitives]
                    (6) Cell: rnor1 [Lib: primitives]
                    (6) Cell: dt1sqn [Lib: user]
                        6 instances
                         2 instance(s) of cell 'rnor1'
                         2 instance(s) of cell 'rnor2'
                         2 instance(s) of cell 'rnor3'
                          (7) Cell: rnor1 [Lib: primitives]
                          (7) Cell: rnor2 [Lib: primitives]
                          (7) Cell: rnor3 [Lib: primitives]
(3) Cell: selunit0 [Lib: user]
    12 instances
     6 instance(s) of cell 'rnor1'
     3 instance(s) of cell 'rnor2'
     1 instance(s) of cell 'rnor4'
     1 instance(s) of cell 'valuecomp'
     1 instance(s) of cell 'counter'
      (4) Cell: rnor1 [Lib: primitives]
      (4) Cell: rnor2 [Lib: primitives]
      (4) Cell: rnor4 [Lib: primitives]
      (4) Cell: valuecomp [Lib: user]
          7 instances
           1 instance(s) of cell 'rnor3'
           1 instance(s) of cell 'rnor2'
           2 instance(s) of cell 'rnor1'
           3 instance(s) of cell 'rexor'
            (5) Cell: rnor3 [Lib: primitives]
            (5) Cell: rnor2 [Lib: primitives]
            (5) Cell: rnor1 [Lib: primitives]
            (5) Cell: rexor [Lib: user]
                3 instances
                 3 instance(s) of cell 'rnor2'
                  (6) Cell: rnor2 [Lib: primitives]
      (4) Cell: counter [Lib: user]
          1 instances
           1 instance(s) of cell 'cou'
```

```
                    (5) Cell: cou [Lib: user]
                        17 instances
                         5 instance(s) of cell ‘rnor3’
                         4 instance(s) of cell ‘rnor2’
                         1 instance(s) of cell ‘rnor4’
                         4 instance(s) of cell ‘rnor1’
                         3 instance(s) of cell ‘dt1sqn’
                          (6) Cell: rnor3 [Lib: primitives]
                          (6) Cell: rnor2 [Lib: primitives]
                          (6) Cell: rnor4 [Lib: primitives]
                          (6) Cell: rnor1 [Lib: primitives]
                          (6) Cell: dt1sqn [Lib: user]
                              6 instances
                               2 instance(s) of cell ‘rnor1’
                               2 instance(s) of cell ‘rnor2’
                               2 instance(s) of cell ‘rnor3’
                                (7) Cell: rnor1 [Lib: primitives]
                                (7) Cell: rnor2 [Lib: primitives]
                                (7) Cell: rnor3 [Lib: primitives]
  (2) Cell: tbsel [Lib: user]
      23 instances
       10 instance(s) of cell ‘rnor2’
       12 instance(s) of cell ‘rnor1’
       1 instance(s) of cell ‘rnor4’
        (3) Cell: rnor2 [Lib: primitives]
        (3) Cell: rnor1 [Lib: primitives]
        (3) Cell: rnor4 [Lib: primitives]
  (2) Cell: rnor1 [Lib: primitives]
  (2) Cell: rexor [Lib: user]
      3 instances
       3 instance(s) of cell ‘rnor2’
        (3) Cell: rnor2 [Lib: primitives]
  (2) Cell: resout [Lib: user]
      51 instances
       21 instance(s) of cell ‘rnor1’
       30 instance(s) of cell ‘rnor2’
        (3) Cell: rnor1 [Lib: primitives]
        (3) Cell: rnor2 [Lib: primitives]
  (2) Cell: valcomp4 [Lib: user]
      5 instances
       4 instance(s) of cell ‘rexor’
       1 instance(s) of cell ‘rnor4’
        (3) Cell: rexor [Lib: user]
            3 instances
             3 instance(s) of cell ‘rnor2’
              (4) Cell: rnor2 [Lib: primitives]
        (3) Cell: rnor4 [Lib: primitives]
```

F.9 Cwheel1_0

```
(1) Top Cell: design [Lib: dp_design]ist]
    208 instances
```

```
24 instance(s) of cell '74als251'
6 instance(s) of cell '74as821'
2 instance(s) of cell '74ls688'
20 instance(s) of cell 'tant22u'
1 instance(s) of cell 'c'
2 instance(s) of cell '10x25'
1 instance(s) of cell 'zener'
1 instance(s) of cell 'diode'
4 instance(s) of cell '74als74'
1 instance(s) of cell '74ls540'
11 instance(s) of cell '2018b'
2 instance(s) of cell '74s38'
36 instance(s) of cell '74als253'
3 instance(s) of cell '6116lp'
3 instance(s) of cell '74s283'
31 instance(s) of cell '74als257'
18 instance(s) of cell '74als574'
9 instance(s) of cell '74as867'
5 instance(s) of cell '74ls85'
3 instance(s) of cell '82s09'
4 instance(s) of cell '74ls645_1'
4 instance(s) of cell '74ls541'
1 instance(s) of cell '74as257'
1 instance(s) of cell 'osc'
2 instance(s) of cell '22v10'
1 instance(s) of cell 'pls168_33'
6 instance(s) of cell '96'
1 instance(s) of cell '74as574'
1 instance(s) of cell 'r'
2 instance(s) of cell '9e_1k'
2 instance(s) of cell 'spst8'
 (2) Cell: 74als251 [Lib: dp_devices]
 (2) Cell: 74as821 [Lib: dp_devices]
 (2) Cell: 74ls688 [Lib: dp_devices]
 (2) Cell: tant22u [Lib: dp_devices]
 (2) Cell: c [Lib: dp_devices]
 (2) Cell: 10x25 [Lib: dp_devices]
 (2) Cell: zener [Lib: dp_devices]
 (2) Cell: diode [Lib: dp_devices]
 (2) Cell: 74als74 [Lib: dp_devices]
 (2) Cell: 74ls540 [Lib: dp_devices]
 (2) Cell: 2018b [Lib: dp_devices]
 (2) Cell: 74s38 [Lib: dp_devices]
 (2) Cell: 74als253 [Lib: dp_devices]
 (2) Cell: 6116lp [Lib: dp_devices]
 (2) Cell: 74s283 [Lib: dp_devices]
 (2) Cell: 74als257 [Lib: dp_devices]
 (2) Cell: 74als574 [Lib: dp_devices]
 (2) Cell: 74as867 [Lib: dp_devices]
 (2) Cell: 74ls85 [Lib: dp_devices]
 (2) Cell: 82s09 [Lib: dp_devices]
 (2) Cell: 74ls645_1 [Lib: dp_devices]
 (2) Cell: 74ls541 [Lib: dp_devices]
```

```
(2) Cell: 74as257 [Lib: dp_devices]
(2) Cell: osc [Lib: dp_devices]
(2) Cell: 22v10 [Lib: dp_devices]
(2) Cell: pls168_33 [Lib: dp_devices]
(2) Cell: 96 [Lib: dp_devices]
(2) Cell: 74as574 [Lib: dp_devices]
(2) Cell: r [Lib: dp_devices]
(2) Cell: 9e_1k [Lib: dp_devices]
(2) Cell: spst8 [Lib: dp_devices]
```

Appendix G

Example of a System Run

This appendix contains the results of the system for the analysis of the design 'multmilldesign' described in chapter 8 (partially). The hierarchy of this design is given in appendix F.6.

```
-- INDICATORS OF DESIGN COMPLICATION --

   1. Number of Cells: 17
   2. Number of Primitive Cells: 15
   3. Graph Complexity: 1.29
   4. Highest Depth Level: 3
   5. Instantiation Ratio: 11.41
   6. Average no. Ports per Cell: 6.18
   7. Average no. Paths Leading to a Cell: 1.38


-- INITIALISATION CYCLE --

   [1/17] Cell: ctinv1 [Library: cad042]
          Hierarchy Level: 3

          NetWork Analysis:
            NetWork Contents: flat cell

          Plausible Cell Types:
            - Logic Type: [combinational,sequential,non]
            - Abstraction Level: [bit,gate,transistor]
            - DataFlow Type: [modifier,transporter,non]
            - Cell Purpose: [operator,storage,control,switch,non]

          Max. No. of Knowledge-extraction Plans: 33
          Plans Derived from Knowledge-extraction: 33
          Duplicated plans: 32

          Knowledge Plan Matches:
            - Class Model: inverter (0.80)
          -- Matching with Class Model [inverter] (0.80) is best

          No. of models: 2
          No. of plans: 3
```

```
       Solution plan/models ignored: 0

[2/17] Cell: split [Library: netlibrary]
       Hierarchy Level: 3

       NetWork Analysis:
         NetWork Contents: flat cell

       Plausible Cell Types:
         - Logic Type: [combinational,sequential,non]
         - Abstraction Level: [bit,gate,transistor]
         - DataFlow Type: [modifier,transporter,non]
         - Cell Purpose: [operator,storage,control,switch,non]

       Max. No. of Knowledge-extraction Plans: 35
       Plans Derived from Knowledge-extraction: 35
       Duplicated plans: 34

       Knowledge Plan Matches:
         - Class Model: inverter (0.80)
       -- Matching with Class Model [inverter] (0.80) is best

       No. of models: 2
       No. of plans: 3

       Solution plan/models ignored: 0

[3/17] Cell: ctnand2 [Library: cad042]
       Hierarchy Level: 3

       NetWork Analysis:
         NetWork Contents: flat cell

       Plausible Cell Types:
         - Logic Type: [combinational,sequential,non]
         - Abstraction Level: [bit,gate,transistor]
         - DataFlow Type: [modifier,transporter,non]
         - Cell Purpose: [operator,storage,control,switch,non]

       Max. No. of Knowledge-extraction Plans: 13
       Plans Derived from Knowledge-extraction: 13
       Duplicated plans: 12

       Knowledge Plan Matches:
         - Class Model: nand (0.86)
         - Class Model: nor (0.80)
         - Class Model: or (0.80)
         - Class Model: exor (0.80)
         - Class Model: and (0.80)
         - Class Model: flip_flop (0.50)
         - Class Model: adder (0.50)
       -- Matching with Class Model [nand] (0.86) is best
```

```
       No. of models: 8
       No. of plans: 9

       Solution plan/models ignored: 1
                       .
                       .
                       .

[15/17] Cell: multadd [Library: netlibrary]
        Hierarchy Level: 2

        NetWork Analysis:
          NetWork Contents:
             - Sub-cell: split [Library: netlibrary]
               Number of Instances: 3
             - Sub-cell: ctnand4 [Library: cad042]
               Number of Instances: 3
             - Sub-cell: ctexor [Library: cad042]
               Number of Instances: 2
             - Sub-cell: ctnand2 [Library: cad042]
               Number of Instances: 2
             - Sub-cell: ctinv1 [Library: cad042]
               Number of Instances: 2
             - Sub-cell: ctnand3 [Library: cad042]
               Number of Instances: 4
          Type of Network: combinational network with no feed-back loops
          Levels of the network:
             1- [xxxgate3,xxxgate6,xxxgate4,xxxgate16,xxxgate2,xxxgate1]
             2- [xxxgate9,xxxgate5,xxxgate14,xxxgate15,xxxgate10,xxxgate11]
             3- [xxxgate8,xxxgate12,xxxgate13]
             4- [xxxgate7]
          Abstraction Level of Sub-Cells: [gate]

        Plausible Cell Types:
          - Logic Type: [combinational]
          - Abstraction Level: [vector,bit]
          - DataFlow Type: [modifier,transporter]
          - Cell Purpose: [operator,control,switch]

        Max. No. of Knowledge-extraction Plans: 6
        Plans Derived from Knowledge-extraction: 6
        Duplicated plans: 5

        Knowledge Plan Matches:
        -- No Matches
           (1 incomplete model considered)

        No. of models: 1
        No. of plans: 2

        Solution plan/models ignored: 0

[16/17] Cell: ctfadd [Library: cad042]
```

```
        Hierarchy Level: 2

        NetWork Analysis:
          NetWork Contents: flat cell

        Plausible Cell Types:
          - Logic Type: [combinational,sequential]
          - Abstraction Level: [vector,bit]
          - DataFlow Type: [modifier,transporter]
          - Cell Purpose: [operator,storage,control,switch]

        Max. No. of Knowledge-extraction Plans: 13
        Plans Derived from Knowledge-extraction: 13
        Duplicated plans: 7

        Knowledge Plan Matches:
          - Class Model: counter (0.70)
          - Class Model: comparator (0.62)
          - Class Model: decoder (0.56)
          - Class Model: adder (0.55)
          - Class Model: encoder (0.50)
          - Class Model: delay_latch (0.50)
          - Class Model: demultiplexer (0.46)
        -- Matching with Class Model [counter] (0.70) is best

        No. of models: 8
        No. of plans: 14

        Solution plan/models ignored: 249

[17/17] Cell: multmill [Library: netlibrary]
        Hierarchy Level: 1

        NetWork Analysis:
          NetWork Contents:
             - Sub-cell: ctinv4 [Library: cad042]
               Number of Instances: 13
             - Sub-cell: ctinv1 [Library: cad042]
               Number of Instances: 30
             - Sub-cell: split [Library: netlibrary]
               Number of Instances: 31
             - Sub-cell: ct2anor [Library: cad042]
               Number of Instances: 11
             - Sub-cell: ctnand2 [Library: cad042]
               Number of Instances: 10
             - Sub-cell: ctnor2 [Library: cad042]
               Number of Instances: 32
             - Sub-cell: ctaoi [Library: cad042]
               Number of Instances: 1
             - Sub-cell: ctnor3 [Library: cad042]
               Number of Instances: 4
             - Sub-cell: ctoai [Library: cad042]
               Number of Instances: 3
```

```
        - Sub-cell: ctnand3 [Library: cad042]
          Number of Instances: 8
        - Sub-cell: ct2onand [Library: cad042]
          Number of Instances: 11
        - Sub-cell: ctexor [Library: cad042]
          Number of Instances: 3
        - Sub-cell: ctnor4 [Library: cad042]
          Number of Instances: 2
        - Sub-cell: ctnand4 [Library: cad042]
          Number of Instances: 3
        - Sub-cell: multadd [Library: netlibrary]
          Number of Instances: 4
        - Sub-cell: ctfadd [Library: cad042]
          Number of Instances: 12
     Type of Network: network with sequential sub-cells
     Abstraction Level of Sub-Cells: [vector,bit,gate]

   Plausible Cell Types:
     - Logic Type: [sequential]
     - Abstraction Level: [processor,vector]
     - DataFlow Type: [modifier,transporter]
     - Cell Purpose: [operator,storage,control,switch,processor,transducer]

   Max. No. of Knowledge-extraction Plans: 2
   Plans Derived from Knowledge-extraction: 2
   Duplicated plans: 1

   Knowledge Plan Matches:
   -- No Matches
      (1 incomplete model considered)

   No. of models: 1
   No. of plans: 2

   Solution plan/models ignored: 0

-- Selection of Cell Models --

   - Candidate Sets:
       Cell ctinv1: 2 sets (0 failed, 2 new)
       Cell split: 2 sets (0 failed, 2 new)
       Cell ctnand2: 8 sets (0 failed, 8 new)
       Cell ctnand3: 11 sets (0 failed, 11 new)
       Cell ctexor: 8 sets (0 failed, 8 new)
       Cell ctnand4: 9 sets (0 failed, 9 new)
       Cell ctinv4: 2 sets (0 failed, 2 new)
       Cell ct2anor: 9 sets (0 failed, 9 new)
       Cell ctnor2: 8 sets (0 failed, 8 new)
       Cell ctaoi: 11 sets (0 failed, 11 new)
       Cell ctnor3: 11 sets (0 failed, 11 new)
       Cell ctoai: 11 sets (0 failed, 11 new)
       Cell ct2onand: 9 sets (0 failed, 9 new)
       Cell ctnor4: 8 sets (0 failed, 8 new)
```

```
        Cell multadd: 25344 sets (0 failed, 25344 new)
        Cell ctfadd: 8 sets (0 failed, 8 new)
        Cell multmill: 2797938671616 sets (0 failed, 2797938671616 new)

    - Select Best Candidate Set
        Candidate Solution Set:
          - ctfadd: 1
          - multadd: 1
          - ctnand4: 1
          - ctnor4: 1
          - ctexor: 1
          - ct2onand: 1
          - ctnand3: 1
          - ctoai: 1
          - ctnor3: 1
          - ctaoi: 1
          - ctnor2: 1
          - ctnand2: 1
          - ct2anor: 1
          - split: 1
          - ctinv1: 1
          - ctinv4: 1
          - multmill: 1
        Design Evaluation: 0.450616

-- K-PROPAGATION/K-GENERATION CYCLES --

   Reasoning Cycle No: 2

   -- Model-based Reasoning --

        Failed Set for Cell multadd: 1-[1,1,1,1,1,1]
        Two situation Cells with Same Model:
        cell 'ctinv1' and cell 'split' (model: inverter).

        Failed Set for Cell multmill: 1-[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
        Two situation Cells with Same Model:
        cell 'split' and cell 'ctinv1' (model: inverter).

        Failed Set for Cell multmill: 1-[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
        Two situation Cells with Same Model:
        cell 'split' and cell 'ctinv4' (model: inverter).

        Failed Set for Cell multmill: 1-[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
        Two situation Cells with Same Model:
        cell 'ctinv1' and cell 'ctinv4' (model: inverter).
                       .
                       .
                       .

        Failed Set for Cell multmill: 1-[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
        Two situation Cells with Same Model:
        cell 'ctnor3' and cell 'ctaoi' (model: nand).
```

```
    Failed Set for Cell multmill: 1-[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    Two situation Cells with Same Model:
    cell `ctnor2' and cell `ctnand2' (model: nand).

    Failed Set for Cell multmill: 1-[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    Two situation Cells with Same Model:
    cell `ctexor' and cell `ctoai' (model: exor).

- Select Best Candidate Set
    Candidate Solution Set:
      - ctfadd: 1
      - multadd: 1
      - ctnand4: 4
      - ctnor4: 4
      - ctexor: 1
      - ct2onand: 1
      - ctnand3: 1
      - ctoai: 5
      - ctnor3: 4
      - ctaoi: 4
      - ctnor2: 5
      - ctnand2: 1
      - ct2anor: 1
      - split: 1
      - ctinv1: 2
      - ctinv4: 2
      - multmill: 1
    Design Evaluation: 0.398770

    Failed Set for Cell multmill: 1-[2,2,1,1,1,5,4,4,5,1,1,1,4,4,1,1]
    Two situation Cells with Same Model:
    cell `ctnand4' and cell `ctnor4' (model: and).

    Failed Set for Cell multmill: 1-[2,2,1,1,1,5,4,4,5,1,1,1,4,4,1,1]
    Two situation Cells with Same Model:
    cell `ctoai' and cell `ctnor3' (model: and).

    Failed Set for Cell multmill: 1-[2,2,1,1,1,5,4,4,5,1,1,1,4,4,1,1]
    Two situation Cells with Same Model:
    cell `ctoai' and cell `ctaoi' (model: and).

    Failed Set for Cell multmill: 1-[2,2,1,1,1,5,4,4,5,1,1,1,4,4,1,1]
    Two situation Cells with Same Model:
    cell `ctnor3' and cell `ctaoi' (model: and).

- Select Best Candidate Set
    Candidate Solution Set:
      - ctfadd: 1
      - multadd: 1
      - ctnand4: 4
      - ctnor4: 3
      - ctexor: 1
```

```
    - ct2onand: 1
    - ctnand3: 1
    - ctoai: 4
    - ctnor3: 3
    - ctaoi: 4
    - ctnor2: 5
    - ctnand2: 1
    - ct2anor: 1
    - split: 1
    - ctinv1: 2
    - ctinv4: 2
    - multmill: 1
  Design Evaluation: 0.398770
                .
                .
                .

-- Recognition-targeted Reasoning --

   [1/17] Cell: ctinv4 [Library: cad042]
          Hierarchy Level: 2

          NetWork Analysis:
            NetWork Contents: flat cell

          Plausible Cell Types:
            - Logic Type: [combinational,sequential,non]
            - Abstraction Level: [bit,gate,transistor]
            - DataFlow Type: [modifier,transporter,non]
            - Cell Purpose: [operator,storage,control,switch,non]

          Plans Derived from Model-based Reasoning: 13
          Duplicated plans: 12
          Already existing plans: 1

          Knowledge Plan Matches:
          -- No Matches

          No. new models considered: 0 (No. of models: 2)
          No. new plans 0 (No. of plans 3)

          Existing solution plans: 0
          Existing models: 0
          Solution plan/models ignored: 0

   [2/17] Cell: ctinv1 [Library: cad042]
          Hierarchy Level: 3

          NetWork Analysis:
            NetWork Contents: flat cell

          Plausible Cell Types:
            - Logic Type: [combinational,sequential,non]
```

```
      - Abstraction Level: [bit,gate,transistor]
      - DataFlow Type: [modifier,transporter,non]
      - Cell Purpose: [operator,storage,control,switch,non]

    Plans Derived from Model-based Reasoning: 30
    Duplicated plans: 23
    Already existing plans: 1

    Knowledge Plan Matches:
      - Class Model: inverter (0.88)

    No. new models considered: 1 (No. of models: 3)
    No. new plans 7 (No. of plans 10)

    Existing solution plans: 0
    Existing models: 0
    Solution plan/models ignored: 5
                .
                .
                .

-- Selection of Cell Models --

   - Candidate Sets:
       Cell multmill: 13949949911040 sets (3 failed, 11152011239424 new)
       Cell ctfadd: 9 sets (0 failed, 1 new)
       Cell multadd: 74880 sets (1 failed, 49536 new)
       Cell ctnand4: 13 sets (0 failed, 4 new)
       Cell ctnor4: 12 sets (0 failed, 4 new)
       Cell ctexor: 8 sets (0 failed, 0 new)
       Cell ct2onand: 9 sets (0 failed, 0 new)
       Cell ctnand3: 15 sets (0 failed, 4 new)
       Cell ctoai: 11 sets (0 failed, 0 new)
       Cell ctnor3: 11 sets (0 failed, 0 new)
       Cell ctaoi: 11 sets (0 failed, 0 new)
       Cell ctnor2: 8 sets (0 failed, 0 new)
       Cell ctnand2: 8 sets (0 failed, 0 new)
       Cell ct2anor: 9 sets (0 failed, 0 new)
       Cell split: 2 sets (0 failed, 0 new)
       Cell ctinv1: 3 sets (0 failed, 1 new)
       Cell ctinv4: 2 sets (0 failed, 0 new)

   - Select Best Candidate Set
       Candidate Solution Set:
         - ctfadd: 9
         - multadd: 1
         - ctnand4: 10
         - ctnor4: 12
         - ctexor: 1
         - ct2onand: 2
         - ctnand3: 12
         - ctoai: 2
         - ctnor3: 2
```

```
          - ctaoi: 2
          - ctnor2: 2
          - ctnand2: 1
          - ct2anor: 1
          - split: 2
          - ctinv1: 3
          - ctinv4: 2
          - multmill: 1
        Design Evaluation: 0.450679

Reasoning Cycle No: 3

-- Model-based Reasoning --

        Failed Set for Cell multmill: 1-[2,3,2,1,1,2,2,2,2,12,2,1,12,10,1,9]
        Two situation Cells with Same Model:
        cell ‘ct2onand’ and cell ‘ct2anor’ (model: nor).

        Failed Set for Cell multmill: 1-[2,3,2,1,1,2,2,2,2,12,2,1,12,10,1,9]
        Two situation Cells with Same Model:
        cell ‘ctnor3’ and cell ‘ctaoi’ (model: nor).

        Failed Set for Cell multmill: 1-[2,3,2,1,1,2,2,2,2,12,2,1,12,10,1,9]
        Two situation Cells with Same Model:
        cell ‘ctnand3’ and cell ‘ctoai’ (model: nand).

   - Select Best Candidate Set
        Candidate Solution Set:
          - ctfadd: 9
          - multadd: 1
          - ctnand4: 10
          - ctnor4: 12
          - ctexor: 1
          - ct2onand: 3
          - ctnand3: 12
          - ctoai: 3
          - ctnor3: 3
          - ctaoi: 2
          - ctnor2: 2
          - ctnand2: 1
          - ct2anor: 1
          - split: 2
          - ctinv1: 3
          - ctinv4: 2
          - multmill: 1
        Design Evaluation: 0.450679

        Failed Set for Cell multmill: 1-[2,3,2,1,1,2,2,3,3,12,3,1,12,10,1,9]
        Two situation Cells with Same Model:
        cell ‘ctoai’ and cell ‘ctaoi’ (model: nor).
                        .
                        .
                        .
```

```
  -- Recognition-targeted Reasoning --

     [1/17] Cell: ctinv4 [Library: cad042]
            Hierarchy Level: 2

            NetWork Analysis:
              NetWork Contents: flat cell

            Plausible Cell Types:
              - Logic Type: [combinational,sequential,non]
              - Abstraction Level: [bit,gate,transistor]
              - DataFlow Type: [modifier,transporter,non]
              - Cell Purpose: [operator,storage,control,switch,non]

            Plans Derived from Model-based Reasoning: 13
            Duplicated plans: 12
            Already existing plans: 1

            Knowledge Plan Matches:
            -- No Matches

            No. new models considered: 0 (No. of models: 2)
            No. new plans 0 (No. of plans 3)

            Existing solution plans: 0
            Existing models: 0
            Solution plan/models ignored: 0
                        .
                        .
                        .

-- INDICATORS OF PROCESSING COMPLEXITY --

  1. Processing Time:

     (a) Total Processing Time: 273.44 sec
     (b) Average Processing Time per Cell: 16.08 sec

  2. Reasoning Cycles:

     (a) Number of Reasoning Cycles: 3
     (b) Average Number of Productive Cycles per Cell: 1.3
     (c) Number of Failed Situation Sets: 6
     (d) Average Number of Failed Situation Sets: 3.00

  3. Number of Plans and Models:

     (a) Number of Plans Generated: 189
     (b) Average Number of Plans Generated per Cell: 11.12
     (c) Number of Models Generated: 133
     (d) Average Number of Models Generated per Cell: 7.82
```

```
   4. Heuristic Model Selection:

      (a) Number of Candidate Sets: 13949949911040
      (b) Average No of Situation Candidate Sets: 6974974992960.00
      (c) Number of Failed Sets: 186297413
      (d) Effectiveness of Set Selection: 1.00

-- EVALUATION OF KNOWLEDGE DERIVED --

   1. Average Confidence in the Models Selected: 0.66
   2. Confidence in the Design Representation: 0.45
   3. Average Complexity Deviation: 10 %

-- STATISTICS OF THE DESIGN CELLS --

   [1] Cell: multmill [Library: netlibrary]
       Hierarchy Level: 1

       Number of Models: 1
       Number of Plans: 3

       Number of Productive Cycles: 1
       Number of Candidate Situation Sets: 13949949911040
       Number of Failed Sets: 5
       Effectiveness of Set Selection: 1.00

       Evaluation of Selected Set: 0.451
       Complexity Deviation Factor: 0.31

   [2] Cell: ctinv4 [Library: cad042]
       Hierarchy Level: 2

       Primitive cell.

       Number of Models: 2
       Number of Plans: 3

       Number of Productive Cycles: 1

       Evaluation of Selected Model: 0.000
                         .
                         .
                         .

   [17] Cell: ctnand4 [Library: cad042]
        Hierarchy Level: 3

        Primitive cell.

        Number of Models: 13
        Number of Plans: 18

        Number of Productive Cycles: 2
```

```
     Evaluation of Selected Model: 0.941

-- NUMBER OF PLANS/MODELS PER CYCLE --

   Reasoning Cycle No: 1   -- No. Models: 119 - No. Plans: 158 --
   Reasoning Cycle No: 2   -- No. Models: 14 - No. Plans: 28 --
   Reasoning Cycle No: 3   -- No. Models: 0 - No. Plans: 3 --

-- HEURISTIC MODELS OF THE DESIGN CELLS --

   Model Name: ct2anor
   Model Types:
        Logic Type: combinational
        Abstraction Level: gate
        DataFlow Type: non
        Cell Purpose: non
   Model Function: nor (0.94)
   Model Interface:
        Port Group:
             Type: data_in
             Number of Signals: 4
             Port Collections:
                  Port Collection:
                       Port Name: [c,b,a,d]
                       Function: data (0.53)
                       Number of Signals: 4
        Port Group:
             Type: data_out
             Number of Signals: 1
             Port Collections:
                  Port Collection:
                       Port Name: f
                       Function: data (0.20)
                       Number of Signals: 1
   Model Data Flow Information: []
   Model Derived From:
        Library: cad042
        Cell: ct2anor
        View: netview
   Model Generated on: Mon Jun  7 20:57:29 1993
                       .
                       .
                       .

   Model Name: multmill
   Model Types:
        Logic Type: B
        Abstraction Level: E
        DataFlow Type: C
        Cell Purpose: D
   Model Function: A
   Model Interface:
```

```
      Port Group:
           Type: data_in
           Number of Signals: 16
           Port Collections:
                Port Collection:
                     Port Name: sa
                     Function: F
                     Number of Signals: 4
                Port Collection:
                     Port Name: y
                     Function: G
                     Number of Signals: 4
                Port Collection:
                     Port Name: dd
                     Function: H
                     Number of Signals: 4
                Port Collection:
                     Port Name: sb
                     Function: I
                     Number of Signals: 4
      Port Group:
           Type: data_out
           Number of Signals: 8
           Port Collections:
                Port Collection:
                     Port Name: xxxoput2
                     Function: J
                     Number of Signals: 1
                Port Collection:
                     Port Name: xxxoput4
                     Function: K
                     Number of Signals: 1
                Port Collection:
                     Port Name: xxxoput3
                     Function: L
                     Number of Signals: 1
                Port Collection:
                     Port Name: res
                     Function: M
                     Number of Signals: 4
                Port Collection:
                     Port Name: xxxoput1
                     Function: N
                     Number of Signals: 1
      Port Group:
           Type: control_in
           Number of Signals: 15
           Port Collections:
                Port Collection:
                     Port Name: ctrl
                     Function: control (0.90)
                     Number of Signals: 15
 Model Data Flow Information: []
```

```
Model Derived From:
     Library: netlibrary
     Cell: multmill
     View: netview
Model Generated on: Mon Jun  7 20:59:06 1993
```