Logics for digital circuit verification : theory, algorithms, and applications

DOI:
10.6100/IR520460

Document status and date:
Published: 01/01/1999

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Logics for Digital Circuit Verification

Theory, Algorithms, and Applications

G.L.J.M. Janssen

Logics for Digital Circuit Verification
Theory, Algorithms, and Applications

Logics for Digital Circuit Verification
Theory, Algorithms, and Applications

PROEFSCHRIFT

Ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 24 februari 1999 om 16.00 uur

door

Gradus Leonardus Johannes Maria Janssen

geboren te Oss

Dit proefschrift is goedgekeurd door de promotoren

prof.Dr.-Ing. J.A.G. Jess, en
prof.dr. J.C.M. Baeten

Copromotor:

dr.ir. C.A.J. van Eijk

Druk: Universiteitsdrukkerij Technische Universiteit Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Janssen, Gradus L.J.M.

Logics for digital circuit verification : theory, algorithms, and applications / by Gradus L.J.M. Janssen. - Eindhoven : Technische Universiteit Eindhoven, 1999.
Proefschrift. - ISBN 90-386-1560-4
NUGI 832
Trefw.: digitale geintegreerde schakelingen / digitale systemen ; CAD / correctheidsbewijzen / temporele logica / hardwarebeschrijving / sequentiele machines.
Subject headings: digital integrated circuits / logic CAD / formal verification / temporal logic / sequential circuits.

voor mijn vader

Summary

This thesis presents the results of investigating various logics with respect to their application to verification of digital hardware design. The approach highlights both the end-user aspects and the implementor's aspects.

The thesis is structured in 3 parts: part I discusses verification problems in the area of combinational circuits, part II focuses on sequential circuit verification, and part III presents the software tools that have been developed and discusses details of their implementations. Also, part III contains a number of test cases that exhibit the typical modelling of problems in terms of the investigated logics and shows how they are solved by the presented tools:

- bdd - a boolean function manipulation package
- ptl - a temporal logic satisfiability checker
- mu - a propositional μ-calculus tool
- bsn2veri - a combinational circuit equivalence checker
- bsn2mc - a Fair-CTL model checker

This thesis focuses on techniques for hardware verification. The approach is formal, i.e., mathematical theories will be presented that form the basis for modelling the hardware and reasoning about its behaviour. The work concentrates on decidable theories, for which algorithms exist that can be used to prove certain properties of the circuit. Central to this thesis are the application of the theory and the development of efficient algorithms.

Samenvatting

Dit proefschrift presenteert de resultaten van een onderzoek naar de toepassing van diverse logica's met betrekking tot de verificatie van digitale hardware. De aanpak belicht zowel de eindgebruikeraspecten als ook de aspecten t.a.v. de implementator.

Deze thesis is onderverdeeld in 3 delen: deel I beschouwt verificatieproblemen op het gebied van combinatorische circuits, deel II focusseert op verificatie van sequentiële circuits en deel III presenteert de software pakketten die zijn ontwikkeld en gaat in op de details van hun implementatie. Daarnaast presenteert deel III een aantal voorbeelden die de typische modellering van problemen in termen van de onderzochte logica's duidelijk maken en laat zien hoe deze opgelost worden met de ontwikkelde programma's:

- bdd - een pakket voor manipulatie van boolse functies
- ptl - een *satisfiability checker* voor LTL temporele logica
- mu - een programma voor propositionele μ-calculus
- bsn2veri - een equivalentie checker voor combinatorische circuits
- bsn2mc - een *model checker* voor Fair-CTL

Deze thesis richt zich vooral op technieken voor hardware verificatie. De aanpak is formeel, d.w.z. mathematische theorieën worden behandeld die de basis vormen voor het modelleren van hardware en het redeneren over het gedrag ervan. We concentreren ons op beslisbare theorieën, waarvoor geldt dat er algoritmen bestaan die gebruikt kunnen worden om zekere eigenschappen van het circuit te bewijzen. Centraal staan de applicatie van de theorie en de ontwikkeling van efficiënte algoritmen.

Contents

Chapter 1

General Introduction

1.1 Introduction and thesis overview

This thesis is about verification. The purpose of this general introduction is to set the context for our meaning of the term 'verification' and to argue why verification is an important subject in the realm of digital circuit design. We also give an overview of the contents of the thesis. Lastly, my personal history of work in this field is narrated and acknowledgements to people that influenced me are made.

In many industrial production areas, quality is of utmost importance. And quality entails measurements and standards: the standards set the goals and by measurements it is verified whether these goals are met. In the design of complex electronic circuits it is no longer possible to ensure their correctness by mere visual inspection (the features on a chip are simply too small to see with the naked eye). But even the design data or blueprints, i.e., the schematics, hardware description language texts, and layouts, contain so much detail that inspection by a human designer becomes a truly Herculean job. It is not just the complexity, i.e., large scale of integration, that causes problems; also the intrinsic 'go-or-no-go' characteristic of the subject matter makes it hard to devise good and simple tests. The effect of a single wrong connection during design, or a tiny oversight (read: bug) by the programmer of a logic synthesis tool, may render a whole batch of wafers useless. The sooner the flaw is discovered the better, and the cheaper. Not to mention all the things that can go wrong during the actual chip fabrication process. However, that is not the subject of this thesis.

A first step towards a successful production of a chip is that its design is functionally correct. Correctness is not an absolute notion. One can only meaningfully talk about the correctness of a design with respect to a some predefined specification (the standard). By means of measurement or testing it is possible to assess the quality of a design. But, (non-exhaustive) testing can only indicate the presence of errors; it cannot guarantee their absence. This is a slightly rephrased quotation from Edsger Dijkstra [Dijks76] who was then talking about the correct construction of computer programs. In this dissertation we investigate the applicability of formal methods in verifying the functional correctness of designs for digital circuits. For that we need two things: a mathematical model for the behaviour of the circuit and a theory that allows 'mechanical' reasoning within that model. We use the word 'functional' to stress a certain narrowing of, or abstraction from, all possible issues involved; it is the behaviour that fits a certain model of the circuit that we are interested in, and it is with respect to the models that we tailor the correctness problems and their solutions.

A major part of this thesis is concerned with the practical issues that arise when implementing these formal methods as a computer program suitable for use by electronic circuit designers. Several computer-aided verification tools were developed as part of this thesis work:

— bdd - a package for logic function manipulation

— ptl - a propositional temporal logic satisfiability checker

— mu - implements an extended propositional μ-calculus

— bsn2veri - a combinational circuit equivalence checker

— bsn2mc - a Fair-CTL model checker

Undoubtedly, the design process of computer algorithms and their implementation in a certain programming language parallels the design of integrated circuits. Many mathematical concepts and techniques are similar, if not the same, at least when regarded at a certain level of abstraction. (There is no equivalent to a physical MOS transistor in a programming language; but when modelled as a switch, the ensuing logic circuit may well be simulated by if-then-else statements). In fact, computer sciencists recognized this and not surprisingly they were the ones that coined the phrase 'VLSI programming' [Niess88]. True, much of the earlier work on formal verification was initiated by mathematicians and computer scientists. Unfortunately, this has led to the situation where much of this work was not accepted by the electrical engineering community and the CAD tool makers. Also, the goals they set out do not always coincide with the needs of a hardware designer; there often exists a large gap between theoretical results and useful applications. We should not forget that the main purpose of computer aided design is to actually help a designer in getting his job done and not to put up yet another barrier for him to cross.

This thesis focuses on techniques for hardware verification. The approaches are formal, i.e., mathematical theories will be presented that form the basis for modelling the hardware and reasoning about its behaviour. Moreover, we concentrate on decidable theories, this means that algorithms exist that can be used to prove certain properties of the circuit. Central to this thesis are the applications of the theory and the development of efficient algorithms.

This thesis consists of three parts: part I deals with combinational circuits, part II treats sequential circuits, and part III presents practical approaches to solving the problems raised in the preceding parts. It is valid to say that the first two parts mainly define a number of important verification problems and the necessary theory to describe them, and that in the last part of this thesis choices are made as to how to practically solve those problems in the form of computer programs. The last part also presents examples and quantitative data of experiments. The three parts are entitled:

Part I: Verification of Combinational Circuits.
In this part the relations between combinational circuits, propositional logic, and canonical representations of switching functions (DNF, BDD) are made explicit.

Part II: Verification of Circuits with Memory.
Here we extend the combinational circuit model to include time-dependent behaviour. This leads to the introduction of the propositional temporal logics LTL and CTL, and an even more powerful system: μ-calculus.

Part III: Programs and Examples.
The theory presented in the first two parts culminated in the development of several verification tools: the ptl program for temporal logic, the mu program for μ-calculus, the bsn2veri program for combinational circuit verification based on a hardware description language, and last but not least a BDD package that forms the core of all the aforementioned programs. Part III will discuss some of the more interesting implementational aspects of these programs, in sofar as they were not already covered in the preceding parts, and, more importantly, we will discuss how to effectively use those programs by studying a number of example problems.

1.2 Background and acknowledgements

It has taken me quite some time to prepare this thesis. The best excuse I can offer is that the field of CAD for electronic circuits is so exciting that it is hard for me to tear myself loose from the daily work and sit back and reflect on my own specialized field of hardware verification, let alone to do nothing else but concentrate on writing a whole thesis. The 'trouble' is that my interests are rather diverse, and my character forces me to not be satisfied until a new idea or theory or algorithm is fully understood, which often results in writing my own program for the particular problem just to see what is involved and thereby creating a framework for exploring my own ideas and hopefully to be able to come up with improvements. So, over the years I have developed quite a few programs on a variety of subjects. I once wrote some language development tools, like an LL(1) checker and a program that draws syntax diagrams. I was involved in the design of the Eindhoven Schematic Entry tool "Escher" under the NELSIS project; a program that was later enhanced by some of my students to mature as "Escher+" [Janss89], and that now is in the able hands of Hans Fleurkens who turned it into a very flexible and general graphic design entry and user-interface. Within the same project, I developed the hardware description language NDML to be used with the piece-wise linear simulator also developed in our group [Janss86]. My first contact with temporal logic was when I was working at Philips Telecommunications Industry in Hilversum where Ron Koymans was doing his master thesis work under the guidance of (now Prof.) Jan Vytopil. Their goal was to formalize the message passing semantics of the CCITT programming language CHILL. I was intrigued by the many obscure symbols and seemingly heavy mathematics they were using in their writings. When a few years later I was confronted with the work of Amir Pnueli and Zohar Manna [Manna81] on propositional temporal logic, I decided to write my own satisfiability checker for LTL and apply it to the verification of digital systems. Fortunately, this work became part of the ASCIS project, which gave me a willing audience to promote my ideas and results [Raner93, Janss92]. The initial driving force for my work on temporal logic was fueled by a challenge proposed by Prof. Lars Philipson of Lund University, Sweden [Phili89]. His skepticism towards the applicability of temporal logic to prove correctness of state machines culminated in a hectic E-mail conversation, ending in an invitation to hop over to Lund and explain it all in person. (Those were the good old days.)

I am obliged to thank many people that in one way or other have 'educated' me to become the person that I am now. First of all I want to thank Professor Jochen Jess who has been, and still is, a most fair boss, an inspiring tutor, and amiable colleague (and a good drummer). I praise myself lucky to have the opportunity to work with so many good-hearted, honest, hard working, and outstanding colleagues in the Design Automation Group: Lia, Frans, Jos, Michel, en Oege, thanks. My office mates over the years deserve special mention: Hans Zuidam, André Slenter, Ed Huijbregts, and Pieter, a.k.a. Tiggr, Schoenmakers. They never tired of having to listen to my ranting about bugs, troubles, errors, frustrations, et cetera. I thank all my students who have helped me with the implementations of some of the algorithms, and doing much of the testing and debugging. The most appreciated, but for them less obvious, side-effect was that they forced me to keep ahead of them and to acquire an intimate understanding of the subject at hand. They also kept me young, in spirit that is. I salute all the doctoral students that I saw arrive in our group, do their excellent work, and leave again (*veni, vidi, vici*). I cannot resist to single out Jose Pineda who is a dear, and now unfortunately distant, friend, and Gjalt de Jong who was my sparring partner in the verification arena from the start. He handed over the towel to a knowledgeable substitute in the person of Koen van Eijk. Special thanks go to the members of the

BSN group at IBM T.J. Watson Research Center and their manager Marshall Schor (who generously hosted me during the past five summers), and to Ton Kostelijk and Willem Rovers of Philips Research. I hope we can continue our cooperation in the future.

Part I

Verification of Combinational Circuits

The first part of this thesis consists of three chapters. Chapter 2 defines combinational circuits and states the combinational circuit equivalence problem. A more general problem that arises when correspondences between circuit inputs and outputs are not known is briefly discussed. It presents an elegant formulation of a general-purpose permutator circuit. Chapter 3 introduces propositional logic and derives an important representation for its expressions: disjunctive normal form. This form will be used again in chapter 6 in the implementation of an LTL satisfiability checker. Chapter 4 concludes this part with an introduction to binary decision diagrams (BDDs). All programs presented in this thesis are based on these BDDs.

Chapter 2

Combinational Circuits

2.1 Introduction

In this chapter we define the notion of a combinational circuit and introduce an important verification problem, namely combinational circuit equivalence. We show how boolean functions can be associated with the network graph representation of a combinational circuit. It is assumed that the correspondences between inputs and outputs of the circuits to be compared are known beforehand. In the last section we drop this restriction and consider a more general verification problem.

2.2 Boolean functions and combinational circuits

Let $B = \{0, 1\}$ be the set of boolean truth values: 0 stands for false, and 1 stands for true. We consider functions $f: B^n \longrightarrow B, n \geq 0$, and more generally, m element vectors of such functions $F: B^n \longrightarrow B^m, m > 0$. Geometrically speaking we may say that F maps a point in boolean n space to a point in boolean m space. A real-life realization or implementation of F is called a combinational logic circuit. In a combinational circuit the 0 and 1 values are usually related to non-overlapping voltage intervals; the functions are realized by a network of primitive logic operator cells or gates. When the argument value $\underline{x}$ to the function F is supplied to the inputs of the network, the function value $\underline{y} = F(\underline{x})$ is observed at the outputs of the network. F is an abstract, mathematical model (black-box) for the behaviour of the combinational circuit. It does not necessarily tell us anything about its structure. Figure 2.1 illustrates the correspondence between a combinational circuit (left) and its black-box functional model (right). In fact, the logic circuit itself is a model for a transistor level circuit.

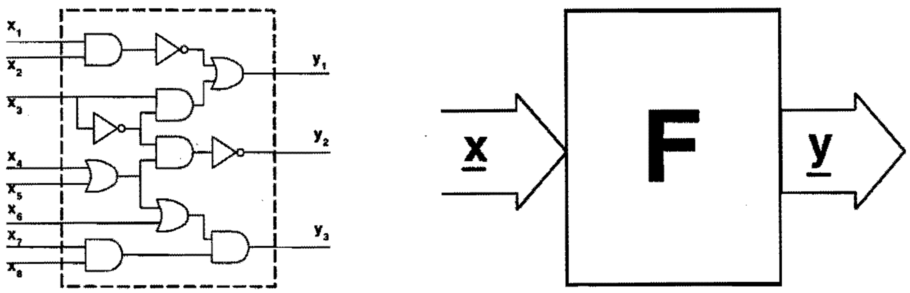


Figure 2.1. Schematic of combinational circuit and its black-box model.

A particular (vector component) function $f: B^n \longrightarrow B$ may be defined in various ways. Usually one introduces place-holder symbols, say $x_1, x_2, \cdots, x_n$, to represent arbitrary argument values and then defines the result $f(x_1, x_2, \cdots, x_n)$ of applying the function to those arguments by some logic expression involving the x_i's. For our example circuit in figure 2.1 we could write:

$$f_1(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = \neg(x_1 \wedge x_2) \vee (x_3 \wedge \neg x_3),$$
$$f_2(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = \neg(\neg x_3 \wedge (x_4 \vee x_5)),$$
$$f_3(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = (x_4 \vee x_5 \vee x_6) \wedge (x_7 \wedge x_8).$$

Intuitively, the meaning of the logic expressions on the right-hand side should be clear. In a following chapter we will formally define a language of logic expressions.

It should be obvious that the position of a place-holder variable in the argument list is very crucial; above this has been indicated using subscripts. Generally, $f(x_1, x_2) \neq f(x_2, x_1)$, and in order to compare two functions, no matter in what way they are represented, the correspondence between their sets of place-holder variables has to be known in advance. With this in mind, the following definition makes sense:

Definition 2.1 (Functional circuit equivalence).
Two combinational circuits C_1 and C_2 are said to be functionally equivalent if their corresponding black-box models F_1 and F_2 are identical (vector) functions.

In practice, this means that we assume the two circuits to have identical input and output labelling (symbols), and that the correspondence is simply defined as identity of these labels. We can now state the main problem that we will address in this chapter:

COMBINATIONAL CIRCUIT EQUIVALENCE
INSTANCE: Two combinational circuits C_1 and C_2 over the same sequence of inputs and outputs.

QUESTION: Are both circuits functionally equivalent? That is, do both circuits for all possible combinations of input values yield the same values at their corresponding outputs?

Note that the formulation of the above question already hints at a possible solution: exhaustive simulation. Once we have established a model for a combinational circuit suitable to automatic evaluation (= operational model), the equivalence problem can be solved by simulating both circuits for all possible combinations of input values, each time checking whether the corresponding output pairs evaluate to the same value. However, simulation is not the topic of this thesis; our goal is to develop verification algorithms that answer this question. Before we do so we first need to define a combinational circuit precisely, and show how we can attach a functional meaning to it. Then the equivalence problem can be rephrased in terms of the circuits' associated functions.

2.3 Boolean network

We define a boolean network as a graph: the vertices of the graph represent the gates of the circuit; the edges represent the connections from a gate output to other gates' inputs. More formally,

Definition 2.2 A boolean network is a vertex-labelled directed graph $G(V, E)$. The set of vertices V consists of the three non-overlapping sets Gates, Inputs, and Outputs, with

- Gates : a finite set of gates. Each gate is labelled by a logical operator symbol chosen from the set $\{\neg, \wedge, \vee\}$. Note that since the $\wedge$ and $\vee$ operators are commutative and associative it is not necessary to define an ordering on the incoming edges to such a vertex. $\neg$ gates have a single incoming edge; $\wedge$ and $\vee$ gates have 2 or more incoming edges.
- Inputs : a finite set of circuit inputs, also called the primary inputs. A primary input cannot have incoming edges.
- Outputs : a finite non-empty set of circuit outputs, also called the primary outputs. A primary output has a single incoming edge and no outgoing edges.
- The directed edges $E \subseteq (\text{Gates} \cup \text{Inputs}) \times (\text{Gates} \cup \text{Outputs})$ represent the connections between gates, primary inputs, and primary outputs. Note that edges between inputs and edges between outputs are not allowed. It is tempting to impose the requirement that the graph be acyclic. However, this is not a necessary condition for a circuit to be combinational [Malik93]. However, for simplicity we do here assume acyclicness.

Figure 2.2 shows a circuit schematic and its associated boolean network graph.

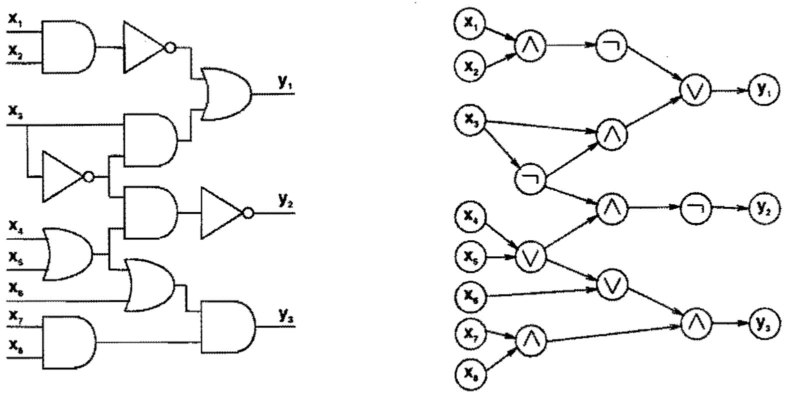


Figure 2.2. Circuit schematic and its boolean network graph.

The following additional definitions for a boolean network will prove to be useful:

Definition 2.3 The direct fanin of a vertex v is the set of its immediate predecessor vertices:

$$\mathrm{fanin}(v) = \{ u \mid (u, v) \in E \}.$$

Note that for a primary input x_i, $\mathrm{fanin}(x_i) = \emptyset$.

Definition 2.4 The direct fanout of a vertex v is the set of its immediate successor vertices:

$$\mathrm{fanout}(v) = \{ w \mid (v, w) \in E \}.$$

Note that for a primary output y_k, $\mathrm{fanout}(y_k) = \emptyset$.

We are now ready to define the meaning (semantics) of a combinational circuit. We do this by giving an inductive definition for its (black-box) boolean function vector F.

Definition 2.5 With each combinational circuit, represented by an acyclic boolean network graph according definition 2.2, we associate a vector of boolean functions $F = [f_1, f_2, \cdots, f_m]$, with $m = |\mathrm{Outputs}|$. We first impose a total ordering on the set of primary $\mathrm{Inputs} = \{x_1, x_2, \cdots, x_n\}$ and primary $\mathrm{Outputs} = \{y_1, y_2, \cdots, y_m\}$, e.g. by the indices. All component functions f_k of F will have the same domain B^n and co-domain B; we abbreviate $x_1, x_2, \cdots, x_n$ to X.

- Each primary input x_i is associated with a projection function $f_{x_i} = \lambda X. x_i$.
- Each gate g is associated with a function $f_g = \lambda X. \underset{u \in \mathrm{fanin}(g)}{\Phi} f_u(X)$, where Φ is the reduction operator corresponding to

the operator labelling the gate, e.g. if the label of g is $\wedge$ then $\Phi = \bigwedge$.

- With each primary output y_j we associate a function
  $f_{y_j} = f_{\text{fanin}(y_j)}$.
- The boolean functions f_k associated with the circuit are the functions we associate with the primary outputs y_k, thus
  $f_k = f_{y_k}$.

Example 2.1 Definition 2.5 applied to the circuit of figure 2.2 results in $F = [f_1, f_2, f_3]$, with

$$f_1 = f_{y_1} = \lambda x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 . (\neg(x_1 \wedge x_2)) \vee (x_3 \wedge (\neg x_3)),$$
$$f_2 = f_{y_2} = \lambda x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 . \neg((\neg x_3) \wedge (x_4 \vee x_5)), \text{ and}$$
$$f_3 = f_{y_3} = \lambda x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 . ((x_4 \vee x_5) \vee x_6) \wedge (x_7 \wedge x_8).$$

□ example 2.1

We conclude this subsection with a restatement of the equivalence problem.

COMBINATIONAL CIRCUIT EQUIVALENCE
INSTANCE: Two combinational circuits C_1 and C_2 represented by their boolean network graphs $G_1(\text{Gates}_1 \cup \text{Inputs} \cup \text{Outputs}, E_1)$ and G_2 $(\text{Gates}_2 \cup \text{Inputs} \cup \text{Outputs}, E_2)$ respectively.
QUESTION: Are both circuits functionally equivalent? That is, does $F_1 = F_2$ hold for the associated vectors of boolean functions F_1 and F_2?

Figure 2.3 presents this problem in the form of a picture.

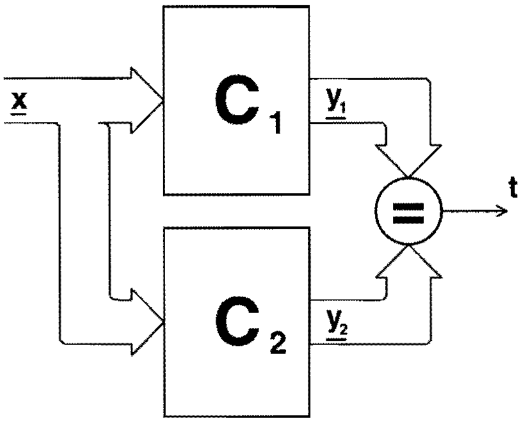


Figure 2.3. Combinational circuit equivalence: $C_1 = C_2$?

The single bit output represents the function $t = \lambda \underline{x} . (F_1(\underline{x}) = F_2(\underline{x}))$. The circuits are equivalent if and only if this function is a tautology, i.e., $\forall_{\underline{x} \in B^n} \; t(\underline{x}) = 1$.

2.4 Higher-Level descriptions

A representation of a combinational circuit by its boolean network graph is called a gate-level description. It is a convenient representation (encoding) for many problems relating to logic optimization and synthesis. The ratio of functionality to representation size at the gate level is quite small: to implement a simple arithmetic function over n-bit numbers requires a large number of gates. Thus we look for more efficient representations yielding a larger ratio. Basically, two approaches come to mind:

1. Introduce hierarchy. This will allows us to exploit the fact that in many designs the same subcircuits are used over and over again. In a hierarchical description a subcircuit representing a certain function is defined only once, and all its uses are references instead of copies. The effect is twofold: descriptions become more compact and algorithms processing them are faster (because of data sharing).
2. Introduce richer domains. Instead of having to express all functionality in terms of operations on boolean variables, we allow a designer to use enumerated types, arrays of booleans, various number types, records, et cetera. Each type comes with a convenient set of operations.

In many hardware description languages both approaches are combined. We will present such a language in chapter 10.

2.5 Other verification problems

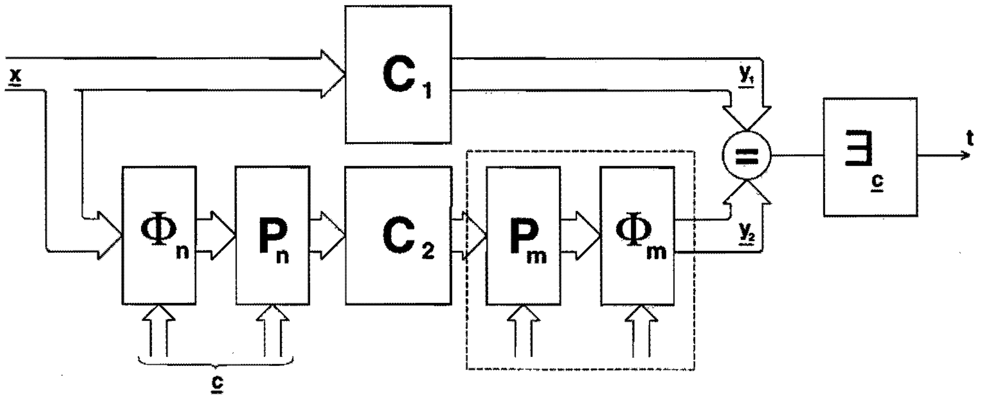


Figure 2.4. Phase and permutation independent comparison.

The problem of comparing two circuits for equivalence becomes much harder when the correspondences between their inputs and outputs are not known in advance. To make the equivalence question even more general, one could also drop the assumption that the phase (or polarity) of corresponding inputs and outputs is the same. By the latter we mean whether or not a certain signal is complemented with respect to a reference signal. In other words, we are considering equivalence modulo complementation: we might know that two inputs x_i and x_j correspond but they still might carry signal values that are each others complement, then $x_i = \neg x_j$ holds. We are facing the problem of phase and permutation independent boolean comparison, see e.g. [Mohnk93]. The general setting of this problem is depicted in figure 2.4.

Often the output correspondence is known so the dashed part in figure 2.4 may be left out. Φ_n stands for an n-bit phasor circuit; P_n is an n-bit permutator circuit. Both introduce a number of control inputs $\underline{c}$ that are to be smoothened out by the $\exists_{\underline{c}}$ block on the right. In fact what we are asking is whether an assignment to the control inputs $\underline{c}$ exists that makes the corresponding outputs of C_1 and C_2 identical for all assignments to the primary inputs $\underline{x}$. The phasor circuit is easily realized by XOR gates (see figure 2.5). The output of a 1-bit phasor equals its input when the control input c is low (0); the output is the complement of the input when c is high (1). A number of n control variables will be needed to generate the 2^n possible phase assignments to an n-bit input vector $\underline{x}$.

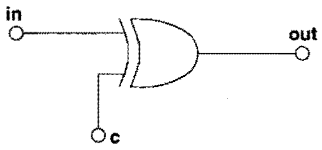


Figure 2.5. A 1-bit phasor circuit.

A permutator circuit causes its inputs to be connected to its outputs according to a certain permutation; in fact, it is a particular type of controlled switch. Its precise implementation will be explained shortly. Clearly, the permutator needs a minimum of $\lceil {}^2\log(n!)\rceil$ control variables. In [Corme90] it is suggested to implement a permutator using the same structure as a sorting network by replacing the comparator nodes by 2-input, 2-output switches. It can then be shown that a number of $c_n = n \cdot {}^2\log(n) - n/2$ control variables is required, with the restriction that n is a power of 2. Table 2.1 lists c_n for some practical values of n together with related quantities.

n	$\lceil {}^2\log(n)\rceil$	n!	$\lceil {}^2\log(n!)\rceil$	c_n
1	0	1	0	0
2	1	2	1	1
3	2	6	3	3
4	2	24	5	6
5	3	120	7	8
6	3	720	10	12

n	$\lceil {}^2\log(n) \rceil$	n!	$\lceil {}^2\log(n!) \rceil$	c_n
8	3	40320	16	20
16	4	$2.1\ 10^{13}$	45	56
32	5	$2.6\ 10^{35}$	118	144
64	6	$1.3\ 10^{89}$	296	352
128	7	$3.9\ 10^{215}$	717	832
256	8	$8.6\ 10^{506}$	1684	1920

Table 2.1. Number of control variables c_n in 'butterfly' permutator P_n.

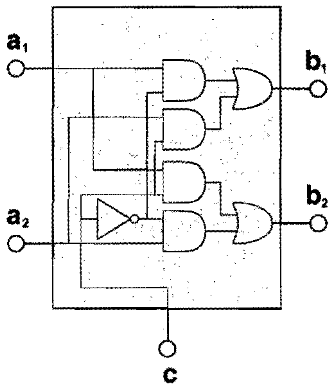
The general n-bit permutator circuit P_n can be inductively defined as follows.

- For the 1 bit case ($n = 1$) the identity is the only possible permutation and the permutator circuit P_1 is a simple connection from input to output:

$P_1 =$

- In case $n = 2$ two permutations exist: either the inputs are directly connected to the outputs, i.e., $b_1 = a_1$ and $b_2 = a_2$, or they are cross-connected, i.e., $b_1 = a_2$ and $b_2 = a_1$.

$P_2 =$

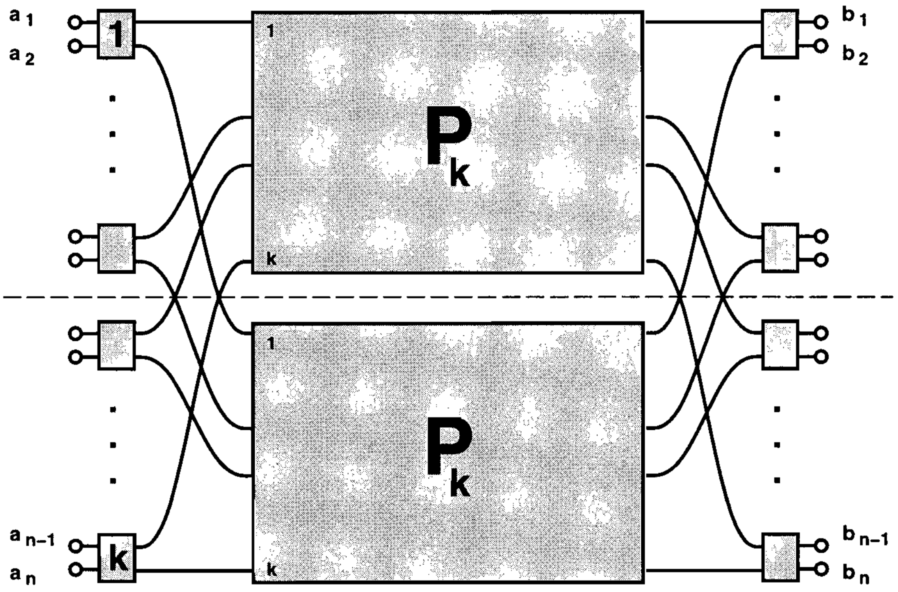


We see that P_2 is realized by two multiplexors: if $c = 0$ then input a_1 is passed on to output b_1, and a_2 is passed on to b_2, else, i.e., $c = 1$, a_1 appears at b_2 and a_2 at b_1.

- For $n \geq 3$, P_n is a network consisting of a $P_{\left\lfloor \frac{n+1}{2} \right\rfloor}$ circuit, a $P_{\left\lfloor \frac{n}{2} \right\rfloor}$ circuit, and $2 \cdot \left\lfloor \frac{n}{2} \right\rfloor$ basic P_2 circuits. The number of control variables c_n equals the number of P_2 nodes in P_n. This number can be found by solving the following recurrence equation:

$$c_n = c_{\left\lfloor \frac{n+1}{2} \right\rfloor} + c_{\left\lfloor \frac{n}{2} \right\rfloor} + 2 \cdot \left\lfloor \frac{n}{2} \right\rfloor,$$ with intial values $c_1 = 0$ and $c_2 = 1$.

It is natural to treat odd and even n separately. For even $n = 4, 6, \cdots$, P_n has the following structure, where $n = 2 \cdot k$:



For odd $n = 3, 5, \cdots$, P_n has the following slightly different structure (now $n = 2 \cdot k + 1$):

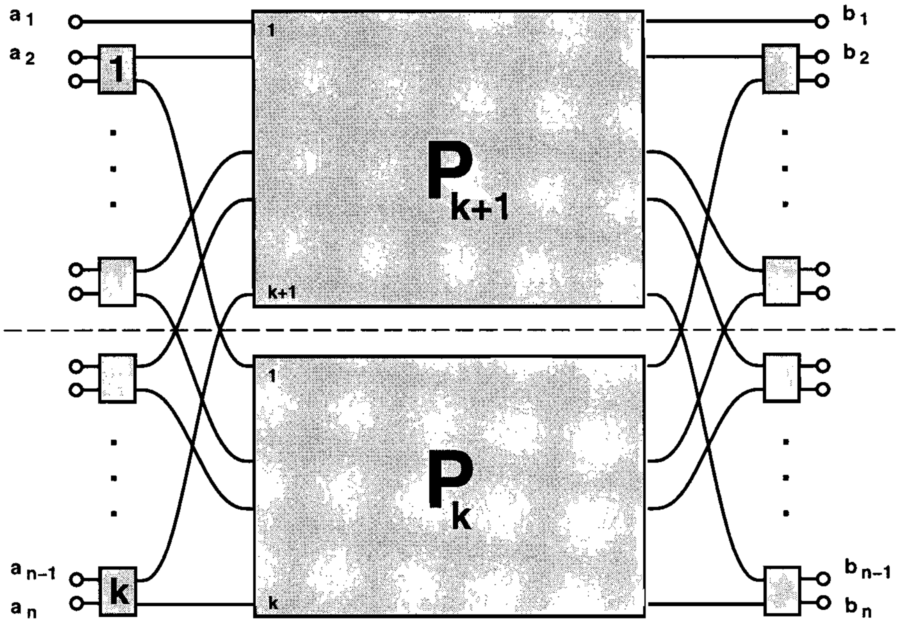


Figure 2.6 shows a 16-bit permutator as a network of 56 P_2 instances.

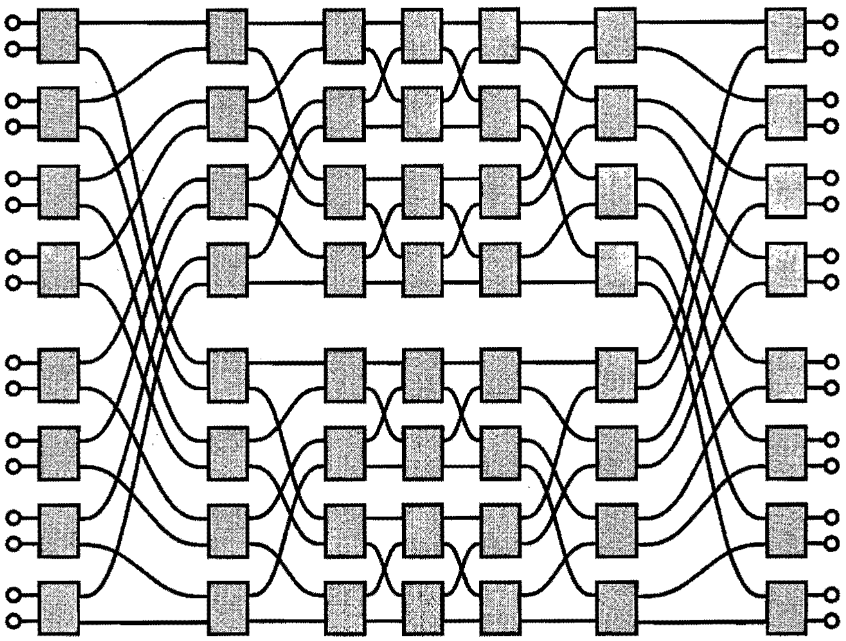


Figure 2.6. P_{16} permutator network.

It turns out that phasor and permutator circuits can be compactly represented by BDDs (to be discussed in chapter 4). The BDD size of Φ_n is $2 \cdot n$. The size for a permutator P_n is $c_n + n$ in case of an optimal variable ordering. Later it will be shown that existential quantification over all control variables can also be efficiently implemented as an operation on a BDD. However, in combination with a circuit C_2 (figure 2.4) the BDD sizes tend to grow unacceptably. Table 2.2 convincingly indicates this when we choose C_2 to be an adder. (The BDD sizes reported here do not count the constant nodes; both complemented edges and inverted-input edges are used.)

		BDD size		
n	$c_n + n$	$add_{n/2}$	$\Phi_n + P_n$	$\Phi_n + P_n + add_{n/2}$
2	3	2	5	5
4	10	5	14	79
6	18	10	24	1538
8	28	15	36	47876
10	36	20	46	>2000000

Table 2.2. Adder circuit with phasor and permutator at its inputs.

Chapter 3

Propositional Logic

3.1 Introduction

In this chapter we will look at the theory of propositional logic. It is by far the most commonly used mathematical framework in today's logic synthesis tools. It nicely fits the modelling of combinational circuits. Since our formalism constitutes a decidable theory, it is readily implemented in a computer program. We start with a brief overview of the theory and main results of propositional logic. We will hint at some practical applications and show some implementation details of the computer programs that were developed.

We will later see that many of the results need only slightly be adjusted and extended for the temporal case. Our aim is to provide enough of a mathematical basis to appreciate the engineering applications we have in mind. Therefore, theorems will often be stated in an informal way and their proofs are mostly only hinted upon. A rigorous treatment of the matter may be found in the many books on logic [Galli87]. Our goal is to explain what it means when we say that a proposition is satisfiable and show several approaches to implement such a satisfiability test.

3.2 The language of propositional logic

We introduce Propositional Logic as a language PL over an alphabet AP of atomic propositions and we will assign a meaning to each string (proposition) of the language by means of a truth assignment.

Definition 3.1 $AP = \{ p_0, p_1, p_2, \cdots \}$ is a countably infinite set of atomic propositions.

To denote an arbitrary atomic proposition we will use the letters P, Q, R, $\cdots$.

Definition 3.2 PL is the language defined by the following grammar:

```
formula ::= P
        | '¬' formula
        | '(' formula ( '∧' | '∨' ) formula ')' .

where P is an atomic proposition taken from AP.
```

The symbols $\neg$, $\wedge$, and $\vee$ are called the logical operator symbols, sometimes called the logical connectives. In the sequel, the letters A, B, C, $\cdots$ will be used to denote arbitrary formulas of PL.

Definition 3.3 A function $f : B^n \longrightarrow B$ is called an n-ary boolean function. The set $B = \{ 0, 1 \}$ is the set of truth values; 0 stands for false and 1 stands for true.

Definition 3.4 A function $v : AP \longrightarrow B$ that assigns to each atomic proposition a truth value is called a *valuation function* or *truth assignment*.

The semantics of formulas is defined by associating a function with each formula. This function is the extension $\hat{v} : PL \longrightarrow B$ of v under the usual interpretation of the logical operator symbols. The interpretation of $\neg$ is the function $H_\neg : B \longrightarrow B$ defined in table 3.1.

P	$H_\neg(P)$
0	1
1	0

Table 3.1. Semantics of $\neg$.

The interpretation of the $\wedge$ connective is the function $H_\wedge : B \times B \longrightarrow B$, and $\vee$ is interpreted by the function $H_\vee : B \times B \longrightarrow B$. Both are defined in table 3.2. (It is easy to extend these functions to apply to more than two arguments: $H_\wedge$ will be 1 only if all arguments are 1; $H_\vee$ will be 1 if at least one argument is 1.)

P	Q	$H_\wedge(P,Q)$	$H_\vee(P,Q)$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Table 3.2. Semantics of $\wedge$ and $\vee$.

Definition 3.5 Let v be a valuation function, then $\hat{v}$ is inductively defined by:

$$\begin{aligned}\hat{v}(P) &= v(P)\\ \hat{v}(\neg A) &= H_\neg(\hat{v}(A))\\ \hat{v}((A \wedge B)) &= H_\wedge(\hat{v}(A), \hat{v}(B))\\ \hat{v}((A \vee B)) &= H_\vee(\hat{v}(A), \hat{v}(B))\end{aligned}$$

Definition 3.6 Let PROP(A) stand for the set of distinct atomic propositions occurring in the formula A. PROP : $PL \longrightarrow 2^{AP}$ is inductively defined by:

$$\begin{aligned}PROP(P) &= \{P\}\\ PROP(\neg A) &= PROP(A)\\ PROP((A \wedge B)) &= PROP(A) \cup PROP(B)\\ PROP((A \vee B)) &= PROP(A) \cup PROP(B)\end{aligned}$$

Since A is a finite string, PROP (A) is finite too.

Definition 3.7 Given a formula A with $PROP(A) = \{p_1, p_2, \cdots, p_n\}$, its associated boolean function $f : B^n \longrightarrow B$ is $f = \lambda p_1, p_2, \cdots, p_n . \hat{v}(A)$.

With the semantics defined, we can now define satisfiability of a formula:

Definition 3.8 A valuation v satisfies a formula A, denoted $v \models A$, if and only if $\hat{v}(A) = 1$.

Definition 3.9 $\models A$ expresses that all valuations satisfy the formula A. Equivalently, A is called a tautological formula or tautology.

The satisfiability problem in PL asks whether a valuation exists that makes a given formula true. The tautology problem asks whether $\models A$ holds for a particular formula A. This latter problem may be stated as a satisfiability problem: to check whether $\models A$ holds it suffices to show that there is no valuation v such that $v \models \neg A$ holds. If one exists it follows that A is not a tautology. Be aware that both problems are not of the same complexity; satisfiability is NP-complete, whereas tautology is not even known to be in NP (in fact it is co-NP-complete).

Definition 3.10 Two formulas A and B are said to be logically equivalent, denoted $A \simeq B$, iff $\hat{v}(A) = \hat{v}(B)$ for all valuations v.

Theorem 3.1 For all propositions A, B, C the following properties hold:

$$\begin{aligned}
((A \vee B) \vee C) &\simeq (A \vee (B \vee C)) \\
((A \wedge B) \wedge C) &\simeq (A \wedge (B \wedge C)) \\
(A \vee B) &\simeq (B \vee A) \\
(A \wedge B) &\simeq (B \wedge A) \\
(A \vee (B \wedge C)) &\simeq ((A \vee B) \wedge (A \vee C)) \\
(A \wedge (B \vee C)) &\simeq ((A \wedge B) \vee (A \wedge C)) \\
\neg (A \vee B) &\simeq (\neg A \wedge \neg B) \\
\neg (A \wedge B) &\simeq (\neg A \vee \neg B) \\
(A \vee A) &\simeq A \\
(A \wedge A) &\simeq A \\
\neg\neg A &\simeq A \\
(A \vee (A \wedge B)) &\simeq A \\
(A \wedge (A \vee B)) &\simeq A
\end{aligned}$$

Proof: Use the properties of the underlying interpretation functions $H_\neg$, $H_\wedge$, and $H_\vee$.
□ theorem 3.1

We see that the syntax for formulas is rather restrictive. We often define some rules of precendence for the operators and use their associativity to drop a number of parentheses. The grammar in EBNF of figure 3.1 incorporates these considerations.

```
formula ::= term { '∨' term } .

term ::= factor { '∧' factor } .

factor ::= P
        | '¬' factor
        | '(' formula ')' .

where P is an atomic proposition taken from AP.
```

Figure 3.1. Relaxed grammar for PL.

Of course, this does not violate any of our previous results; the semantics may still be defined in a similar way but now it seems easier to do this with respect to the parse tree of a formula.

3.3 Truth table method

A straightforward algorithm to solve the satisfiability and tautology problems is provided by the truth table method. Given a formula A over a certain set of atomic propositions PROP (A), all we have to do is list all possible assignments v to these symbols and evaluate the formula for each of them, i.e., compute $\hat{v}(A)$. If in at least one case the evaluation leads to a true result the formula is satisfiable. If all cases evaluate to true the formula indeed expresses a tautology, and if none of the cases evaluate to true the formula is unsatisfiable or also called a contradiction. For satisfiability, the worst case complexity of this method is clearly of the order 2^n where n is the number of distinct atomic propositions appearing in the formula. Since we know that the satisfiability problem is NP-complete there is not much hope of ever finding a method that beats the exponential time complexity. However, in many problem instances the truth table method must be considered too brute force.

A more ingenious way of determining tautology is based on a search for a falsifying assignment to the formula under test. If the search fails we must conclude that the formula is indeed a tautology else we have found a counter example (refutation), namely an assignment v that yields $\hat{v}(A) = 0$. More indirectly, one could investigate the satisfiability of the negated formula; if not satisfiable the original formula is a tautology. There are several methods based on the above principles. They have in common that they are purely based on syntactical transformations of the formula under test. Typically a number of axioms (or usually axiom schemata) is defined that comprise the terminating cases of the search, which on intermediate stages is guided by a set of rewrite or inference rules. Care has to be taken that the system of axioms and inference rules is sound and complete, i.e., only tautological formulas will be classified as such and none is missed. Fortunately, for propositional logic such systems do exist, see e.g. section 3.4 "Proof Theory of Propositional Logic" in [Galli87].

3.4 Disjunctive normal form

A simple way to test for satisfiability is the disjunctive normal form (DNF) method. Here we will define a DNF as a set of sets of literals:

Definition 3.11 A DNF is a finite set $\{C_1, C_2, \cdots, C_n\}$ of clauses C_i, each C_i is a set of literals $\{L_{i1}, L_{i2}, \cdots L_{i_m}\}$. A literal is either an atomic proposition (also called positive literal) or its negation (negative literal). Negative literals will be denoted by a prime (′) after the name of the atomic proposition. A clause C_i expresses a conjunction over its literals; a DNF set expresses a disjunction over its clauses.

Definition 3.12 The transformation of a formula to a DNF set is inductively defined as follows:

$$\begin{aligned}
\mathrm{DNF}(P) &= \{\{P\}\}\\
\mathrm{DNF}(\neg P) &= \{\{P'\}\}\\
\mathrm{DNF}(\neg\neg A) &= \mathrm{DNF}(A)\\
\mathrm{DNF}(\neg(A \wedge B)) &= \mathrm{DNF}(\neg A) \cup \mathrm{DNF}(\neg B)\\
\mathrm{DNF}(\neg(A \vee B)) &= \mathrm{DNF}(\neg A) \circ \mathrm{DNF}(\neg B)\\
\mathrm{DNF}((A \wedge B)) &= \mathrm{DNF}(A) \circ \mathrm{DNF}(B)\\
\mathrm{DNF}((A \vee B)) &= \mathrm{DNF}(A) \cup \mathrm{DNF}(B)
\end{aligned}$$

with P an arbitrary atomic proposition, A and B arbitrary formulas, and $S \circ T = \{ s \cup t \mid s \in S, t \in T \text{ and } s^+ \cap t^- = s^- \cap t^+ = \emptyset \}$ for DNFs S and T and clauses s and t. We use s^+ to denote the set of positive literals of a clause s; likewise, s^- denotes its set of negative literals. Clearly, the $\circ$ operator on DNF sets selectively unites pairs of clauses, omitting pairs containing literals with opposite sign.

Definition 3.13 The meaning of a DNF set is defined by the function $h : \mathrm{DNF} \longrightarrow (B^n \longrightarrow B)$ as follows: (here we already make use of the syntax of figure 3.1 and assume $\hat{v}$ to be adjusted accordingly)

Bottom case: DNF set = $\emptyset$, $h(\emptyset) = 0$.

Inductive case: DNF set = $\{C_1, C_2, \cdots, C_n\}$, then

$h(\{C_1, C_2, \cdots, C_n\}) = \hat{v}(h(\{C_1\}) \vee h(\{C_2\}) \vee \cdots \vee h(\{C_n\}))$.

Where for each clause $C_i = \{L_1, L_2, \cdots, L_m\}$,

$h(\{\{L_1, L_2, \cdots, L_m\}\}) = \hat{v}(g(L_1) \wedge g(L_2) \wedge \cdots \wedge g(L_m))$,

in which for each positive literal $L_i = P$, $g(P) = P$,

and for each negative literal $L_i = P'$, $g(P') = \neg P$.

Now the definition of the $\circ$ operator should be clear: having a atomic proposition and its negation appear in the same clause will be interpreted as false and since all clauses are "or"-ed together we might as well avoid false clauses in the first place. Notice that the definition of h has been chosen in a way such that:

Theorem 3.2 $\hat{v}(A) = h(\mathrm{DNF}(A))$ holds for every formula A.

Proof: Use the definitions of v and h and use the following identities of theorem 3.1:

$$
\begin{aligned}
(A \vee (B \wedge C)) &= ((A \vee B) \wedge (A \vee C)) \\
(A \vee B) &= (B \vee A) \\
\neg (A \vee B) &= (\neg A \wedge \neg B) \\
\neg (A \wedge B) &= (\neg A \vee \neg B) \\
\neg\neg A &= A
\end{aligned}
$$

□ theorem 3.2

Corollary 3.1 The satisfiability problem — is there a v such that $v \models A$? — can now be rephrased as "is the DNF set nonempty, i.e., $\mathrm{DNF}(A) \neq \varnothing$?" Similarly, $\models A$ holds if and only if $\mathrm{DNF}(\neg A) = \varnothing$.

Example 3.1 Does $\models ((P \wedge Q) \vee (\neg P \vee R))$ hold?

$$
\begin{aligned}
&\mathrm{DNF}(\neg ((P \wedge Q) \vee (\neg P \vee R))) \\
&= \mathrm{DNF}(\neg (P \wedge Q)) \circ \mathrm{DNF}(\neg (\neg P \vee R)) \\
&= (\mathrm{DNF}(\neg P) \cup \mathrm{DNF}(\neg Q)) \circ (\mathrm{DNF}(\neg\neg P) \circ \mathrm{DNF}(\neg R)) \\
&= (\{\{P'\}\} \cup \{\{Q'\}\}) \circ (\{\{P\}\} \circ \{\{R'\}\}) \\
&= \{\{P'\}, \{Q'\}\} \circ \{\{P, R'\}\} \\
&= \{\{P, Q', R'\}\}
\end{aligned}
$$

So we get a nonempty result, meaning the original formula is not a tautology.
□ example 3.1

3.5 Containment (or subsumption)

As with formulas of PL, we can have many different DNF sets with the same meaning. For instance the sets $\{\{P\}\}$ and $\{\{P\}, \{P, Q', R\}\}$ have the same meaning $\lambda p, q, r.\, p$. The clauses of a DNF set are partially ordered according the subset relation $\subseteq$. The following lemma is useful in minimizing the number of clauses:

Lemma 3.1 Let $\mathrm{DNF}(A) = \{C_1, C_2\}$. If $C_1 \subseteq C_2$ then $h(\{C_1, C_2\}) = h(\{C_1\})$. We will say, although this may sound counter-intuitive, that C_1 contains C_2. (The term 'contains' stems from the fact that when a clause is interpreted as a set of points in B^n, we have that whenever $C_1 \subseteq C_2$ then the points of C_2 are contained in the set of points of C_1.)

Proof: Clauses express a conjunction ($\wedge$) of their elements. Each element (literal) in a clause must evaluate to 1 in order for the clause to be 1. For C_2 to be 1 at least all elements in it that are also in C_1 must thus be 1, so C_1 will be 1 but then $\mathrm{DNF}(A)$ is already 1 independent of the remaining literals in C_2. Conversely, if C_1 evaluates to 0 then also C_2.
□ lemma 3.1

Given a DNF set we may replace it with the set consisting of the minimal elements of that DNF set under the $\subseteq$ relation. To find these minimal elements requires a quadratic number of $\subseteq$ comparisons, viz. every element has to be compared with the rest.

3.6 Implementation of DNF

In a program that tests satisfiability based on the DNF set approach we need to represent sets of sets of literals. Regarding the operations that are to be performed on this abstract data structure we derive that clauses are conveniently implemented by means of bitvectors and that a set of them can be represented by a singly-linked list. This way uniting DNF's requires appending (or concatenating) lists, and the $\circ$ operation can be based on bitwise 'and' and 'or' operations on the bitvectors. Each clause will be represented by 2 bitvectors, one to express the inclusion of a positive literal and one to do the same for negative literals. A DNF set is then represented by a list of pairs of bitvectors.

```
list-of-pairs-of-bitvectors DNF(PL f)
{
  switch (f) {
  case P:
    clause C:=pair (pos, neg) of all-0's bitvectors;
    set bit in pos(C) corresponding with P;
    return singleton list with C as element;

  case ¬P:
    clause C:=pair (pos, neg) of all-0's bitvectors;
    set bit in neg(C) corresponding with P;
    return singleton list with C as element;

  case ¬¬A:
    return DNF(A);
  case ¬(A∧B):
    return concat(DNF(¬A), DNF(¬B));
  case ¬(A∨B):
    return ∘(DNF(¬A), DNF(¬B));
  case (A∧B):
    return ∘(DNF(A), DNF(B));
  case (A∨B):
    return concat(DNF(A), DNF(B));
  }
}
```

Algorithm 3.1. Implementation of DNF.

Algorithm 3.1 shows a straightforward implementation of the conversion of a PL formula to its DNF form as defined in definition 3.12. Details of the $\circ$ operator are presented in algorithm 3.2.

```
list-of-pairs-of-bitvectors ∘(list-of-pairs-of-bitvectors S, T)
{
 L:=empty list;
 for (s∈S)
   for (t∈T)
     if (   (pos(s) & neg(t))=all-0's
         && (neg(s) & pos(t))=all-0's) {
       clause C:=pair (pos, neg) of all-0's bitvectors;
       pos(C):=pos(s) | pos(t);
       neg(C):=neg(s) | neg(t);
       L:=concat(L, singleton list with C as element);
     }
 return L;
}
```

Algorithm 3.2. Implementation of the ∘ operator.

The worst-case performance of the above algorithm is exponential in the size of the formula, and, considering that satisfiability is an NP-complete problem, there is little hope to ever find a polynomial-time algorithm.

The resulting DNF list can often be simplified using containment. A good heuristic that we use in our implementation is to define a lexicographical ordering on the clauses (based on a linear ordering of the literals) and then while doing a mergesort on the clauses mark the ones that are subsumed (or contained) by others. The marked clauses may be deleted from the DNF list. This way only a total of $n^2\log(n)$ containment tests is performed.

The DNF representation will be used in chapter 6 when we look at an algorithm for the satisfiability of propositional linear-time temporal logic.

Chapter 4

Binary Decision Diagrams

4.1 Introduction

So far we have looked at two distinct representations of boolean functions: PL formulas and DNF sets. To solve the tautology problem we are basically asking whether the formula or DNF set denotes the **1** function, i.e., the function that is invariably true. The major drawback of the representations is that many formulas and many DNF sets denote the same function. So it is not always directly obvious that two different descriptions indeed describe the same circuit (i.e., have the same logic behaviour). Here we will introduce a representation for a boolean function that does not have this disadvantage: the Binary Decision Diagram (BDD) also known as Boolean Function Graph. Seminal work by [Bryan86, Karpl89] has shown that given a fixed ordering of the function's place-holder variables, binary decision diagrams are canonical: each distinct function may be represented by a unique binary decision diagram. Moreover, we will shortly see that in an implementation each function can be represented by a unique reference (pointer value), thus tautology checking reduces to testing whether two references are the same. However, constructing a binary decision diagram might in the worst case require an exponential number of operations. The success of binary decision diagrams lies in the fact that for many practical functions it is known that the diagram has a size which is only polynomially related to the number of variables.

4.2 Notation and terminology

We consider boolean functions all with the same arity, i.e., they all have the same, fixed number of arguments $n \geq 0$. These functions can be denoted by propositional formulas $\phi(X)$ over a set of atomic propositions $X = \{x_1, x_2, \cdots, x_n\}$ as $f = \lambda x_1, x_2, \cdots, x_n . \phi(X)$. Here the x_i's will be called boolean variables. The functions will be total or complete in the sense that they are defined for every element of the domain B^n. Two functions are considered identical when application of each to all 2^n distinct argument values always gives the same result value.

Our presentation of BDD's follows the terminology used in [Brace90].

Definition 4.1 A BDD is a labelled, possibly multi-rooted, directed acyclic graph (DAG) $G(V, E)$. We distinguish two kinds of vertices: I-nodes and T-nodes. The edge set E is defined by a relation $E \subseteq \text{I-nodes} \times (\text{I-nodes} \cup \text{T-nodes})$. The details are as follows:

- I-nodes is a finite set of internal vertices, each having precisely 2 outgoing edges to (other) internal or terminal vertices. One is said to be the else-edge, the other is the then-edge. The successor vertex of an internal vertex v via its else-edge is denoted by $\text{else}(v)$; the successor accessible via its then-edge is denoted by $\text{then}(v)$. Each internal vertex v is labelled with a variable denoted by $\text{var}(v)$. The set of variables is assumed to be totally ordered, i.e., we postulate the existence of a ranking function $\text{rank} : X \longrightarrow \{1, 2, \cdots, n\}$; $\text{rank}(x)$ then denotes the rank number of variable x in the ordering. When no confusion can occur, we write $\text{rank}(v)$ instead of $\text{rank}(\text{var}(v))$.
- $\text{T-nodes} = \{\text{zero}, \text{one}\}$ is the set of terminal vertices. Terminal vertices have no outgoing edges. For uniformity we define the rank of a terminal vertex to be $n + 1$.

Definition 4.2 An Ordered BDD (OBDD) is a BDD where for each internal vertex $v \in$ I-nodes we have:

$$\text{rank}(v) < \min(\text{rank}(\text{else}(v)), \text{rank}(\text{then}(v))).$$

Definition 4.3 A Reduced Ordered BDD (ROBDD) is an OBDD such that for each internal vertex its two successors are distinct and the graph does not contain isomorphic subgraphs.

For any BDD (not necessarily reduced and ordered) we can define a meaning in the following way:

Definition 4.4 Assume a fixed set of variables $\{x_1, x_2, \cdots, x_n\}$. With each vertex $v \in V$ in the BDD we associate a boolean function $I(v)$. The (higher-order) interpretation function $I : V \longrightarrow (B^n \longrightarrow B)$ is inductively defined on the structure of the graph. We abbreviate $x_1, x_2, \cdots, x_n$ to X, and assume that the indexing is such that $\text{rank}(x_i) = i$.

- For terminal vertices zero and one:

  $\mathsf{I}(\mathrm{zero}) = \mathbf{0}$, i.e., the function $\lambda X.\,0$
  $\mathsf{I}(\mathrm{one}) = \mathbf{1}$, i.e., the function $\lambda X.\,1$

- For every internal vertex v:

  $$\mathsf{I}(v) = \mathrm{ITE}(\lambda X.\,\mathrm{var}(v), \mathsf{I}(\mathrm{then}(v)), \mathsf{I}(\mathrm{else}(v)))$$

  with $\mathrm{ITE}(F,G,H) = \lambda X.\,H_{\mathrm{ITE}}(F(X),G(X),H(X))$. ITE is a higher-order function manipulating functions with signature $B^n \longrightarrow B$, based on the 3-argument boolean function $H_{\mathrm{ITE}}(P,Q,R) = H_{\vee}(H_{\wedge}(P,Q), H_{\wedge}(H_{\neg}(P),R))$, see table 4.1. The name ITE is an acronym for If-Then-Else.

P	Q	R	$H_{\mathrm{ITE}}(P,Q,R)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Table 4.1. Semantics of the ITE operator.

- The functions associated with the BDD are the functions associated with its root vertices.

It is customary to let the projection function associated with a variable be denoted by the name of that variable, thus if x_i is some variable then x_i stands for the function $\lambda X.\,x_i$. Also, often the same logical operator symbol is used to denote the corresponding higher-order operation, e.g. one often sees $\mathrm{ITE}(F,G,H) = (F \wedge G) \vee (\neg F \wedge H)$. Here, however, we will use the symbol $\cdot$ to denote the higher-order "And" operation and the symbol + to denote the higher-order "Or" operation; higher-order "Not" is indicated by placing a bar $\bar{\ }$ above the function name.

From now on, we will simply write BDD when in fact we mean a Reduced Ordered BDD. We also assume some ranking function $\mathrm{rank} : X \longrightarrow \{1,2,\cdots,n\}$ to be defined for the variables (e.g. we may take $\mathrm{rank}(x_i) = i$, however, this is only one of the $n!$ possibilities).

Note that for each internal vertex v in a BDD with $\mathrm{var}(v) = x$, $T = \mathsf{I}(\mathrm{then}(v))$, and $E = \mathsf{I}(\mathrm{else}(v))$, $\mathrm{ITE}(x,T,E)$ is by definition a unique function (within that BDD). We may therefore use the triple $\langle x,T,E\rangle$ as a unique identification for the internal vertex v, but also as an identification for the sub-BDD rooted at that vertex. For convenience we will denote the BDD by the function it represents, which is already done here for the BDDs T and E; in general, i.e., when they are not bottom cases, T and E themselves can be represented by such triples (they are the labels of the vertices then (v) and else (v) respectively). Bottom cases are the terminal vertices; they will be labeled by their respective functions **0** and **1**. We refer to them as constant BDDs.

Example 4.1 The BDD for the function $\lambda a, b, c.\, a \wedge \neg b \vee \neg a \wedge b \wedge c$ and represented by $\langle a, \langle b, \mathbf{0}, \mathbf{1} \rangle, \langle b, \langle c, \mathbf{1}, \mathbf{0} \rangle, \mathbf{0} \rangle \rangle$ is depicted in figure 4.1.

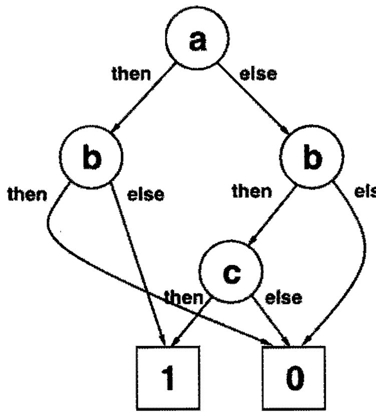


Figure 4.1. Example BDD graph for $f(a, b.c) = a \wedge \neg b \vee \neg a \wedge b \wedge c$.

In drawing a BDD the following conventions will be adhered to:

1. A vertex is represented by a circle containing the name of its variable label.
2. By letting incoming edges enter at the top of the circle and outgoing edges leave at the bottom of the circle there is no need to draw an arrow to indicate their direction.
3. Instead of labelling the edges with then and else we will use 1 and 0 respectively, or preferably even do without these labels and then it is understood that the edge exiting on the bottom left of a circle is the then-edge, and the edge exiting on the bottom right is the else-edge.
4. Terminal vertices are not explicitly drawn. We simply let the edge end in a 0 or a 1 symbol.

Figure 4.2 shows the example BDD drawn according the above rules.

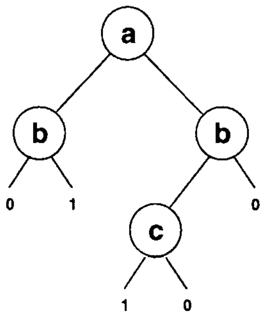


Figure 4.2. Example BDD graph drawn in 'minimalistic' style.

□ example 4.1

4.3 BDD construction

Now we investigate how BDDs can be put to practical use, i.e., manipulate them by means of logical operators. The following table 4.2 lists useful operations on functions F and G and shows how the same result may be achieved in terms of the higher-order ITE function. The objective of this section is to derive an algorithm for ITE operating on the BDD representation of the functions. We shall first formulate ITE in a recursive way.

Name:	Notation:	ITE form:
not	$\overline{F}$	ITE(F, **0**, **1**)
and	$F \cdot G$	ITE(F, G, **0**)
xor	$F \oplus G$	ITE(F, $\overline{G}$, G)
or	$F + G$	ITE(F, **1**, G)
nor	$\overline{F + G}$	ITE(F, **0**, $\overline{G}$)
equiv	$F \leftrightarrow G$	ITE(F, G, $\overline{G}$)
implies	$F \rightarrow G$	ITE(F, G, **1**)
nand	$\overline{F \cdot G}$	ITE(F, $\overline{G}$, **1**)

Table 4.2. Operations on functions and equivalent ITE formulation.

Definition 4.5 The Shannon expansion of a boolean function F with respect to a variable x is the decomposition of F in its cofactors F_x and $F_{\bar{x}}$ according to $F = x \cdot F_x + \bar{x} \cdot F_{\bar{x}}$. The cofactor F_x is the restriction of F under $x = 1$, similarly $F_{\bar{x}}$ is the restriction of F under $x = 0$. (Note that for any variable, a cofactor of a constant function is the function itself). The support of a function is the set $\{ x_i \mid F_{x_i} \neq F_{\bar{x}_i} \}$. Constant functions have empty support.

Shannon expansion gives us a recursive procedure for BDD construction. Observe that the following derivation is valid:

$$\begin{aligned}
\mathsf{ITE}(F,G,H) &= x\cdot \mathsf{ITE}(F,G,H)_x + \bar{x}\cdot \mathsf{ITE}(F,G,H)_{\bar{x}}\\
&= x\cdot (F\cdot G+\bar{F}\cdot H)_x + \bar{x}\cdot (F\cdot G+\bar{F}\cdot H)_{\bar{x}}\\
&= x\cdot (F_x\cdot G_x+\bar{F}_x\cdot H_x) + \bar{x}\cdot (F_{\bar{x}}\cdot G_{\bar{x}}+\bar{F}_{\bar{x}}\cdot H_{\bar{x}})\\
&= x\cdot (\mathsf{ITE}(F_x,G_x,H_x)) + \bar{x}\cdot (\mathsf{ITE}(F_{\bar{x}},G_{\bar{x}},H_{\bar{x}}))\\
&= \mathsf{ITE}(x, \mathsf{ITE}(F_x,G_x,H_x), \mathsf{ITE}(F_{\bar{x}},G_{\bar{x}},H_{\bar{x}}))
\end{aligned}$$

and the latter stands for the BDD:

$$\langle x, \mathsf{ITE}(F_x,G_x,H_x), \mathsf{ITE}(F_{\bar{x}},G_{\bar{x}},H_{\bar{x}})\rangle$$

All that remains to be shown is how to compute the cofactors of a function and provide the bottom cases of the recursive expansion sketched above. Without violating the validity of the expansion, we may choose x to be the variable with the smallest rank among the variables labelling the root vertices of the BDDs for the functions F, G and H. Remember that the rank of a constant BDD is $n+1$ by definition. Assume the BDD for F non-constant, so let $F=\langle y,T,E\rangle$. Surely $\mathsf{rank}(x)\le\mathsf{rank}(y)$. If $\mathsf{rank}(x)<\mathsf{rank}(y)$ then x is not in the support of F, hence $F_x = F_{\bar{x}} = F$; otherwise we must have $x = y$, and so $F_x = T$ and $F_{\bar{x}} = E$ (figure 4.3). For the functions G and H we can follow the same reasoning. Clearly, the bottom cases are determined by F being constant: $\mathsf{ITE}(\mathbf{0},G,H)=H$ and $\mathsf{ITE}(\mathbf{1},G,H)=G$.

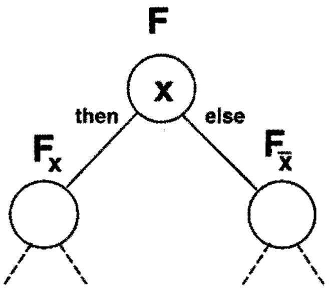


Figure 4.3. Function F and its cofactors w.r.t. the top variable x.

An implementation of the high-order ITE function operating on boolean functions represented by BDDs is outlined in algorithm 4.1. In an actual implementation, for instance in Pascal or C, it is convenient to define the type BDD to be a pointer to a vertex record (or struct). Then testing for equality of two BDDs can be done using the equality operation (=) on pointer values provided that BDD DAGs are uniquely stored, i.e., no isomorpic subgraphs exist. This uniqueness of subgraphs is achieved by keeping track of all the $\langle x,T,E\rangle$ triples in a hash table. The BDDs for the projection functions $\lambda X.\, x_i$ for the variables that are denoted by the triples $\langle x_i,\mathbf{1},\mathbf{0}\rangle$ are also kept in this hash table. The information stored for each triple is the BDD pointer that points to the root vertex of the particular DAG for that BDD. In algorithm 4.1 the statement `if (T = E) then return T;`

ensures the first condition mentioned in definition 4.3 (successors are distinct) to be met and returning the unique BDD for the triple $\langle x, T, E\rangle$ ensures the second condition to be met, namely that no isomorphic subgraphs are constructed.

```
BDD ite(BDD F,G,H)
{
  /* Bottom cases: */
  if (F = 0) return H;
  if (F = 1) return G;
  /* Recursive case: */
  x:=min(var(F),var(G),var(H));
  T:=ite($F_x$,$G_x$,$H_x$);
  E:=ite($F_{\bar{x}}$,$G_{\bar{x}}$,$H_{\bar{x}}$);
  if (T = E) return T;
  return unique BDD for ⟨x,T,E⟩;
}
```

Algorithm 4.1. The basic BDD constructor function ite.

Many optimizations of the ite function may be considered, e.g. including more tests for special cases and supplying ite with a memory function that avoids recomputation. For the latter we keep a table of argument and result values that we consult upon entry (lookup) and update upon exit (remember). Some of these ideas are incorporated in algorithm 4.2.

```
BDD ite(BDD F,G,H)
{
  if (F = 0) return H;
  if (F = 1) return G;
  /* Special case: */
  if (G = 1 && H = 0) return F;
  /* Make use of memory function: */
  R:=lookup(F,G,H);
  if (R ≠⊥) return R; /* value ⊥ signals absence */
  x:=min(var(F),var(G),var(H));
  T:=ite($F_x$,$G_x$,$H_x$);
  E:=ite($F_{\bar{x}}$,$G_{\bar{x}}$,$H_{\bar{x}}$);
  if (T = E) R:=T; else R:=unique BDD for ⟨x,T,E⟩;
  /* Supply memory function with new data: */
  remember "ite(F,G,H)=R";
  return R;
}
```

Algorithm 4.2. A more practical ite function.

More implementation issues are discussed in chapter 8.

Part II

Verification of Circuits with Memory

This part consists of three chapters. Its structure mimics the first part: first we introduce sequential circuits in chapter 5; then chapter 6 addresses two temporal logics, CTL and PTL, that are useful in reasoning about the behaviour of this type of circuit; lastly, chapter 7 presents a more general formalism that encompasses both temporal logics, namely μ-calculus. For all three formalisms, basic implementations are presented.

Chapter 5

Sequential Circuits

5.1 Introduction

A sequential circuit can be defined as a logical circuit for which the values of the outputs not only depend on the present values on the inputs, but also on the history of the system. For that, the circuit needs some way of remembering what happened in the past: it needs memory. Time therefore becomes an explicit parameter in the logical functions that describe the outputs of a sequential circuit. We usually abstract from a real time value and use the notion of a clock instead. This means that we are only interested in observing the system at certain discrete points in time, e.g. the moment just after the clock signal has risen from 0 to 1. The mathematical model of such systems is the finite automaton. The system's memory is then replaced by the concept of state. The finiteness of the automaton implies that its behaviour is of an intrinsic repetitive nature. This directly translates to the decidability of the equivalence problem for automata.

5.2 Finite automata and sequential circuits

A finite automaton is usually defined by the quintuple $(Q, \Sigma, \delta, q_0, F)$, where

Q is a finite, non-empty, set of states;
Σ is a finite, non-empty, set of input symbols, the alphabet;
$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the transition relation, with ε the empty word;
$q_0 \in Q$ is the initial state of the machine; and
$F \subseteq Q$ is a non-empty set of final states.

Although the above definition of δ is the most general, we often like to restrict it to a mapping (or even a total function: the next-state function): $\delta : Q \times \Sigma \longrightarrow Q$. That is, instead of considering non-deterministic machines we focus on deterministic ones. Despite the fact that the transformation from non-deterministic to deterministic finite automaton (e.g. through subset construction) may lead to a possible exponential increase in the number of states, non-deterministic and deterministic finite automata have the same expressive power: the sets of words recognized by finite automata are precisely the regular languages. A word (or string) over Σ is the concatenation of a finite number of symbols from Σ, commonly denoted by juxtaposition. The set of all words over Σ is denoted by Σ^*. Algorithm 5.1 gives an operational definition of recognition or acceptance of a word by a deterministic finite automaton.

```
Bool accept (Σ* w)
{
  for (i:=0, q:=q0; i<|w|; i++)
    q:=δ(q, w[i]);
  return q∈F;
}
```

Algorithm 5.1. Deterministic finite automaton acceptance.

For our purposes we like to alter this machine to have outputs instead of merely a set of final states. We define Γ as our output alphabet and introduce the output function $\gamma : Q \times \Sigma \longrightarrow \Gamma$, which leads to a so-called Mealy machine model. If the output does not depend on the current input value, i.e., $\gamma : Q \longrightarrow \Gamma$, we have a Moore type of machine.

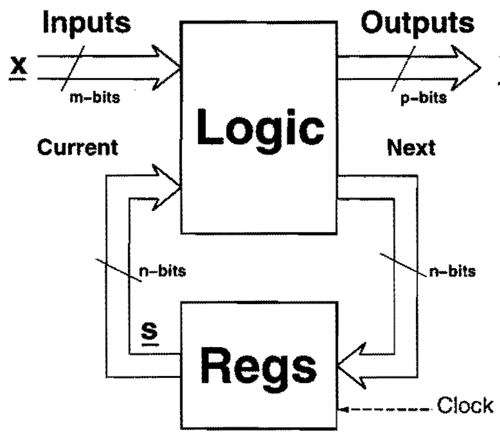


Figure 5.1. Basic sequential machine model: logic with feedback through registers.

Moreover, when all sets are appropriately coded over boolean spaces, we obtain the basic hardware model of a sequential machine. The machine's current state (in coded form) is kept in a bank of 1-bit registers (flip-flops). The next-state and output-function are realized by combinational logic, see figure 5.1. The update of the registers is assumed to be instantaneous and synchronized by the clock. It is customary to leave out the clock signal in a drawing.

Again let $B = \{0, 1\}$ be the set of boolean truth values. A coding of a finite set A is a one-to-one function $\rho : A \longrightarrow B^n$ with appropriately chosen n, e.g. it is always possible to construct a ρ such that $n = \lceil {}^2\log(\,|A|\,)\rceil$. Assume the sets Σ, Q, and Γ are coded in the sets B^m, B^n, and B^p respectively. Then a model for a Mealy machine is the sixtuple (S, I, O, N, S_0, Y), with

$S = B^n$ the set of states (contents of the register);

$I = B^m$ the set of input bit-patterns;

$O = B^p$ the set of output bit-patterns;

$N : B^n \times B^m \longrightarrow B^n$ the (possibly partial) next-state function;

$S_0 \subseteq S$ the non-empty set of initial states, and

$Y : B^n \times B^m \longrightarrow B^p$ the (possibly partial) output function.

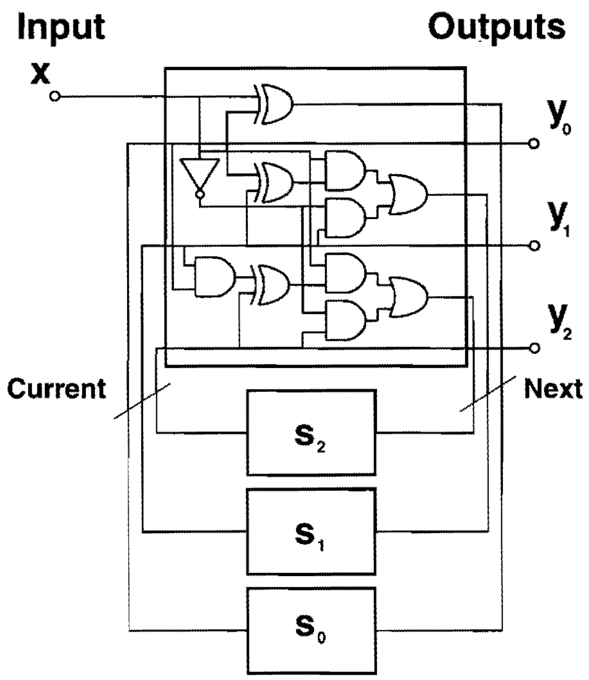


Figure 5.2. Modulo-8 incrementer: adds value of x to s̲ (mod 8).

To denote an element of a cartesian power of the set B we allow both a tuple notation and vector notation and use them interchangeably. Also, constant values may be denoted by a string of 0's and 1's. In the sequel we assume N and Y to be total functions. In that case the machine is fully specified when N, Y, and S_0 are known, and we might as well specify the machine (circuit) by the triple (N, S_0, Y). In general, we will allow multiple initial states, and denote them by S_0; even in case of a single initial state q_0 we will use S_0 and then $S_0 = \{q_0\}$. Often, we assume $q_0 = 0^n$.

Figure 5.2 depicts a modulo-8 incrementer moulded in our model. Clearly, $N = [N_2, N_1, N_0]$ may be defined by the following boolean functions:

$$N_0 = \lambda s_2, s_1, s_0, x.\, s_0 \oplus x,$$
$$N_1 = \lambda s_2, s_1, s_0, x.\, x \wedge (s_0 \oplus s_1) \vee \neg x \wedge s_1,$$
$$N_2 = \lambda s_2, s_1, s_0, x.\, x \wedge (s_2 \oplus (s_1 \wedge s_0)) \vee \neg x \wedge s_2.$$

The output function $Y = [Y_2, Y_1, Y_0]$ is simply the identity on the state variables:

$$Y_0 = \lambda s_2, s_1, s_0.\, s_0,$$
$$Y_1 = \lambda s_2, s_1, s_0.\, s_1,$$
$$Y_2 = \lambda s_2, s_1, s_0.\, s_2.$$

Note that we prefer to number the indices starting at 0 from right to left; this is most natural when one wants to interpret a vector of boolean values as a decimal number. For instance, we will write state $\underline{s} = (1, 1, 0) = 6$ decimal.

5.3 Verification problem

An important verification problem for sequential circuits can be formulated as follows:

SEQUENTIAL CIRCUIT EQUIVALENCE
INSTANCE: Two sequential circuits M_1 and M_2 according to the Mealy model over the same set of inputs and outputs.
QUESTION: Are both circuits functionally equivalent? That is, do both circuits for all possible sequences of input bit-patterns yield the same sequences of bit-patterns at their corresponding outputs?

One way to visualize this problem is depicted in figure 5.3. The two circuits are assumed to operate synchronously through the control of the implicit clock signal. Both circuits are fed the same input values. The single bit output signal is derived from logically AND-ing the results of the bitwise equivalence of the respective outputs. The composition of both circuits (Mealy machines) in this manner is said to form the product machine. The question is now whether the product machine yields the output value 1 for all possible input sequences. In terms of automata, a 'correct' product machine corresponds to recognizing the universal language over B^m i.e., $(B^m)^*$. (By definition, a Mealy machine outputs ε on input ε, but this technicality is only of minor theoretical interest; in practice we do not consider ε a feasible input).

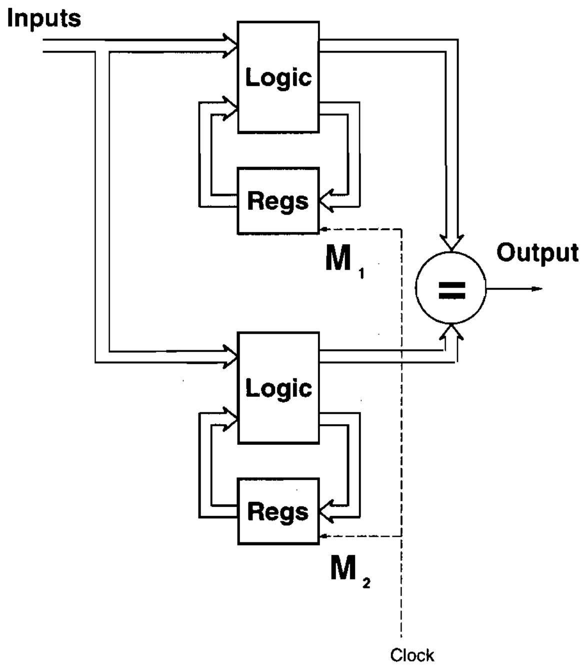


Figure 5.3. The product of two Mealy machines.

The above problem statement does not say anything about the respective starting states of the two machines M_1 and M_2. One practical way of interpreting the problem is to assume the equivalence to exist when the machines are *started* in any of the possible combinations of their initial states. The next sections present a method to solve the equivalence problem.

5.4 State-space exploration

This section is not a chapter from the Captain's Log of the Starship Enterprise. (Come to think of it, "Enterprise" would be a good name for a sequential circuit verification program.) Consider a finite automaton $M = (Q, \Sigma, \delta, S_0)$. We allow multiple initial states and for the moment are not interested in the final states. For generality we assume δ to be specified as a relation. Our goal will be the exploration of the states reachable from the initial states S_0 at each δ-step considering all possible input symbols from the set Σ. Then only the structure of the state-space is of importance, and for that we define the immediate neighbourhood functions:

$\eta : Q \times Q \longrightarrow B$, with $\eta(s,t) = \text{true}$ if $\exists_{x \in \Sigma}\ (s,x,t) \in \delta$ and false otherwise, and

$H : 2^Q \longrightarrow 2^Q$, with $H(S) = \{t \in Q \mid \exists_{s \in S}\ \eta(s,t)\}$.

The latter function over sets of states is also known as a predicate transformer [Burch94] or a functional [McMil93]. These terms will become clear shortly. Algorithm 5.2 gives the traditional breadth-first computation of the set of reachable states.

```
$2^Q$ explore ($2^Q$ $S_0$)
{
  Reach:=$\emptyset$;   New:=$S_0$;
  do {
    Reach:=Reach$\cup$New;
    Next:=H(New);
    New:=Next$\setminus$Reach;
  } while (New$\neq\emptyset$);
  return Reach;
}
```

Algorithm 5.2. Breadth-first state-space exploration.

Note that algorithm 5.2 is rather strict in the sense that it does not explicitly indicate the possible freedom of choice for the argument to H: there is no harm in including some states that have already been reached at that point. This may seem not very useful in the current context, but depending on the representation of the sets in an implementation some computational advantage may be gained. For later reference we also add the iteration counter k. Algorithm 5.3 incorporates these observations.

```
$2^Q$ explore ($2^Q$ $S_0$)
{
  k:=0;   Reach:=$\emptyset$;   New:=$S_0$;
  do {
    k++;
    Reach:=Reach$\cup$New;
    Choose Front such that New$\subseteq$Front$\subseteq$Reach;
    Next:=H(Front);
    New:=Next$\setminus$Reach;
    /* Generic situation at this point is depicted in figure 5.4. */
  } while (New$\neq\emptyset$);
  /* k = number of iterations. */
  return Reach;
}
```

Algorithm 5.3. Modified breadth-first state-space exploration.

Figure 5.4 shows a Venn diagram relating the various sets of states at the indicated point during the execution of algorithm 5.3. The shaded area denotes the set of newly discovered states after the kth application of H. The dashed curve indicates the reachable states afterwards.

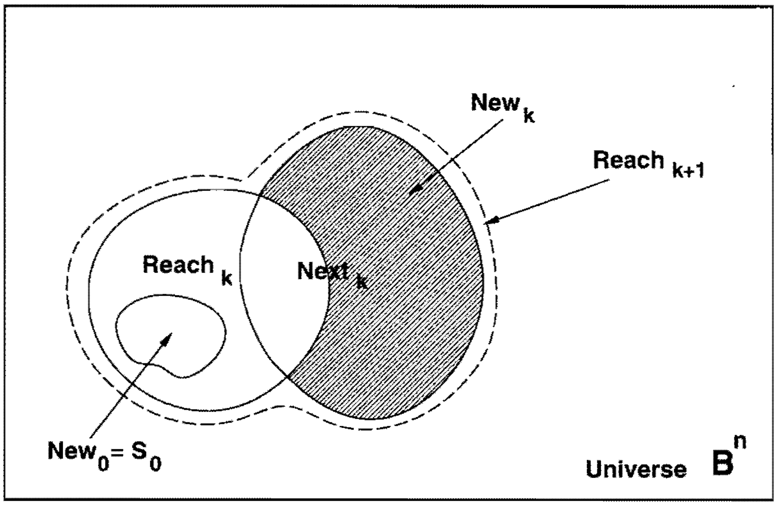


Figure 5.4. Partitioning of the state-space during exploration.

As an example, table 5.1 shows the results of applying `explore` to the modulo-8 incrementer of figure 5.2. For brevity, the states have been coded in decimal. We see the typical behaviour of counter-like structures: in each step only a single new state is discovered. Modulo-n counters are interesting test cases because their reachable state-space is the whole universe B^n and the space may be said to have minimal structure: there is precisely 1 successor state for each state.

k	Reach$_k$	New$_k$
0	∅	{0}
1	{0}	{1}
2	{0,1}	{2}
3	{0,1,2}	{3}
4	{0,1,2,3}	{4}
5	{0,1,2,3,4}	{5}
6	{0,1,2,3,4,5}	{6}
7	{0,1,2,3,4,5,6}	{7}
8	{0,1,2,3,4,5,6,7}	∅

Table 5.1. State-space exploration of the modulo-8 incrementer.

The explicit determination of the `New` set in algorithm 5.3 can be avoided and its test for emptiness replaced by a set-equality test by a simple rearrangement of the statements. Note that in case we happen to choose `Front`$\supset$`New_Reach`$\setminus$`Reach`, the union is no longer disjoint. This results in algorithm 5.4.

```
$2^Q$ explore ($2^Q$ $S_0$)
{
  k:=1;   Front:=Reach:=$S_0$;
  do {
    New_Reach:=Reach$\cup$H(Front);
    if (New_Reach = Reach) return Reach;
    k++;
    Choose Front such that New_Reach\Reach$\subseteq$Front$\subseteq$New_Reach;
    Reach:=New_Reach;
  } forever;
}
```

Algorithm 5.4. More refined reachability analysis.

5.5 Symbolic computation

The algorithms for state-space exploration presented thus far are expressed in terms of operations on sets of states. In general, finite sets can conveniently be implemented using a bitvector data type. For large state-spaces however, this approach quickly becomes inefficient, if not altogether infeasible, both with respect to memory and runtime. Assuming that all bitvector operations needed for algorithm 5.2 take $O(|Q|)$ time and in the worst case one state is discovered 'reachable' per iteration, then the overall complexity is $O(|Q|^2)$. This doesn't sound too bad, except when one realizes that the number of reachable states is in the worst case exponential in the number of register bits. For the (worst case) modulo-8 counter example we experimentically find a runtime of $O(4^n)$. The key idea to improve on the above is to refrain from representing the sets of states explicitly; instead, sets will be represented by their characteristic function. And, more precisely, these characteristic functions will themselves be represented by BDDs. This approach is coined symbolic or implicit reachability analysis.

We first introduce the necessary theory and notational conventions. In the following B denotes the set of boolean truth values. The characteristic function of a set $A \subseteq U$ is defined as

$$\chi_A : U \longrightarrow B, \text{ with } \chi_A = \lambda a.\, a \in A.$$

The isomorphism between boolean algebra and the algebra of subsets of a finite set suggests to adopt the convention to overload the name of the set to also denote its accompanying characteristic function which we shall occasionally use as a predicate. Similarly, for a binary relation R on a set A, we define its characteristic function:

$$\chi_R : A \times A \longrightarrow B, \text{ with } \chi_R = \lambda a, b.\, (a, b) \in R.$$

In the context of a fixed universe U we will abbreviate quantifications $\exists_{u \in U}$ to $\exists_u$, and the same holds for universal quantifications. We will use the following notations for a pair belonging to a binary relation:

$(a, b) \in R$, or $a\,R\,b$, or $R(a, b)$.

The last form is an example of the use of the name of the relation as a predicate. Also, if we allow 'currying', i.e., supply less arguments to a function than its arity prescribes, $R(a)$ should be understood to mean (the characteristic function of) the image of a under R.

We define the composition of two relations R_1 and R_2 on the same universe U by $R_1 \circ R_2 = \lambda s, t. \exists_u [R_1(s, u) \wedge R_2(u, t)]$. Note that composition is in general not commutative.

Given a function $f : 2^A \longrightarrow 2^A$, a *least fixed-point* of f is a minimal set $Z \subseteq A$ such that $f(Z) = Z$ holds. If f is (non-decreasing) monotonic, i.e., $\forall_{S, V \in 2^A}\ S \subseteq V \Rightarrow f(S) \subseteq f(V)$, the existence and uniqueness of a least fixed-point is guaranteed [Tarsk55] and in this case we can use the notation $\mu Z. f(Z)$ to denote this least fixed-point. Likewise, the *greatest fixed-point* of a function $f : 2^A \longrightarrow 2^A$ (if it exists) may be defined by $\nu Z. f(Z)$ as the largest set $Z \subseteq A$ for which $f(Z) = Z$ holds. Note that in terms of characteristic functions, f should have been defined as $f : (A \longrightarrow B) \longrightarrow (A \longrightarrow B)$, which clearly explains the names 'predicate transformer' and 'functional', i.e., function operating on functions.

Let I stand for the identity relation on A, i.e., $I = \{ (a, a) \mid a \in A \}$. Then the reflexive closure R^r of R is defined as: $R^r = R \cup I$. The following holds for the transitive R^+ and reflexive-transitive R^* closures: $R^* = (R^r)^+ = (R^+)^r = R^+ \cup I$. (Note: in general, $R^+ = R^* \setminus I$ does not hold.) The following are equivalent definitions for the transitive closure in terms of a least fixed-point (note the use of the logical operator $\vee$ to stress that we are dealing with characteristic functions):

1. $R^+ = \mu Z. R \vee (Z \circ R)$, or equivalently
2. $R^+ = \mu Z. R \vee (R \circ Z)$, or equivalently
3. $R^+ = \mu Z. R \vee (Z \circ Z)$.

The computation proceeding along the third alternative is known as the iterative squaring method. The first two definitions might be dubbed linear methods. We will present some examples of these methods after we have explained how a fixed-point can be computed.

The set of reachable states, given the set of initial states S_0 and the immediate neighbourhood relation η, can likewise be expressed as a fixed-point:

$\mathsf{Reach} = \mu Z. S_0 \vee \lambda t. \exists_s [Z(s) \wedge \eta(s, t)]$.

This should be read as: the set Reach comprises all the initial states (S_0) united with all the states that are immediate successors ($t \in \eta(s)$) of a reachable state ($Z(s)$).

Expressed in terms of H, using $H(S) = \lambda t. \exists_s [S(s) \wedge \eta(s,t)]$, we can write:

$$\text{Reach} = \mu Z. S_0 \vee H(Z).$$

Of course it is also possible to first compute the transitive closure of η and use it to obtain the reachable states through:

$$\text{Reach} = S_0 \vee \lambda t. \exists_s [S_0(s) \wedge \eta^+(s,t)].$$

This says that the reachable states are the initial states united with all states that can be reached from them by 1 or more applications of η. Note that this form does not seem to involve a fixed-point computation; but, in fact, the least fixed-point is needed in order to calculate η^+.

From fixed-point theory it follows that the least fixed-point of a monotonic function f on a finite set A may be expressed as the limit of repeated applications:

$$\mu Z. f(Z) = \lim_{i \to \infty} f^i(\emptyset),$$

and that there is a least i such that $\forall_{k>i}\, f^k(\emptyset) = f^i(\emptyset)$. Likewise, for the greatest fixed-point we find:

$$\nu Z. f(Z) = \lim_{i \to \infty} f^i(A).$$

Example 5.1 Let's look at the calculation of the transitive closure of the relation $R = \{(0,1), (1,2), (2,3), (3,4)\}$ on the set $A = \{0,1,2,3,4\}$ according the definition $R^+ = \mu Z. R \vee (Z \circ R)$. Using sets instead of their characteristic functions, this reads as $R^+ = \mu Z. R \cup \{(s,t) \mid \exists_u[Z(s,u) \wedge R(u,t)]\}$. The fixed-point iteration proceeds as follows:

$$\begin{aligned}
Z_0 &= \emptyset \\
Z_1 &= R \\
Z_2 &= R \cup \{(s,t) \mid \exists_u[Z_1(s,u) \wedge R(u,t)]\} \\
&= R \cup \{(s,t) \mid \exists_u[R(s,u) \wedge R(u,t)]\} \\
&= R \cup \{(0,2), (1,3), (2,4)\} \\
&= \{(0,1), (0,2), (1,2), (1,3), (2,3), (2,4)\} \\
Z_3 &= R \cup \{(s,t) \mid \exists_u[Z_2(s,u) \wedge R(u,t)]\} \\
&= R \cup \{(0,2), (0,3), (1,3), (1,4), (2,4)\} \\
&= \{(0,1), (0,2), (0,3), (1,2), (1,3), (1,4), (2,3), (2,4)\} \\
Z_4 &= R \cup \{(s,t) \mid \exists_u[Z_3(s,u) \wedge R(u,t)]\} \\
&= R \cup \{(0,2), (0,3), (0,4), (1,3), (1,4), (2,4)\} \\
&= \{(0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4)\} \\
Z_5 &= Z_4
\end{aligned}$$

The calculation according $R^+ = \mu Z. R \vee (R \circ Z)$ proceeds in a similar fashion. However, using the definition $R^+ = \mu Z. R \vee (Z \circ Z)$ we get:

$$\begin{aligned}
Z_0 &= \emptyset \\
Z_1 &= R
\end{aligned}$$

$$\begin{aligned}
Z_2 &= R \cup \{ (s,t) \mid \exists_u [Z_1(s,u) \wedge Z_1(u,t)] \} \\
&= R \cup \{ (s,t) \mid \exists_u [R(s,u) \wedge R(u,t)] \} \\
&= R \cup \{ (0,2), (1,3), (2,4) \} \\
&= \{ (0,1), (0,2), (1,2), (1,3), (2,3), (2,4) \} \\
Z_3 &= R \cup \{ (s,t) \mid \exists_u [Z_2(s,u) \wedge Z_2(u,t)] \} \\
&= R \cup \{ (0,2), (0,3), (0,4), (1,3), (1,4), (2,4) \} \\
&= \{ (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4) \} \\
Z_4 &= Z_3
\end{aligned}$$

We need fewer steps to reach the fixed-point. This can be explained as follows. We can view the calculation of the transitive closure R^+ as a search for paths in the directed graph $G(A,R)$ associated with the relation R. The linear methods consider an extension of the paths by a single edge per fixed-point iteration step; the iterative squaring method considers a potential doubling of the path lengths per step, hence leading to a logarithmic number of steps as compared to the linear methods. The name 'squaring' stems from the form of the definition: $R^+ = \mu Z. R \vee (Z \circ Z) = \mu Z. R \vee Z^2$.
□ example 5.1

Now we will look at how to calculate $\text{Reach} = \mu Z. S_0 \vee H(Z)$ in terms of BDDs. The iterative formulation of a fixed-point computation straightforwardly leads to the very elegant algorithm 5.5 for the reachable state calculation.

```
$2^Q$ Reach ($2^Q$ $S_0$)
{
  S:=∅;
  do {
    S':=$S_0 \cup H(S)$;
    if (S' = S) return S;
    S:=S';
  } forever;
}
```

Algorithm 5.5. $\text{Reach} = \mu Z. S_0 \cup H(Z)$.

With the help of table 5.2 it is easy to express this algorithm in terms of BDD data types. Note that for the correct interpretation of a set as a BDD, we should *a priori* fix a set of place-holder variables $\{ d_i \mid 0 \le i < n \}$, and assume all characteristic functions be expressed in them according to $\chi_S = \lambda d_{n-1}, \ldots, d_0. \cdots$. For instance, the set $S = \{00, 01, 11\}$ has the characteristic function $\chi_S = \lambda d_1, d_0. \neg d_1 \vee d_0$. The BDD only records the expression $\neg d_1 \vee d_0$, and if we interpret this wrongly as $\lambda d_0, d_1. \neg d_1 \vee d_0$ we obtain the set $\{00, 10, 11\}$. We make sure that the set of place-holder variables is disjoint from any other variables that appear as BDD variables. Mind that the ordering of the place-holder variables in principle has no relation with the BDD variable ordering, although it might prove beneficial in practice to conform the BDD variable ordering with the place-holder variable ordering. When applying a functional, care has to be taken to correctly express the BDD of the image set in terms of the variables of the original. For instance when applying H to a set S represented by a BDD in terms of the place-holder variables $\underline{d}$, the result will be a BDD expressed over $\underline{d}$ representing $H(S)$. The parallel substitution of variables $\underline{d}$ by other variables $\underline{x}$ (or other BDDs) in an expression E will be denoted by $E[\underline{d} := \underline{x}]$. The vectors of variables can easily be implemented by some suitable array or list data type.

Set	BDD
$\varnothing$	**BDD_0**
$\overline{S}$	**bdd_not(S)**
$S \cup T$	**bdd_or(S,T)**
$S \cap T$	**bdd_and(S,T)**
$S = T$	**S=T**

Table 5.2. Set notation versus BDD notation.

In case H is not explicitly available, we have to fall back on η and ultimately on the circuit's next-state vector N. Recall that $H(S) = \lambda t.\exists_s [S(s) \wedge \eta(s,t)]$. Given the vector $N = \{N_1, N_2, \cdots N_n\}$ of next-state components $N_i : B^n \times B^m \longrightarrow B$, η can be expressed as $\eta = \lambda s,t.\exists_x \bigwedge_{1 \le i \le n} (N_i(s,x) \equiv t)$.

```
BDD $\eta$(void)
{
  conj:=BDD_1;
  for (i:=1; i<=n; i++)
    conj:=bdd_and(conj,bdd_equiv($N_i[\underline{d}:=\underline{s}\cdot\underline{x}]$,$t_i$));
  return bdd_exist($\underline{x}$,conj)[$\underline{s}\cdot\underline{t}:=\underline{d}$];
}

BDD H(BDD S)
{
  return bdd_exist($\underline{s}$,bdd_and(S[$\underline{d}:=\underline{s}$],$\eta$[$\underline{d}:=\underline{s}\cdot\underline{t}$]))[$\underline{t}:=\underline{d}$];
}

BDD Reach (BDD $S_0$)
{
  S:=BDD_0;
  do {
    S':=bdd_or($S_0$,H(S));
    if (S'=S) return S;
    S:=S';
  } forever;
}
```

Algorithm 5.6. The BDD version of Reach.

This indeed is done in algorithm 5.6 that shows the reachable state calculation in terms of BDDs. Of course the vectors of variables are assumed to be of the appropriate size that is determined by the arity of the characteristic functions that are applied to them, e.g. in the case $\eta[\underline{d} := \underline{s} \cdot \underline{t}]$, $\underline{s}$ and $\underline{t}$ are of size n and are concatenated to match $\underline{d}$ which is of size $2n$.

It is obvious that instead of departing from the formulation $\mathrm{Reach} = \mu Z.\, S_0 \vee H(Z)$, any of the other algorithms presented in this section could have been implemented with BDDs in much the same way. Chapter 7 will show that the fixed-point expressions that we have encountered thus far are special cases of a more general framework, the so-called propositional μ-calculus.

5.6 Product machine verification

Let's go back to the product machine and show how it can be constructed from two separate sequential circuits. Next we show how to modify the algorithms of the previous section to obtain a procedure to verify the equivalence of the two circuits. Each circuit will be assumed to have m inputs and p outputs and also the correspondence among inputs and outputs is known. The number of registers, however, may be different and they may have different state encodings. Let circuit $M_1 = (N_1, \mathrm{Init}_1, Y_1)$ and circuit $M_2 = (N_2, \mathrm{Init}_2, Y_2)$. Circuit M_1 has n_1 register bits, and circuit M_2 has n_2 register bits. The product machine therefore will have $n = n_1 + n_2$ register bits. The concatenation of two bitvectors or function vectors into one larger vector will be denoted by the $\cdot$ operator. The product machine $M = M_1 \times M_2$ is now defined by $M = (N, S_0, Y)$ with:

- $N : B^n \times B^m \longrightarrow B^n$, $N = N_1 \cdot N_2$, i.e.,

  $\forall_{s_1 \in B^{n_1}, s_2 \in B^{n_2}}\ N(s_1 \cdot s_2, x) = N_1(s_1, x) \cdot N_2(s_2, x)$;

- $Y : B^n \times B^m \longrightarrow B$, $Y = \bigwedge_p (Y_1 \equiv Y_2)$, i.e.,

  $\forall_{s_1 \in B^{n_1}, s_2 \in B^{n_2}}\ \forall_{x \in B^m}\ Y(s_1 \cdot s_2, x) = \bigwedge_p (Y_1(s_1, x) \equiv Y_2(s_2, x))$.

  In words, we obtain the single-bit output Y by taking the conjunction over the component-wise logical XNOR of Y_1 and Y_2.

- $S_0 \subseteq B^n$, $S_0 = \{ s_1 \cdot s_2 \mid s_1 \in \mathrm{Init}_1 \wedge s_2 \in \mathrm{Init}_2 \}$.

For equivalence of M_1 and M_2 we require $\forall_{s \in \mathrm{Reach}}\ \forall_{x \in B^m}\ Y(s, x) = 1$. This test is easily incorporated in algorithm 5.5, which results in algorithm 5.7.

We can use the correctness test to define a predicate on states that when true means that the state is 'good', i.e., for that state no input pattern exists that violates the test. Obviously, this predicate can be seen as the characteristic function of the set of good states:

$\mathrm{Good} : B^n \longrightarrow B$, where $\mathrm{Good}(s) = \forall_{x \in B^m}\ Y(s, x)$.

```
Bool Correct ($S_0$)
{
  S:=$\emptyset$;
  do {
    S':=$S_0 \cup H(S)$;
    if ($\neg\forall_{s\in S'}\forall_x Y(s,x)$) return false;
    if (S' = S) return true;
    S:=S';
  } forever;
}
```

Algorithm 5.7. Product machine verification.

In algorithm 5.7 we may therefore replace

```
    if ($\neg\forall_{s\in S'}\forall_x Y(s,x)$) return false;
```

by

```
    if (S' $\not\subseteq$Good) return false;
```

In fact what we have here is a predicate that is supposed to hold for all reachable states. In general, any invariant for the reachable states can be checked likewise. We might conclude that sequential circuit equivalence checking by means of the concept of a product machine is just a special case of model checking. This should not be misunderstood to imply that it is always practical to use a model checker (or model checking algorithms) to do sequential circuit verification. The reason is that our starting point is not the product machine but the separate descriptions of the circuits to be verified. Often building the product machine leads to an unnecessary expansion of the state-space: in the extreme case where the circuits are indeed equivalent and each has its universe as reachable states, traversal of the product machine will be based on a universe $B^{n_1+n_2}$ whereas examination of only the smaller of the 2^{n_1} and 2^{n_2} number of states would have sufficed. Also, since usually we start from a structural description of the circuits under comparison, a preprocessing step could be applied to exploit any similarities in structure to result in a reduction of the eventual product machine [EijkC96].

Chapter 6

Temporal Logic

6.1 Introduction

The idea of temporal logic is to supply a vehicle that allows one to reason about system behaviour as it evolves in time. We are already familiar with traditional propositional logic as a means to reason about combinational circuits. We could roughly say that temporal logic is its counterpart for sequential circuits.

In this chapter we first look at a general structure that defines a model for the subsequent temporal logics. This means that we use the structure to define the semantics of the various logic symbols. The temporal logics that we study in detail are *Computation Tree Logic* (CTL) and *Linear-time Temporal Logic* (LTL).

CTL has been invented with the primary purpose to allow efficient testing of certain system properties. These properties are often classified as *liveness* properties and *safety* properties. Informally speaking, liveness properties express that something good will eventually happen; safety properties express that nothing bad will ever happen. For instance, in a practical situation where we have several processors competing to gain access to a shared bus, we could assert the typical safety property that at any time at most one processor gains access to the bus. A typical liveness property in this case would be to require that every processor eventually gets its turn. Problems of this nature can be solved by a verification method known as *model checking*: properties expressed in CTL are checked against a state model of the system.

CTL belongs to the so-called branching-time temporal logics. If we throw away the possibility to quantify over paths, and in effect only consider a single path of states as our model, the class of so-called *linear-time temporal logics* ensues. A typical representative of this class is LTL. Linear-time temporal logic has a more natural appeal to it: we intuitively consider time to progress along a non-branching time line, i.e., we only consider a single future. LTL therefore is a serious competitor of CTL in describing system behaviour. LTL is less suitable for model checking (the computational complexity is much worse). LTL is commonly used with a satisfiability checker tool. Such a program determines whether an LTL formula can be made true. By complementing the LTL formula input we can use the same tool as a tautology checker.

6.2 A few words about time

As the name suggests, temporal logic has to do with time. The way we model time depends on our application. Throughout this chapter we consider time to be discrete; let time start at some initial timepoint denoted as time instant 0; and assume an infinite future. This discrete time model is convenient because we set out to study state-based models of digital systems. Many of such systems are synchronous, it is therefore natural to let our time model coincide with the ticking of the master system clock. But the model holds also for asynchronous systems and hybrid systems as long as the events of interest can be linearly ordered and mapped onto discrete time points.

6.3 Kripke structures

A Kripke (or temporal) structure is a triple $M = (S, R, L)$ [Wolpe83] with

S : a (possibly infinite) set of states,

$R \subseteq S \times S$: a binary relation that is total, i.e., $\forall_{s \in S}\ \exists_{t \in S}\ (s, t) \in R$, and

L : a state labelling function $S \rightarrow 2^{AP}$, where AP is a set of atomic propositions.

The labelling function L is intended to associate with each state of S an interpretation of the atomic propositions AP, i.e., through L we know for each state which atomic propositions are assigned **true** and which are assigned **false**. The atomic propositions are meant to convey particular facts about the system under study and for now are left without any further interpretation. There are several alternative ways to express the above assignment:

$L : AP \rightarrow 2^S$, which gives for each atomic proposition the states it is assigned **true**.

$L : S \times AP \rightarrow B$, making L a boolean function that evaluates to true when a certain atomic proposition is assigned **true** in a certain state, evaluating to false otherwise.

L : S →(AP → B), which makes L (s) an interpretation function of an atomic proposition at state s.

In the sequel, we will use whichever definition is the most convenient.

A Kripke structure may be viewed as a labelled directed graph: the states are the graph's vertices and the relation R defines the edges. Note that because R is required to be total, each vertex in the graph must have at least one outgoing edge. To map this graph onto our model of time, we single out a certain vertex s_0 and announce that to be our initial state, i.e., the state of the system at timepoint 0. Its immediate successors will then be at timepoint 1, et cetera. This operation will effectively 'unwind' the relation R and cause the graph to be drawn as a tree (figure 6.1).

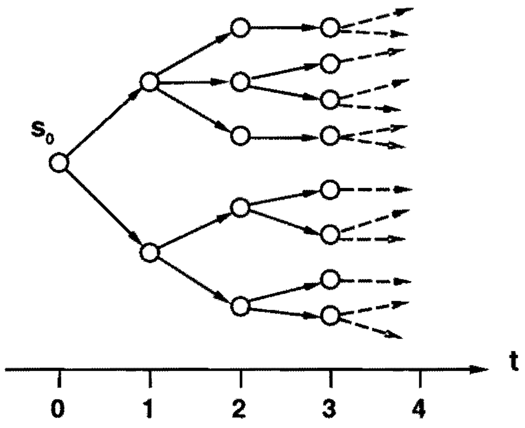


Figure 6.1. Kripke structure 'projected' on time line.

A Kripke structure can be regarded as a state model of a system: the atomic propositions labelling a vertex define the 'state' of the system at that vertex. The possible behaviours of the system are paths through the graph. Such a path will be called an execution trace. Note that a Kripke structure reflects a so-called branching-time model; each state corresponds to a point in time and branches (via its outgoing edges) to a number of possible futures. Note also that a Kripke structure is very similar to a *State Transition Graph* (or state diagram).

6.4 Computation tree logic

Computation tree logic is a logic that is specifically tailored to reason about atomic propositions and their change of 'value' in time as laid down by a given Kripke structure. One also says that the Kripke structure is a model for CTL or that CTL formulas are interpreted over Kripke structures.

The syntax of the most general logic called CTL* consists of 2 groups of only 3 rules each:

S_1 Atomic Propositions and the truth values **false** and **true** are *state* formulas;
S_2 If f, g are *state* formulas, so are $\neg f$, $f \wedge g$, and $f \vee g$;
S_3 If f is a *path* formula then E f, A f are *state* formulas.

P_1 Each *state* formula is a *path* formula;
P_2 If f, g are *path* formulas, so are $\neg f$, $f \wedge g$, and $f \vee g$;
P_3 If f, g are *path* formulas, so are X f, f U g.

The division in state and path formulas has to do with their meaning and will become clear in the sequel. Intuitively, the meaning of a state formula is the set of states in the Kripke structure for which the formula is true; the meaning of a path formula is the set of paths in the Kripke structure for which the formula is true.

What is known as CTL is a restricted form of the above logic. It consists of the same state formulas generated by the rules S_1, S_2, and S_3, but with the rules for the path formulas replaced by a single new rule:

P_0 If f, g are *state* formulas then X f, f U g are *path* formulas.

Observe that the difference in CTL* and CTL syntax is that in the latter path formulas may no longer be nested; they require the use of an E or A operator to make a path formula into a state formula. When no distinction is made between CTL* and CTL we will denote this by CTL$^{(*)}$. We define the language of CTL$^{(*)}$, i.e., the set of all formulas, to be all the *state* formulas generated by the above syntax rules. Hence, from now on when formulas are not explicitly qualified, state formulas are to be understood.

6.4.1 Semantics

Here we will define the semantics for both CTL* and CTL, although we will only be using the simpler logic CTL in the sequel. The meaning of a CTL$^{(*)}$ formula is defined with respect to a Kripke structure $M = (S, R, L)$ with designated initial state s_0. An infinite path in the graph of the Kripke structure will be called a *fullpath*, e.g., $x = (s_0, s_1, \cdots)$ denotes a fullpath starting at state s_0 followed by state s_1 and so on. We use the notation x^i to denote the suffix fullpath $(s_i, s_{i+1}, \cdots)$ of x, i.e., the fullpath x after deletion of a prefix of length i.

In figure 6.2 the semantics of a formula is inductively defined according to the syntax rules. These semantic definitions should be read as follows. For a state formula f, "$M, s_0 \models f$ iff condition" means that the formula f holds in (or is satisfied by) the model M with initial state s_0 when the "condition" is met (= true). Of course, the condition may refer to the model. So the semantic rule S_1 (figure 6.2) says that an atomic proposition (which is itself a state formula) is satisfied by the model (M, s_0) when that atomic proposition is assigned true by the labelling of s_0. For a path formula f, "$M, x \models f$ iff condition" means that the formula f holds for the fullpath x in the model M when the condition is true. So the semantic rule P_3 for the formula $p \, U \, q$ says that $p \, U \, q$ is satisfied by (M, x) when there exists a suffix fullpath x^i such that $M, x^i \models q$ holds and for all suffix fullpaths x^j, such that $0 \leq j < i$, $M, x^j \models p$ holds. This is clearly an inductive definition.

S_1	$M, s_0 \models \text{false}$	$\equiv$	false	
	$M, s_0 \models \text{true}$	$\equiv$	true	
	$M, s_0 \models p$	iff	$p \in L(s_0)$	
S_2	$M, s_0 \models \neg f$	iff	not $M, s_0 \models f$	
	$M, s_0 \models f \wedge g$	iff	$M, s_0 \models f$ and $M, s_0 \models g$	
	$M, s_0 \models f \vee g$	iff	$M, s_0 \models f$ or $M, s_0 \models g$	
S_3	$M, s_0 \models E f$	iff	$\exists_{x = (s_0, s_1, \cdots)} M, x \models f$	
	$M, s_0 \models A f$	iff	$\forall_{x = (s_0, s_1, \cdots)} M, x \models f$	
P_1	$M, x \models f$	iff	$M, s_0 \models f$	with $x = (s_0, s_1, \cdots)$
P_2	$M, x \models \neg f$	iff	not $M, x \models f$	
	$M, x \models f \wedge g$	iff	$M, x \models f$ and $M, x \models g$	
	$M, x \models f \vee g$	iff	$M, x \models f$ or $M, x \models g$	
P_3	$M, x \models X f$	iff	$M, x^1 \models f$	
	$M, x \models f \, U \, g$	iff	$\exists_{i \geq 0} M, x^i \models g$ and $\forall_{j<i} M, x^j \models f$	
P_0	$M, x \models X f$	iff	$M, s_1 \models f$	with $x = (s_0, s_1, \cdots)$
	$M, x \models f \, U \, g$	iff	$\exists_{i \geq 0} M, s_i \models g$ and $\forall_{j<i} M, s_j \models f$	with $x = (s_0, s_1, \cdots)$

Figure 6.2. Semantics of $CTL^{(*)}$.

A $CTL^{(*)}$ formula ϕ is said to be satisfiable iff there exists a model for it, i.e., there exists a Kripke structure M and a state s such that $M, s \models \phi$ holds.

A $CTL^{(*)}$ formula ϕ is said to be valid (the word "tautology" would be appropriate as well) iff for every structure M and for every state s of M, $M, s \models \phi$ holds.

When needed, these definitions can easily be rephrased for path formulas.

It can be shown that CTL is strictly weaker in expressiveness than CTL^*. So, there are properties that can be expressed as a CTL^* formula but no equivalent CTL formula exists. An example will be presented at the end of the next section.

6.4.2 CTL operators

In CTL, path formulas cannot be nested, e.g. p U (q U r) is not allowed by the CTL syntax. CTL path formulas are constructed using the X and U operators and must immediately be preceded by a unary E or A operator to turn them into state formulas. We therefore combine these possibilities into 4 separate operators with a slightly different notation for ease of writing:

E (X f)	becomes	EX f
E (f U g)	becomes	f EU g
A (X f)	becomes	AX f
A (f U g)	becomes	f AU g

The abstract syntax of CTL may now be expressed by the single BNF production rule:

$$\begin{array}{lll} \mathrm{CTL} & ::= & \mathrm{AP} \mid \neg\mathrm{CTL} \mid \mathrm{CTL} \wedge \mathrm{CTL} \\ & \mid & \mathrm{CTL} \vee \mathrm{CTL} \mid \mathrm{EX\,CTL} \mid \mathrm{AX\,CTL} \mid \mathrm{CTL\,EU\,CTL} \mid \mathrm{CTL\,AU\,CTL}\,. \end{array}$$

In practice, one often chooses a different set of basic CTL operators. Here we select the operators EX, EG, and EU. EG is a new operator we haven't seen before. It is intended to express that *there exists a fullpath such that the operand holds for all states on that path.* A more formal definition is given in figure 6.3.

$$\boxed{\begin{array}{lll} M, s_0 \models \mathrm{EX}\,f & \text{iff} & \underset{(s_0,s_1,\cdots)}{\exists}\ M, s_1 \models f \\ M, s_0 \models \mathrm{EG}\,f & \text{iff} & \underset{(s_0,s_1,\cdots)}{\exists}\ \underset{i\geq 0}{\forall} M, s_i \models f \\ M, s_0 \models f\,\mathrm{EU}\,g & \text{iff} & \underset{(s_0,s_1,\cdots)}{\exists}\ \underset{j\geq 0}{\exists} M, s_j \models g \text{ and } \underset{i<j}{\forall} M, s_i \models f \end{array}}$$

Figure 6.3. Selection of basic CTL operators.

It is easily derived that $\mathrm{EG}\,f = \neg\,(\mathrm{true\,AU}\,\neg f)$. We will further introduce the truth values false and true and include the following set of derived operators:

$$\begin{array}{lll} \mathrm{EF}\,f & = & \mathrm{true\,EU}\,f \\ \mathrm{AX}\,f & = & \neg\mathrm{EX}\neg f \\ \mathrm{AG}\,f & = & \neg\mathrm{EF}\neg f \\ f\,\mathrm{AU}\,g & = & \neg\,((\neg g\,\mathrm{EU}\,(\neg f \wedge \neg g)) \vee \mathrm{EG}\neg g) \\ \mathrm{AF}\,f & = & \mathrm{true\,AU}\,f \end{array}$$

In total we now have 8 temporal operators in our version of CTL.

Example 6.1 A property that can be expressed in CTL* but not in CTL is the following: an atomic proposition, say p, is true infinitely often along a path. To be precise, we claim the (state) formula E G Fp to be true for the initial state of a certain model. (In terms of the original operators the formula reads $E\neg(\text{true}\,U\neg(\text{true}\,U\,p))$.) A simple model that satisfies the property is drawn in figure 6.4.

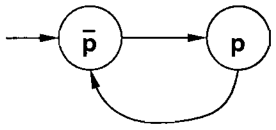


Figure 6.4. Model that satisfies E G Fp.

The closest CTL formulas that come to mind are EG EF p and EG AF p. However, it should be intuitively clear that both formulas do not require the path that satisfies the EF (resp. AF) subformula to coincide with the path that satisfies EG. Because of the restrictive syntax of CTL to prefix path formulas with a quantor, it is not possible to impose that the selected paths being quantified will be one and the same path. Figure 6.5. shows a model for EG EF p that is not also a model for E G Fp.

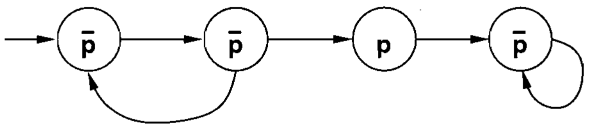


Figure 6.5. Model that satisfies EG EF p but not E G Fp.

□ example 6.1

6.5 CTL model checking

Our purpose of introducing CTL is to arrive at a method for automatically verifying properties of systems. If the systems we are considering have a finite set of states, the behaviour of such a system may be modelled by a finite Kripke structure M. Often we know that the system starts in some initial state s_0. Then a certain property of the system may be expressed as a CTL formula ϕ and checked against the model, i.e., we try to prove that $M, s_0 \models \phi$.

On the other hand, observe that any CTL formula ϕ could also be interpreted within a given Kripke structure M as denoting a set of states, namely those states s for which $M, s \models \phi$ holds. We therefore define:

$Q(\phi) \subseteq S$ is the set of states associated with CTL formula ϕ, such that:

$Q(\phi) = \{ s \mid M, s \models \phi \}$.

One way to find out whether a certain property ϕ holds for a given system, is to compute $Q(\phi)$ and check whether $s_0 \in Q(\phi)$. The Q sets for each possible form of CTL formula are easily derived from the semantics. Figure 6.6 provides a complete list based on a given Kripke structure $M = (S, R, L)$. We use the notation $R(s)$ to stand for the set $\{t \in S \mid (s,t) \in R\}$.

$$
\begin{array}{lcl}
Q(\text{false}) & = & \varnothing \\
Q(\text{true}) & = & S \\
Q(p) & = & \{s \in S \mid p \in L(s)\} \\
Q(\neg f) & = & S \setminus Q(f) \\
Q(f \wedge g) & = & Q(f) \cap Q(g) \\
Q(f \vee g) & = & Q(f) \cup Q(g) \\
Q(EXf) & = & \{s \in S \mid R(s) \cap Q(f) \neq \varnothing\} \\
Q(EGf) & = & Q(f) \cap Q(EX\,EGf) \\
Q(f\,EU\,g) & = & Q(g) \cup Q(f) \cap Q(EX\,(f\,EU\,g))
\end{array}
$$

Figure 6.6. State-sets for the basic CTL formulas.

The last two equations are recurrent. Luckily their solutions are well-defined and can easily be computed as we shall see in the next section. For now we assume the existence of the functions Q_{EX}, Q_{EG}, and Q_{EU} defined in figure 6.7.

$$
\begin{array}{lcl}
Q_{EX}(Q(f)) & = & Q(EXf) \\
Q_{EG}(Q(f)) & = & Q(EGf) \\
Q_{EU}(Q(f), Q(g)) & = & Q(f\,EU\,g)
\end{array}
$$

Figure 6.7. Auxiliary state-set functions.

The association of a state-set with a formula is implemented by algorithm 6.1.

```
2^S Q(CTL f)
{
  switch (f) {
  case false:           return ∅;
  case true:            return S;
  case P:               return {s ∈ S | P ∈ L(s)};
  case ¬g:              return S\Q(g);
  case g ∧ h:           return Q(g) ∩ Q(h);
  case g ∨ h:           return Q(g) ∪ Q(h);
  case EXg:             return Q_EX (Q(g));
  case EGg:             return Q_EG (Q(g));
  case gEUh:            return Q_EU (Q(g),Q(h));
  }
}
```

Algorithm 6.1. Derivation of state-set from a CTL formula.

6.5.1 Model checking algorithms

The state-sets for EG and EU are expressed as recurrent equations. These equations are derived from the following logical equivalences for these operators:

$$\mathrm{EG}\,f = f \wedge \mathrm{EX}\,\mathrm{EG}\,f$$

$$f\,\mathrm{EU}\,g = g \vee f \wedge \mathrm{EX}\,(f\,\mathrm{EU}\,g)$$

One might look at these equations as theorems of CTL and prove them by resorting to their semantic definitions. Note that $Q(\mathrm{EX}f)$ is defined to be those states that have at least one successor that belongs to $Q(f)$. In other words, $Q(\mathrm{EX}f)$ is the *image* of $Q(f)$ under the converse relation R^{-1}; this is usually called the *pre-image* under R. Apart from R^{-1} we will use the following notation for the various sets associated with R:

$R \subseteq A \times A$	Relation
$R : A \rightarrow 2^A$	Function
$R : 2^A \rightarrow 2^A$	Extended function
$R(s) = \{t \in S \mid (s,t) \in R\}$	(definition)
$R(S) = \bigcup_{s \in S} R(s)$	(extension)
$R^0(S) = S$	Identity
$R^1(S) = R(S)$	Image
$R^k(S) = R^{k-1}(R(S))$	Iterated application
$R^{-1}(t) = \{s \in S \mid (s,t) \in R\}$	(converse)
$R^{-1}(T) = \bigcup_{t \in T} R^{-1}(t)$	Pre-image

Using $Q(\mathrm{EX}f) = R^{-1}(Q(f))$ and identifying R^{-1} with Q_{EX}, we can rewrite the state-sets for EG and EU as follows:

$$Q(\mathrm{EG}f) = Q(f) \cap Q_{EX}(Q(\mathrm{EG}f))$$

$$Q(f\,\mathrm{EU}\,g) = Q(g) \cup Q(f) \cap Q_{EX}(Q(f\,\mathrm{EU}\,g))$$

To solve these equations we can apply fixed-point theory. Assuming that $Q(f)$ and $Q(g)$ are known, i.e., the terms may be considered constant, say Q_f and Q_g, we are dealing with functions F of signature $2^S \rightarrow 2^S$ for which we like to find a fixed-point value. Our task is to solve the following fixed-point equations:

$$Z_{EG} = F_{EG}(Z_{EG}) = Q_f \cap Q_{EX}(Z_{EG})$$

$$Z_{EU} = F_{EU}(Z_{EU}) = Q_g \cup Q_f \cap Q_{EX}(Z_{EU})$$

Without proof we here state the correct fixed-point characterizations of the state-sets for the EG and EU operators:

$Q(EG\,f) = \nu Z.\, Q_f \cap Q_{EX}(Z)$, i.e , a *greatest fixed-point* computation,

$Q(f\,EU\,g) = \mu Z.\, Q_g \cup Q_f \cap Q_{EX}(Z)$, i.e. , a *least fixed-point* computation.

The computation of $Q(EG\,f)$ proceeds as follows. We start with our initial approximation $Z_0 = S$ because we have a greatest fixed-point at hand. Then we calculate $Z_1 = Q_f \cap R^{-1}(Z_0) = Q_f \cap R^{-1}(S)$. Using this result, we calculate Z_2, and so on, till we find at some step $k \geq 0$ that $Z_{k+1} = Z_k$, in which case we are done and the solution is Z_k. The function $Q_{EG} = \lambda Q_f.\, \nu Z.\, Q_f \cap Q_{EX}(Z)$ is presented in pseudo-C code in Algorithm 6.2.

```
$2^S$ $Q_{EG}$ ($2^S$ $Q_f$)
{
  for (k:=0, $Z_k$:=S;; k++) {
    $Z_{k+1} := Q_f \cap Q_{EX}(Z_k)$;
    if ($Z_{k+1}=Z_k$) return $Z_k$;
  }
}
```

Algorithm 6.2. Greatest fixed-point calculation of $Q(EG\,f)$.

In an analogous way we can derive the procedure to compute $Q(f\,EU\,g)$ by defining the function $Q_{EU} = \lambda Q_f, Q_g.\, \mu Z.\, Q_g \cup Q_f \cap Q_{EX}(Z)$. This is shown in algorithm 6.3.

```
$2^S$ $Q_{EU}$ ($2^S$ $Q_f$, $2^S$ $Q_g$)
{
  for (k:=0, $Z_k$:=$\emptyset$;; k++) {
    $Z_{k+1} := Q_g \cup (Q_f \cap Q_{EX}(Z_k))$;
    if ($Z_{k+1}=Z_k$) return $Z_k$;
  }
}
```

Algorithm 6.3. Least fixed-point calculation of $Q(f\,EU\,g)$.

The auxiliary routine Q_{EX} simply returns the pre-image of a set:

```
$2^S$ $Q_{EX}$ ($2^S$ $Q_f$)
{
  return $R^{-1}(Q_f)$;
}
```

Algorithm 6.4. Pre-image calculation.

Note that for a Kripke structure $R^{-1}(S) = S$ and $R^{-1}(\emptyset) = \emptyset$. Therefore we could have slightly simplified the above algorithms by using a different initialization and then skipping the first iteration step. Apart from the need to calculate R^{-1}, i.e., the pre-image, the algorithms solely use set operations which may be implemented in variety of ways. In symbolic model-checking the choice is to use BDDs.

6.6 Linear-time temporal logic

As stated in the introduction, LTL and CTL are closely related. Both are restricted versions of the more general logic CTL*. We will here use the more classical notation for the LTL operators [Manna81]: $\Box$, pronounce *always*, instead of the CTL*'s G; $\Diamond$, pronounce *sometime*, instead of CTL*'s F; U, pronounce (*strong*) *until*; and $\circ$, pronounce *next*, instead of CTL*'s X. The U operator is the so-called strong until operator. We will also include the weak until operator U_w.

The truth of a temporal formula is determined by the truth values of its atomic propositions which may vary from time instant to time instant. The temporal operators can informally defined by:

For any temporal formula f and g,

$\Box$f, is the proposition that for every time instant, now and in the future, the formula f will be true,

$\Diamond$f, is the proposition that the formula f will ever, perhaps now but definitely (if not now) sometime in the future, become true,

f U g, is the proposition that the formula f will be true at least for all time instants from the present until (but not necessarily including) the time when g becomes true, and the latter must inevitably happen,

f U_w g, is the proposition that the formula f will be true at least for all time instants from the present until (but not necessarily including) the time when g becomes true, but the latter need not ever happen,

$\circ$f, is the proposition that f is true in the next time instant, i.e. the one immediately following the present,

a propositional formula f is true in LTL if it is true in the present time instant, irrespective of whatever its variables' values will be in the future.

Propositional temporal logic may be completely formalized in a manner very similar to propositional logic, defining a number of axiom schemata and rules of inference. It can be proven that in this way a sound and complete theory is established. A decision procedure is available that determines for each temporal formula whether it is a theorem or not. This decision problem is shown to be PSPACE-complete (a class of problems that includes the NP problems [Garey79]) in [Sistl85]. We will present a program that decides the satisfiability of an LTL formula in section 6.7.

We define LTL as a language over an alphabet of atomic propositions AP. For the latter we use the same definition as with propositional logic. The syntax of LTL formulas is shown in figure 6.8.

$$
\begin{array}{lll}
\text{formula} & ::= & \text{term } [\ \vee \text{ formula }]\ . \\
\text{term} & ::= & \text{factor } [\ \wedge \text{ term }]\ . \\
\text{factor} & ::= & \text{primary } [\ (\ \mathrm{U} \mid \mathrm{U}_w\ )\ \text{factor }]\ . \\
\text{primary} & ::= & \text{false} \\
 & \mid & \text{true} \\
 & \mid & \mathrm{P} \\
 & \mid & \neg\ \text{primary} \\
 & \mid & \circ\ \text{primary} \\
 & \mid & \diamond\ \text{primary} \\
 & \mid & \Box\ \text{primary} \\
 & \mid & (\ \text{formula}\ )\ .
\end{array}
$$

where P is an atomic proposition taken from AP.

Figure 6.8. LTL Formula Syntax.

The meaning of a temporal logic formula is defined with respect to a Kripke structure $M = (S, R, L)$, where S is a finite set of states, $R : S \rightarrow S$ a total successor function giving for each state a *unique* next state and $L : S \rightarrow 2^{AP}$ a labelling of a state with a set of atomic propositions true in that state.

The truth of an LTL formula is inductively defined relative to a structure M and a state s by figure 6.9.

If the set of states S is finite and the successor relation is a total function, any infinite sequence of time instants, to be more precise: any infinite sequence of occurrences of states, may be represented in a finite way by a ω-regular string over the alphabet S, i.e., it consists of a certain possible empty prefix sequence followed by an endless repetition of a cycle of 1 or more states. This can be depicted by a lasso-shaped graph.

An LTL formula f is satisfiable, i.e. can be made true, if we can find a model (M, s_0) such that $M, s_0 \models f$ is true. If a formula is true in a model we also say that the model, or sequence of states with associated truth-assignment (since that uniquely determines the model), verifies or satisfies the formula.

A formula is said to be valid iff it is true in every model, notation: $\models f$. We will adopt the term tautology introduced in propositional logic for valid formulas. A formula that cannot be satisfied by any model is a contradiction. Two formulas f and g are said to be equivalent, notation $f = g$, when $\models (f \leftrightarrow g)$ holds. Note that a

$M, s \models \text{false}$	$\equiv$	false
$M, s \models \text{true}$	$\equiv$	true
$M, s \models p$	iff	$p \in L(s)$ (for $p \in AP$)
$M, s \models \neg f$	iff	not $M, s \models f$
$M, s \models f \vee g$	iff	$M, s \models f$ or $M, s \models g$
$M, s \models f \wedge g$	iff	$M, s \models f$ and $M, s \models g$
$M, s \models \circ f$	iff	$M, R(s) \models f$
$M, s \models \Diamond f$	iff	$\underset{i \geq 0}{\exists} M, R^i(s) \models f$
$M, s \models \Box f$	iff	$\underset{i \geq 0}{\forall} M, R^i(s) \models f$
$M, s \models f \ U \ g$	iff	$\underset{i \geq 0}{\exists} (M, R^i(s) \models g$ and $\underset{0 \leq j < i}{\forall} M, R^j(s) \models f)$
$M, s \models f \ U_w \ g$	iff	$M, s \models (f U g) \vee \Box f$

where $R^i(s)$ denotes the i^{th} successor of s.

Figure 6.9. LTL Formula Semantics.

formula is unsatisfiable if and only if its negation is a tautology and conversely a formula is valid iff its negation is unsatisfiable.

6.7 An LTL satisfiability checker

The satisfiability problem of an LTL formula asks whether there exists a truth assignment to the atomic propositions in the formula at each time instant that make the formula true. Here we will describe an algorithm for checking the satisfiability of an LTL formula. We can distinguish three main steps:

1. Parsing. In this phase a formula is converted to a binary tree.
2. Normalization and optimization. The formula is converted to negation normal form and a number of optimizations are performed.
3. Model construction. In this phase the actual model is constructed.

These steps will now be explained in more detail.

6.7.1 Parsing an LTL formula

The lexical analyser and parser routines are generated by the UNIX utilities *lex* and *yacc* from the LTL token and grammar definition files. The parser constructs a rooted ordered tree representation for the input formula. Note that in the algorithms and examples of this section we use the alternative 'printable' symbols to denote the LTL operators as shown in table 6.1.

Math. notation:	ptl notation:	Meaning:
$\neg$	!	Logical negation
$\wedge$	&	Conjunction
$\vee$	V	Disjunction
$\circ$	@	Next
$\diamond$	<>	Sometime
$\Box$	[]	Always
U	U	Strong until
U_w	Uw	Weak until

Table 6.1. Notation of LTL operators as used by the program.

6.7.2 Normalization

Propagate all NOT operators towards the leaf nodes of the formula tree. Afterwards a NOT operator may only appear directly in front of a variable node. The formula is then said to be in *negation normal form* (nnf). The following identities are used in this process:

$$
\begin{array}{lcllcl}
\neg \text{false} & = & \text{true}, & \neg \text{true} & = & \text{false}, \\
\neg\neg f & = & f, & \neg \circ f & = & \circ \neg f, \\
\neg \diamond f & = & \Box \neg f, & \neg \Box f & = & \diamond \neg f, \\
\neg (f \wedge g) & = & \neg f \vee \neg g, & \neg (f \vee g) & = & \neg f \wedge \neg g, \\
\neg (f\, U\, g) & = & \neg g\, U_w\, (\neg f \wedge \neg g), & \neg (f\, U_w\, g) & = & \neg g\, U\, (\neg f \wedge \neg g).
\end{array}
$$

We use the two, mutually recursive, routines of algorithm 6.5 and algorithm 6.6.

```
Tree neg(Tree f)
{
  switch (f) {
  case  <false>: return <true>;
  case   <true>: return <false>;
  case      <p>: return <! p>;
  case   <!  g>: return nnf(g);
  case   <@  g>: return <@  neg(g)>;
  case   <<> g>: return <[] neg(g)>;
  case   <[] g>: return <<> neg(g)>;
  case <U   g h>: return <Uw neg(h) <& neg(g) neg(h)>>;
  case <Uw  g h>: return <U  neg(h) <& neg(g) neg(h)>>;
  case <&   g h>: return <+  neg(g) neg(h)>;
  case <+   g h>: return <&  neg(g) neg(h)>;
  }
}
```

Algorithm 6.5. Negate formula f.

```
Tree nnf(Tree f)
{
  switch (f) {
  case <false>:
  case  <true>:
  case     <p>: return f;
  case  <!  g>: return neg(g);
  case  <@  g>: return <@  nnf(g)>;
  case  <<> g>: return <<> nnf(g)>;
  case  <[] g>: return <[] nnf(g)>;
  case <U   g h>: return <U  nnf(g) nnf(h)>;
  case <Uw  g h>: return <Uw nnf(g) nnf(h)>;
  case <&   g h>: return <&  nnf(g) nnf(h)>;
  case <+   g h>: return <+  nnf(g) nnf(h)>;
  }
}
```

Algorithm 6.6. Convert formula f to negation normal form.

6.7.3 Optimization

Exhaustively apply the rewrite rules listed in figure 6.10 to the formula in nnf. Optionally, delete all $\diamond$ and $\circ$ operators at the top of the formula tree. Use the meta-identities:

$$\begin{array}{lll} \diamond f \text{ satisfiable} & \text{iff} & f \text{ satisfiable} \\ \circ f \text{ satisfiable} & \text{iff} & f \text{ satisfiable} \end{array}$$

The next step in the optimization process is the reduction of the formula tree to a directed acyclic graph (DAG). All isomorphic subtrees (subformulas) modulo commutativity of AND and OR operators are identified and collapsed. Also nodes with identical left and right children are deleted and replaced by their unique child. The latter is justified because of the identities:

$$\begin{array}{lll} f \wedge f & = & f \\ f \vee f & = & f \\ f \,U\, f & = & f \\ f \,U_w\, f & = & f \end{array}$$

This effectively converts the formula tree into a DAG. The advantage clearly is a reduction of the number of subformulas to be considered in the model construction phase. For brevity and clarity, all algorithms on DAGs in the sequel are presented without the necessary marking of nodes to avoid visiting the same node more than once.

$\neg$false	$\Rightarrow$	true	$\circ$false	$\Rightarrow$	false
$\neg$true	$\Rightarrow$	false	$\circ$true	$\Rightarrow$	true.
$\Diamond$false	$\Rightarrow$	false	$\Box$false	$\Rightarrow$	false
$\Diamond$true	$\Rightarrow$	true	$\Box$true	$\Rightarrow$	true
$\Diamond\circ f$	$\Rightarrow$	$\circ\Diamond f$	$\Box\circ f$	$\Rightarrow$	$\circ\Box f$
$\Diamond\Diamond f$	$\Rightarrow$	$\Diamond f$	$\Box\Box f$	$\Rightarrow$	$\Box f$
$\Diamond\Box\Diamond f$	$\Rightarrow$	$\Box\Diamond f$	$\Box\Diamond\Box f$	$\Rightarrow$	$\Diamond\Box f$
$\Diamond(f \mathbin{U} g)$	$\Rightarrow$	$\Diamond g$	$\Box(f \mathbin{U_w} g)$	$\Rightarrow$	$\Box(f\vee g)$
false U f	$\Rightarrow$	f	false U_w f	$\Rightarrow$	f
f U false	$\Rightarrow$	false	f U_w false	$\Rightarrow$	$\Box f$
true U f	$\Rightarrow$	$\Diamond f$	true U_w f	$\Rightarrow$	true
f U true	$\Rightarrow$	true	f U_w true	$\Rightarrow$	true
$\circ f \mathbin{U} \circ g$	$\Rightarrow$	$\circ$	$\circ f \mathbin{U_w} \circ g$	$\Rightarrow$	$\circ(f \mathbin{U_w} g)$
$f \mathbin{U} \Diamond g$	$\Rightarrow$	$\Diamond g$	$(\Box f) \mathbin{U_w} g$	$\Rightarrow$	$\Box f\vee g$
$p \mathbin{U} \neg p$	$\Rightarrow$	$\Diamond\neg p$	$p \mathbin{U_w} \neg p$	$\Rightarrow$	true
$\neg p \mathbin{U} p$	$\Rightarrow$	$\Diamond p$	$\neg p \mathbin{U_w} p$	$\Rightarrow$	true
$f \mathbin{U} f$	$\Rightarrow$	f*	$f \mathbin{U_w} f$	$\Rightarrow$	f*
false $\wedge f$	$\Rightarrow$	false	false $\vee f$	$\Rightarrow$	f
true $\wedge f$	$\Rightarrow$	f	true $\vee f$	$\Rightarrow$	true
$f\wedge$ false	$\Rightarrow$	false	$f\vee$ false	$\Rightarrow$	f
$f\wedge$ true	$\Rightarrow$	f	$f\vee$ true	$\Rightarrow$	true
$\circ f\wedge\circ g$	$\Rightarrow$	$\circ(f\wedge g)$	$\circ f\vee\circ g$	$\Rightarrow$	$\circ(f\vee g)$
$\Box f\wedge\Box g$	$\Rightarrow$	$\Box(f\wedge g)$	$\Diamond f\vee\Diamond g$	$\Rightarrow$	$\Diamond(f\vee g)$
$p\wedge p$	$\Rightarrow$	p	$p\vee p$	$\Rightarrow$	p
$\neg p\wedge\neg p$	$\Rightarrow$	$\neg p$	$\neg p\vee\neg p$	$\Rightarrow$	$\neg p$
$\neg p\wedge p$	$\Rightarrow$	false	$\neg p\vee p$	$\Rightarrow$	true
$p\wedge\neg p$	$\Rightarrow$	false	$p\vee\neg p$	$\Rightarrow$	true
$f\wedge f$	$\Rightarrow$	f*	$f\vee f$	$\Rightarrow$	f*

Figure 6.10. Rewrite rules. (* indicates 'done during reduction'.)

6.7.4 Model construction

An LTL formula is satisfiable when we can construct an infinite path of states such that all eventualities on that path are fulfilled. Eventualities are subformulas of the kind $\Diamond$ and U. Our approach will be to construct not just a single path but all possible paths as a graph in one go. Edges and vertices of that graph are associated with the disjunctive normal form representation of a temporal formula defined in this section.

First we identify the propositional subformulas and the so-called elementary subformulas in the DAG for a formula. They are defined by means of algorithm 6.7 and algorithm 6.8 respectively. In the ptl program we represent a propositional subformula by a BDD over the set of atomic propositions. Our definition of elementary subformulas is slightly different from the usual one [Burch91]. For one, we don't regard atomic propositions as elementary, and also we don't consider all $\circ$ operators elementary but use their operands instead.

```
Bool propositional(Dag f)
{
  switch (f) {
  case  <false>:
  case   <true>:
  case      <p>:
  case    <! p>: return true;
  case   <@  g>:
  case   <<> g>:
  case   <[] g>:
  case <U   g h>:
  case <Uw  g h>: return false;
  case <&   g h>:
  case <+   g h>: return propositional(g) && propositional(h);
  }
}
```

Algorithm 6.7. Test whether formula f is propositional.

With these definitions of the propositional subformulas and elementary subformulas of a formula f it is possible to express any LTL formula f in the following disjunctive normal form for certain propositional formulas p_i, p_k (which may be true and in that case are omitted) and elementary g_j:

$$f = \bigvee_i \, (p_i \wedge \circ \, (p_k \wedge \bigwedge_j g_j))$$

Moreover, every formula that results from expanding a $\Diamond$, $\Box$, U, or U_w subformula according

$$\begin{array}{lcl} \Diamond f & = & f \vee \circ \Diamond f \\ \Box f & = & f \wedge \circ \Box f \\ f \,\mathsf{U}\, g & = & g \vee f \wedge \circ \, (f \,\mathsf{U}\, g) \\ f \,\mathsf{U}_w\, g & = & g \vee f \wedge \circ \, (f \,\mathsf{U}_w\, g) \end{array}$$

can again be written in that very same form. For later reference, we mention that the term of which $\circ$ is part in the above expansions will be referred to as the inductive term, the other term (g) is called the finite term.

Example 6.2 Consider the formula $\neg \Diamond \neg p \vee \Box q \wedge \circ \, (p \,\mathsf{U} \neg \Box q) \vee \neg \circ \, (\neg p \wedge q)$. We will subject this formula to the processing steps described thus far. The formula in negation normal form reads $\Box p \vee \Box q \wedge \circ \, (p \,\mathsf{U} \Diamond \neg q) \vee \circ \, (p \vee \neg q)$. This may be optimized to $\Box p \vee \Box q \wedge \circ \Diamond \neg q \vee \circ \, (p \vee \neg q)$. The propositional subformulas are easily determined to be $\{ p, q, \neg q, p \vee \neg q \}$; the elementary subformulas are $\{ \Box p \vee \Box q \wedge \circ \Diamond \neg q \vee \circ \, (p \vee \neg q), \Box p, \Box q, \Diamond \neg q \}$. Using the above 'expansion' rules the formula can be written as $p \wedge \circ \Box p \vee q \wedge \circ \Box q \wedge \circ \Diamond \neg q \vee \circ \, (p \vee \neg q)$ which indeed has the required (top-level) sum-of-products of propositional and

```
Set elem_1(Dag f)
{
  if (propositional(f)) return ∅;

  switch (f) {
  case   <@  g>:
    if (propositional(g)) return ∅;
    switch (g) {
    case   <@  h>: return {g}∪elem_1(h);
    case   <<> h>:
    case   <[] h>:
    case <U  h i>:
    case <Uw h i>: return elem_1(g);
    case <&  h i>:
    case <+  h i>: return {g}∪elem_1(h)∪elem_1(i);
    }
  case   <<> g>:
  case   <[] g>: return {f}∪elem_1(g);
  case <U  g h>:
  case <Uw g h>: return {f}∪elem_1(g)∪elem_1(h);
  case <&  g h>:
  case <+  g h>: return elem_1(g)∪elem_1(h);
  }
}

Set elem(Dag f)
{
  if (propositional(f)) return ∅;

  return {f}∪elem_1(f);
}
```

Algorithm 6.8. Determine elementary formulas of f.

elementary subformulas form:

Prop.$\wedge$	$\circ$ Prop.$\wedge$	$\circ \bigwedge$Elementary
p	true	$\{\Box p\}$
q	true	$\{\Box q, \Diamond\neg q\}$
true	$p \vee \neg q$	$\emptyset$

□ example 6.2

Every conjunct in the disjunctive normal form defines an (edge, vertex) pair; the edge represents the propositional formula p_i, a vertex represents the term $p_k \wedge \bigwedge_j g_j$. The initial vertex of the graph represents the formula f under test.

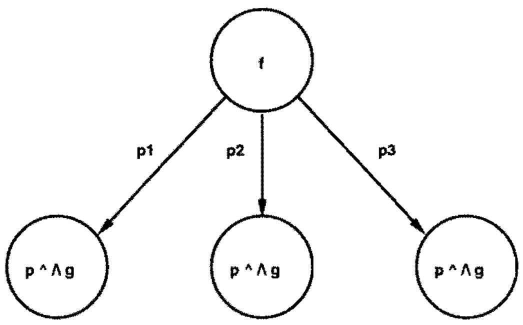


Figure 6.11. Graphical representation of a formula f in disjunctive normal form.

(Technically ○f is represented, but we have already seen that f is satisfiable whenever ○f is.) Figure 6.11 visualizes the aforementioned interpretation of the temporal disjunctive normal form. Note that all vertices, possibly with the exception of the initial vertex, will be labelled by a conjunct consisting of a propositional formula and zero or more elementary formulas. Actually, to get a more uniform vertex labelling one could decide to introduce multiple initial vertices, one for each conjunct in the DNF of the formula f, and then check each for satisfiability separately.

The graph construction proceeds by considering the conjunct labelling a vertex and converting it to disjunctive normal form. This then gives us a number of (edge, vertex) pairs that are the outgoing edges and immediate successors of the vertex under consideration. Note that pairs for which the edge label is identically false need not be included in the graph: they express an unsatisfiable continuation. This can easily be checked when the edge labels are represented by BDDs. A similar remark holds for the propositional subformula that is part of a vertex label: when it is found to be unsatisfiable, again the (edge, vertex) pair is discarded since that vertex cannot have any successors. The maximum number of distinct vertices equals the cardinality of the powerset of the atomic propositions together with the elementary subformulas, and hence is finite. By keeping all vertex labels unique (using a hash table) no duplicate vertices will be created.

Example 6.3 Following up on the previous example, we will now show how the model graph for the formula $\neg\Diamond\neg p \vee \Box q \wedge \circ(p\,\mathcal{U}\,\neg\Box q) \vee \neg\circ(\neg p \wedge q)$ is constructed. The initial vertex (1) is labelled with the formula $\Box p \vee \Box q \wedge \circ\Diamond\neg q \vee \circ(p \vee \neg q)$. This results in 3 outgoing edges:

Edge labelled:	To vertex:
p	(4): $\{\Box p\}$
q	(5): $\{\Box q, \Diamond\neg q\}$
true	(2): $\{p \vee \neg q\}$

Converting each new vertex to DNF gives:

Vertex:	Label:	DNF:
(4)	$\{\Box p\}$	$p \wedge \circ\Box p$
(5)	$\{\Box q, \Diamond\neg q\}$	$q \wedge \circ(\Box q \wedge \Diamond\neg q)$
(2)	$\{p \vee \neg q\}$	$(p \vee \neg q) \wedge \circ\mathsf{true}$

This again results in the following outgoing edges per vertex:

Vertex:	Outgoing (edge, vertex) pairs:
(4)	$\{(p, (4))\}$
(5)	$\{(q, (5))\}$
(2)	$\{(p \vee \neg q, (3))\}$

Vertex (3) is the **true** vertex. Expanding it will result in a single self-looping edge labelled with **true**. The complete model graph is drawn in figure 6.12. The annotations of the vertices will become clear in the sequel.

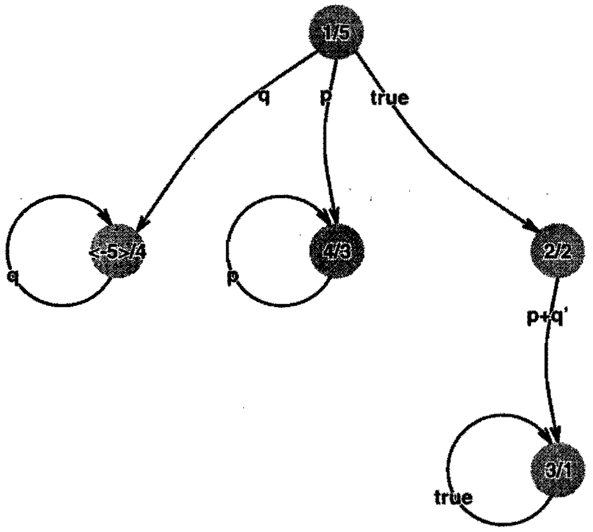


Figure 6.12. Model graph for example formula.

□ example 6.3

Once the model graph is created, an infinite path still has to be found starting at the initial vertex and satisfying all eventualities encountered in the vertices comprising the path. Instead of searching for a path and checking fulfillment of eventualities *a posteriori,* these actions may be combined with the actual model graph construction. Constructing a path entails a depth-first graph construction process; by appropriately marking the vertices on the current path, cycles are easily discovered. The depth-first approach will naturally construct a spanning tree in the graph. A strongly connected component in the graph can be shown to consist of vertices that form a subtree of this spanning tree [Tarja72]. Hence it makes sense to refer to a vertex as being the root of a strongly connected component. Note that the notion of a root only makes sense in the context of a given spanning tree. The following observations will prove helpful in determining a satisfying path in the model graph. They first occurred in [Janss90].

Lemma 6.1 If a vertex in a model graph is satisfiable, then so are all its predecessor vertices.

Proof: By definition of the disjunctive normal form for LTL formulas, a vertex with formula label f is a predecessor of a vertex labelled g, if the DNF of f contains the conjunct $p \wedge \circ g$ for some satisfiable propositional formula p (the edge label). Since we assume g to be satisfiable, $\circ g$ also is satisfiable, hence the conjunct $p \wedge \circ g$ is, and therefore f must be satisfiable.
□ lemma 6.1

Corollary 6.1 If a vertex has a successor vertex that is satisfiable, then the vertex itself is also satisfiable. Equivalently, if no successor is satisfiable then the vertex is not satisfiable.

Proof: Quite obvious restatement of lemma 6.1 in terms of successors.
□ corollary 6.1

Lemma 6.2 Either all vertices on a cycle are satisfiable or none of them is.

Proof: Immediate from lemma 6.1.
□ lemma 6.2

Corollary 6.2 Either all vertices of a strongly connected component (SCC) are satisfiable (we then speak of a satisfiable SCC) or none is.

Proof: Immediate from the definition of an SCC and lemma 6.2.
□ corollary 6.2

Lemma 6.3 If an eventuality is present in some vertex of a cycle and it is not fulfilled by the cycle, it must necessarily reappear in every vertex of the cycle. If, on the other hand, there is some vertex of the cycle where the particular eventuality is absent, we must conclude that the eventuality is fulfilled by the cycle.

Proof: We first examine how eventualities propagate in the model graph. Assume a single eventuality $\diamond g$ to be present in some vertex (the case of multiple eventualities and eventualities caused by strong until operators can be treated analogously). In general, the formula f labelling this vertex is a conjunct like $h \wedge \diamond g$,

where h stands for a conjunction of propositional and elementary formulas. A first step in rewriting formula f into DNF causes the eventuality to be split into its finite and inductive parts that will end up in different conjuncts: $(h \wedge g) \vee (h \wedge \circ \diamond g)$. We notice that in the path continuation via the finite term (g) the eventuality disappears, and that in the path continuation via the inductive term ($\circ \diamond g$) the eventuality reappears as part of the label of a successor vertex. These continuations are of course only valid when the respective propositional parts (the edge labels) are satisfiable, which is assumed to be the case in our model graphs. When the vertex is part of a cycle, we therefore conclude that its eventuality is fulfilled if and only if there exists some other vertex of the cycle in which the particular eventuality is absent, because absence of the eventuality implies that apparently the alternative finite term continuation was indeed taken.
□ lemma 6.3

Theorem 6.1 An SCC in the model graph is satisfiable if and only if

1. another satisfiable SCC can be reached from it, or
2. the root of the SCC has no eventualities and the SCC is not trivial (an SCC is called trivial when it consists of a single vertex without a self-looping edge), or
3. the SCC is non-trivial and the root does contain eventualities but they are all fulfilled within that SCC.

Proof: (1) If another satisfiable SCC can be reached we can apply lemma 6.1 repeatedly to reason backwards along a path to the satisfiable SCC. Note that in case of a trivial SCC either case (1) applies or it doesn't, and then it will be announced unsatisfiable by application of corollary 6.1.

For cases (2) and (3) it suffices to only consider non-trivial SCCs that have no outgoing paths to satisfiable SCCs. These SCCs will all have at least one infinite path that is fully contained within the SCC and therefore their satisfiability solely depends on the fulfillment of any eventualities on such a path. From lemma 6.3 we learn that it suffices to consider root vertex eventualities only, since any eventuality present in some other vertex of the SCC which is not fulfilled within that SCC will also be present in the root. If there are no eventualities present in the root, as stated in case (2), we can directly conclude that the SCC is satisfiable.

Suppose that indeed all eventualities present in the root are fulfilled within the component. Then the root is satisfiable and from corollary 6.2 we learn that all other vertices of the SCC are satisfiable as well. Contrariwise, if the eventualities in the root are not fulfilled, a satisfying path emanating from the root does not exist and we must conclude that the root is unsatisfiable, hence the whole SCC is unsatisfiable. This proves case (3).
□ theorem 6.1

The above theorem tells us that discovering SCCs in the graph is very fruitful. We use a slightly modified version of the well-known Tarjan algorithm [Tarja72]. The advantages are that with our algorithm we just keep one number, the so-called lowlink value, per vertex instead of two as in the Tarjan algorithm. Also we have simplified the lowlink update part (less cases need be considered). It turns out that these modifications have been known to others for some time, for instance Mark P. Jones uses exactly the same approach in his implementation of the functional programming language Gofer. I wasn't able to find the modified algorithm in any textbook or publication. Here is the (pseudo-)C rendition of our SCC algorithm that serves as a skeleton for the LTL satisfiability routine to be presented next.

```
/* Preconditions to top-level call: global dfsnum = 0; lowlink(u) = 0 for all u; vertex stack is empty. */
void SCCs(Vertex u)
{
  lowlink_orig:=lowlink(u):= ++dfsnum;
  push(u);

  foreach_outedge (e, u) {
    v:=dst(e); /* v is destination of edge e. */

    if (!lowlink(v)) SCCs(v);

    if (stacked(v) && lowlink(v) < lowlink(u))
      lowlink(u):=lowlink(v);
  }
  if (lowlink(u) = lowlink_orig) /* u is root of new component */
    /* New component consists of the popped vertices. */
    while (pop() != u);
}
```

Algorithm 6.9. Strongly connected components.

To turn algorithm 6.9 into an LTL satisfiability algorithm three modifications are necessary:

1. We rename the routine from `SCCs` to `sat` and introduce a boolean return value that is true in case a model is found, and false otherwise. From part 1 of theorem 6.1 we learn that when the recursive call returns true, the caller may also directly return true.

2. The for-loop that iterates over every successor `v` of `u` must be changed to iterate over every (edge, vertex) pair derived from the disjunctive normal form of the label for `u`. The latter is calculated by a routine named `dnf`.

3. Once a new SCC is discovered we still need to check parts 2 and 3 of theorem 6.1. This is done by `SCC_sat` which gets the root of the component as argument.

Applying these modifications yields the following algorithm 6.10. The first number annotated with each vertex in figure 6.12 correspond to the depth-first search number. The presence of an eventuality is indicated by putting the dfs-number in angular brackets. We see that vertex (5) indeed contains an eventuality (in fact caused by the $\Diamond\neg$q elementary subformula). The second number is the scc number which clearly is assigned in reverse topological fashion. The model graph in figure 6.12 has 5 SCCs.

```
Bool sat(Vertex u)
{
  lowlink_orig:=lowlink(u):= ++dfsnum;
  push(u);

  foreach (e,v) pair in dnf(u) {
    add edge e=(u,v) to model graph;

    if (!lowlink(v))
      if (sat(v)) return true;

    if (stacked(v) && lowlink(v) < lowlink(u))
      lowlink(u):=lowlink(v);
  }
  if (lowlink(u) = lowlink_orig) { /* u is root of new component */
    if (SCC_sat(u)) return true;
    while (pop() != u);
  }
  return false;
}
```

Algorithm 6.10. SCC modified for LTL satisfiability.

An algorithm for the conversion to disjunctive normal form has already been presented in section 3.4. So, it remains to explain how the function SCC_sat can be implemented. Checking for the second case of theorem 6.1 is obvious: see whether the root vertex has no eventualities and the component is not a single vertex without self-loop. The third case is trickier: there are eventualities among the elementary subformulas in the conjunct labelling the root vertex and we now must show that they are all fulfilled within the component. To ease this test we introduce markings on the edges. Whenever during DNF conversion a $\Diamond$ or U subformula is expanded the (edge, vertex) pair that is created for the finite term of the expansion will be flagged with the $\Diamond$ or U subformula. With this provision, we can check for fulfillment by collecting the markings of all edges belonging to the SCC and comparing this set with the eventualities present in the root vertex.

The eventuality $\Diamond\neg$q in the model graph of figure 6.12 is not fulfilled. This is indicated by the negative dfs-number in angular brackets. In this case we have a singleton SCC with a self-loop. During expansion of the vertex's formula

```
Bool eventualities_fulfilled(Vertex u)
{
  if (!eventualities_present(u)) return true;

  /* Collect all markings on edges of this SCC: */
  markings:=∅;
  for (v∈SCC of which u is root)
    foreach_outedge (e, v)
      if (dst(e)∈SCC of which u is root)
        markings:=markings∪markings(e);

  /* Now check for fulfillment of <> and U one by one: */
  foreach (f | f is <> or U subformula of u)
    if (f∉markings)
      return false;
  return true;
}

Bool SCC_sat(Vertex u)
{
  if (SCC_size(u) = 1) {
    /* Singleton SCC. */
    foreach_outedge (e, u)
      if (dst(e) = u)
        /* Self-looping edge present. */
        return eventualities_fulfilled(u);
    /* No outgoing edges or no self-loop. */
    return false;
  }
  /* Not a trivial SCC. */
  return eventualities_fulfilled(u);
}
```

Algorithm 6.11. Checking an SCC for eventualities.

$\Box q \wedge \Diamond\neg q$ the finite term disappears so no edge in the SCC will be marked.

6.8 Specification of finite state machines in LTL

The Finite State Machine (FSM) has since long been appreciated as a convenient model for the description of the behaviour of control hardware. It has become practice for many designers to specify control-dominated [WolfW90] logic designs using state diagrams or flow charts. Here we show how Finite State Machines can be described by LTL formulas. Many questions concerning the behaviour of the FSM may also be stated within the same formalism. We do not here consider the mathematical aspects of the relation between languages, automata and logics [Emers90]. We set a more modest goal of indicating the required transformations from a design automation point of view.

6.8.1 Notational preliminaries

A Finite Automaton is a quintuple ($Q, \Sigma, \delta, q_0, F$) with Q a finite set of states, Σ the input alphabet, δ the transition mapping, q_0 a start state and F the set of accepting states. The automaton defines a language of strings of input symbols. In terms of a logic circuit, we usually let the input symbols correspond to data bits on a number of input lines and the states correspond to the values contained in the flip-flops. In hardware applications it is the custom to introduce two derived machine concepts, known as the Moore and Mealy type machines. For clarity, we will classify the various machines types as shown in figure 6.13 below.

Without output:	With output:
Fully specified	Fully specified
deterministic	deterministic (Moore/Mealy)
non-deterministic	non-deterministic?
Incompletely specified	Incompletely specified
deterministic	deterministic (Moore/Mealy)
non-deterministic	non-deterministic?

Figure 6.13. Finite State Machine Classification.

We will call a machine incompletely specified if the δ function is not fully defined over its domain of states and symbols. Also we then allow don't care outputs in Moore and Mealy machines. There seems to be no practical sense in defining non-deterministic machines with output, hence the question mark. The basic idea to define a FSM in temporal logic is to associate the states of the machine with the states in the model, and let a transition coincide with a step in time.

We will use `lineprinter` font to denote temporal formulas in the syntax acceptable for our satisfiability checker program. Names for states and symbols of a FSM will be written in *italic*, using subscripts when appropriate. We prefer to leave the `&` (and) operator in LTL implicit. An operator applied to a set of operands is to be understood as the reduction of the operator over the operands, e.g. $\bigvee\{v_i\} = v_1 \vee v_2 \vee \cdots \vee v_n$.

6.8.2 FSM to LTL transformation

Let us start with the simple case of a fully specified, deterministic finite state machine without output. To describe such a machine in LTL we introduce a propositional variable for each input symbol and one for each state (Step 1). Of course, only exactly one input symbol may be offered at a time, but this has to be explicitly stated in our logic (Step 1′). Also, at any time, exactly one state can be the current state. Our interpretation for the variables is that when one is assigned true that symbol/state is the machine's current input symbol/state. For states this may be compared with a one-hot encoding scheme. In LTL, the mutual exclusion of variables v0, v1, v2, ..., vn can be expressed by:

```
[]( v1  v2' v3' ... vn'
  V v1' v2  v3' ... vn'
  ...
  V v1' v2' v3' ... vn )
```

Our δ mapping in this case is a total function δ : $Q \times \Sigma \to Q$. We write a clause for each state/symbol pair: (Qi Ik -> @ <δ (q_i, i_k)>) , where Qi is the LTL variable corresponding to the state q_i and Ik is the variable associated with the input symbol i_k. <δ (q_i, i_k)> stands for the LTL variable associated with the δ function result when applied to the arguments q_i and i_k. All these clauses are AND-ed together and put within an always operator (Step 2). This is illustrated in the examples in the next section. Note that the representation of the δ function in this way is not complete; we are required to explicitly state that only exactly one state variable is true at any time. So again, we include a clause for the mutual exclusion of a set of variables (Step 2′). The initial state of the machine, i.e. the fact that the machine starts in state q_0 can be expressed by: q0 q1' q2' ... qn' (Step 3). If necessary, we can introduce an LTL variable, say Accept, to denote the fact that we are in a final state (Step 4):
[](Accept <-> $\bigvee$<F>).

To avoid having to explicitly specify the mutual exclusivity of the states we can use a predecessor approach in representing the δ function, in the sense that we define clauses like (@ Qi <-> $\bigvee_{i_k}$ ($\bigvee$ <δ^{-1} (q_i, i_k)> Ik)) where <δ^{-1} (q_i, i_k)> denotes the set of LTL variables corresponding to the states that have transitions labelled i_k ending in the state q_i.

Incompleteness of a machine is resolved in the usual way namely by introducing a special state. Whenever δ is undefined in a state we add edges labelled with the missing symbols and directed towards that special state. The special state itself has an outgoing transition for each input symbol ending on itself (Step 5).

For a non-deterministic machine, possibly with ε-moves, we can proceed in much the same way as described above. Obviously, the only differences are expected to occur in the treatment of ε-moves and non-determinism. It turns out that the latter does not require any special treatment: we can define the δ function by the same predecessor approach as sketched above. For ε-moves we distinguish two cases:

1. A state q_i has only incoming ε transitions. Then we add a clause of the form (Qi <-> $\bigvee$<δ^{-1} (q_i, ε)>) . (Note the absence of the @ operator!)
2. A state q_i has except for incoming ε transitions also other incoming transitions. We now add two clauses, one to capture the labelled transitions in the usual way and a second of the form (Qi <- $\bigvee$<δ^{-1} (q_i, ε)>) .

As a last case, let us now consider an incompletely specified Mealy type machine. This means that the output function depends both on the current state and the current input. Steps 1, 2, 3 ,4 and 5 are the same as for the deterministic machine. We merely need to add an extra Step 6 that takes care of defining the outputs: add a clause for each binary output signal stating for what state/input-conditions it is true.

Example 6.4 A simple 4-state example is the Lion Cage Machine [Breid89]. Note that the `I1=1, I2=0` transition for state bada is not specified. Also 2 transitions have output don't cares (see figure 6.14).

```
/* State transition table (not fully specified): */
[](
(@start <-> start (i1' i2' V i1 i2' V i1 i2) V ett i1 i2)
(@ett <-> start i1' i2 V ett (i1' i2' V i1' i2)
       V  nasta i1' i2')
(@nasta <-> ett i1 i2' V nasta (i1 i2' V i1 i2) V bada i1 i2)
(@bada  <-> nasta i1' i2 V bada (i1' i2' V i1' i2)))

/* Output function (not fully specified): */
[] (VARNING' <- start (i1' i2' V i1 i2' V i1 i2))

[] (VARNING  <- ett (i1' i2' V i1' i2 V i1 i2')
               V nasta
               V bada (i1' i2' V i1' i2 V i1 i2))

/* Initial state: */
start ett' nasta' bada'

/* Input restriction: */
/* not ever in bada and seeing I1=1, I2=0 at input: */
[]~(bada i1 i2')
```

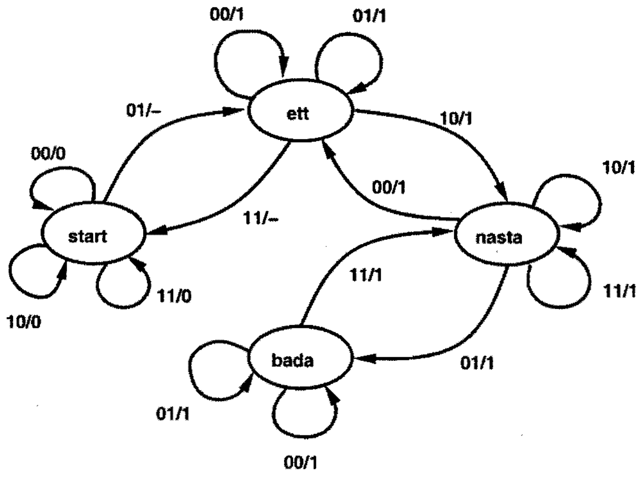


Figure 6.14. Lion Cage State Diagram.

□ example 6.4

The next example is a state machine that recognizes Algol-60 defined numbers and is taken from [Backh80]. We use LTL to prove that a non-deterministic version is equivalent to a minimalized deterministic one. Our experiment pointed out an error in the original diagram of the minimal state machine: one of the states was erroneously marked non-final.

Example 6.5 - ALGOL-60 numbers

```
/* State the exclusive occurrence of an input symbol: */
[](  plus  minus' period' digit' E'
   V plus' minus  period' digit' E'
   V plus' minus' period  digit' E'
   V plus' minus' period' digit  E'
   V plus' minus' period' digit' E )

/* Non-deterministic machine with eps-moves: */
[](
(@q1  <-> false)                 /* no incoming transitions */
(@q2  <-> q1 (plus V minus))
( q2  <-  q1)                    /* eps-move */
( q3  <-> q2)                    /* eps-move */
(@q4  <-> (q3 V q4) digit)
( q5  <-> q2)                    /* eps-move */
(@q6  <-> q5 period)
(@q7  <-> (q6 V q7) digit)
( q8  <-> q2)                    /* eps-move */
(@q9  <-> (q8 V q9) digit)
(@q10 <-> q9 period)
(@q11 <-> (q10 V q11) digit)
( q12 <-> q2 V q4 V q7 V q11)    /* eps-move */
(@q13 <-> q12 E)
(@q14 <-> q13 (plus V minus))
( q14 <-  q13)                   /* eps-move */
(@q15 <-> (q14 V q15) digit)
( q16 <-> q4 V q7 V q11 V q15)   /* eps-move */

/* Make completely specified by introducing err state: */
(@err <-> q2 (plus V minus)     V  q6 digit'
       V  q9 (period V digit)'  V  q10 digit'
       V  q13 (period V E)      V  q16 digit'
       V  err)
)

/* Initial state: */
q1 q2 q3 q4' q5 q6' q7' q8 q9' q10'
q11' q12 q13' q14' q15' q16' err'

/* Final states: */
[](Accept1 <-> q16)
```

Figure 6.15. Non-deterministic machine specified in LTL.

```
/* Deterministic minimal machine: */
[](
(@q1_          <-> false)
(@q2_3_5_8_12 <-> q1_ (plus V minus))
(@q4_9_12_16  <-> (q1_ V q2_3_5_8_12 V q4_9_12_16) digit)
(@q13_14      <-> (  q1_ V q2_3_5_8_12 V q4_9_12_16
                   V q7_11_12_16) E)
(@q6_10       <-> (q1_ V q2_3_5_8_12 V q4_9_12_16) period)
(@q7_11_12_16 <-> (q6_10 V q7_11_12_16) digit)
(@q14_        <-> q13_14 (plus V minus))
(@q15_16      <-> (q13_14 V q14_ V q15_16) digit)
/* Make completely specified: */
(@error <-> (q2_3_5_8_12 V q4_9_12_16) (plus V minus)
         V  q13_14 (period V E)
         V  (q6_10 V q14_ V q15_16) digit'
         V  q7_11_12_16 (digit V E)'
         V  error)
)
/* Initial state: */
q1_ q2_3_5_8_12' q4_9_12_16' q13_14' q6_10' q7_11_12_16'
q14_' q15_16' error'
/* Final states: */
[](Accept2 <-> q15_16 V q4_9_12_16 V q7_11_12_16)
->
[](Accept1 <-> Accept2).
```

Figure 6.16. Deterministic minimal machine in LTL.

□ example 6.5

Chapter 7

μ-Calculus

7.1 Introduction

This chapter will present a formal system called propositional μ-calculus that is powerful enough to encapsulate the two temporal logics that we have studied before: both LTL and CTL formulas can be recast into formulas of the μ-calculus. The purpose and main goal of this chapter is to show how a decision procedure for μ-calculus formulas over the boolean domain can be derived from a formal specification of the calculus' syntax and semantics. We will strive for the implementation of an efficient μ-calculus program. The interesting part is that all we need is propositional logic and a least fixed-point operator. Again BDDs will be used as the main data structure for representation of the boolean relations and functions involved.

The μ-calculus may be characterized as a formal system for manipulating predicates over a certain domain. Its first applications were oriented towards program proving. Our interest is inspired by its use as a specification language for sequential circuits and the analysis of properties thereof. Like the familiar predicate calculus (or first-order logic) [Galli87], μ-calculus has constructs for expressing function application, quantification, and besides domain variables there will also be predicate symbols. New constructs in the μ-calculus are abstraction and fixed-point terms. Abstraction should be familiar from λ-calculus [Baren84, Peyto87] and we have already used λ-abstraction as a means to define functions in previous chapters.

In μ-calculus we distinguish two main syntactic categories: formulas and terms. Intuitively, a formula asserts some relation to hold among the individual variables appearing in the formula such that given values for those variables the formula may be evaluated to a boolean value. A term may be interpreted as a set of tuples of values from the domain, in other words it represents a relation.

Example 7.1 Before going into the precise definition of the syntax and semantics of the μ-calculus, let us first look at a simple example. Let our universe of discourse, i.e. the domain, be the set of vertices V of a directed graph $G(V, E)$. The edges of the graph define a binary relation $E \subseteq V \times V$ over the set of vertices: two vertices u and v are related iff there exists an edge from u to v. The existence of a path (of non-zero length) from a vertex u to a vertex v can also be seen as a relation (in fact it is the transitive closure of the edge relation); in μ-calculus this relation can be expressed by:

$$\mu Z.\lambda u, v. E(u,v) \vee \exists w. Z(u,w) \wedge Z(w,v)$$

In this example, both a least fixed-point construct (μZ) and abstraction ($\lambda u, v$) are present. The above term states that there is a path from u to v when either there is an edge from u to v or there exists a vertex w such that there is a path from u to w and a path from w to v. Note the recursive character of the definition of a path. The 'solution' of the term is the relation that, when applied to two vertices, evaluates to true when indeed a path exists, or to false otherwise. Let us name the above term by the symbol P for path relation. Then the fixed-point construct explicitly denotes the solution of the equation:

$$P = \lambda u, v. E(u,v) \vee \exists w. P(u,w) \wedge P(w,v)$$

Application of both sides to the arguments (u, v) and using η-conversion yields:

$$P(u,v) = E(u,v) \vee \exists w. P(u,w) \wedge P(w,v)$$

When the binary relation E is represented as a boolean matrix, we easily derive Warshall's algorithm [Warsh62] for the computation of the transitive closure. The same notation μZ for the solution of a recursive equation is also used in CSP [Hoare85].
□ example 7.1

The rest of this chapter is organized as follows: first the syntax and semantics of the general μ-calculus are defined; then we restrict the definitions to the boolean domain. For this special case an extended language will be defined. We will give its concrete syntax in Backus-Naur Form; next a denotational semantics is defined, from which the satisfiability checking algorithm is derived through a reinterpretation in propositional logic together with a least fixed-point operator.

7.2 Syntax

The alphabet of μ-calculus consists of the following sets of symbols:

— logical constants: 0 (false), 1 (true),

— logical connectives: $\neg$ (not), $\vee$ (or), $\exists$ (existential quantification), = (equality on values of the domain D),

— variables: $U = \{z_0, z_1, z_2, \cdots\}$, denoting values of the domain D,

— punctuation symbols: (,), , , ., and the symbols μ and λ,

— predicate symbols: $PS = \{P_0, P_1, P_2, \cdots\}$, and a rank function $r : PS \longrightarrow N$ assigning to each predicate symbol a rank or arity. For instance, a predicate symbol P of rank 2 has signature $D \times D \longrightarrow B$. Predicate symbols will also be called relational variables.

An inductive definition of formulas and terms can now be given [Burch91]. No other constructs but the following 4 generate well-formed formulas:

1. The logical constants 0 and 1 are formulas,
2. If z_1 and z_2 are variables then $(z_1 = z_2)$ is a formula,
3. If f and g are formulas and z a variable, then $\neg f$, $f \vee g$, and $\exists z.(f)$ are formulas,
4. If $z_1, z_2, \cdots, z_n$ are variables and R an n-ary term, then the application $R(z_1, z_2, \cdots, z_n)$ is a formula.

The following 3 constructs generate all well-formed n-ary terms:

1. Any n-ary predicate symbol $X \in PS$ is an n-ary term,
2. If $z_1, z_2, \cdots, z_n$ are distinct variables and f is a formula, then the abstraction $\lambda z_1, z_2, \cdots, z_n.(f)$ is an n-ary term,
3. If R is a term with arity n and X is a predicate symbol, then the least fixed-point $\mu X.(R)$ is also a term with arity n provided that R is monotone non-decreasing in X. (This is a necessary condition to ensure the existence of a unique fixed-point; in practice, we will assume the stronger and therefore sufficient condition that the fixed-point term is positive, i.e., all free occurrences of X in R fall under an even number of negations. This is easily checked at the syntactic level.)

Perhaps a few words about free and bound variable occurrences are appropriate here. These notions are directly borrowed from λ-calculus, see e.g. [Peyto87], and are closely related to the notion of variable scope in many modern programming languages. Consider the formula $z_1 \vee z_2$. The occurrences of z_1 and z_2 in this formula are said to be free. Whereas in the term $\lambda z_1.(z_1 \vee z_2)$, the abstraction binds the free occurrence of z_1 but z_2 remains free. In the above syntax rules we see that variables may be bound by existential quantification and by λ-abstractions. Predicate symbols can only be bound by the least fixed-point construct. We will assume the conventional definitions for free and bound entities as listed in table 7.1.

z_0 in formula	occurs free?	occurs bound?
$z_1 = z_2$	No	No
$z_0 = z_2$	Yes	No
$\neg f$	If z_0 occurs free in f	If z_0 occurs bound in f
$f \vee g$	If z_0 occurs free in f or in g	If z_0 occurs bound in f or in g
$\exists z_0.(f)$	No	If z_0 occurs free in f
$\exists z_1.(f)$	If z_0 occurs free in f	If z_0 occurs bound in f
z_0 in term	occurs free?	occurs bound?
$\lambda z_0.(f)$	No	If z_0 occurs free in f
$\lambda z_1.(f)$	If z_0 occurs free in f	If z_0 occurs bound in f
P_0 in formula	occurs free?	occurs bound?
$P_1(z_1, z_2, \cdots)$	No	No
$P_0(z_1, z_2, \cdots)$	Yes	No
P_0 in term	occurs free?	occurs bound?
$\mu P_0.(R)$	No	If P_0 occurs free in R
$\mu P_1.(R)$	If P_0 occurs free in R	If P_0 occurs bound in R

Table 7.1. Free and bound occurrences of variables and predicate symbols.

7.3 Semantics

The meaning of formulas and terms is defined with respect to a structure $M = (D, I_P, I_D)$ where D is a non-empty set called the domain of the structure, I_P is the predicate symbol interpretation function, and I_D is the variable interpretation function. The predicate symbol interpretation function is a mapping from predicate symbols to n-ary predicates of signature $D^n \longrightarrow B$. The arity n is the rank defined for the predicate symbol. Each predicate may thus be regarded as the characteristic function of an n-place relation over D. In the sequel we will make no distinction between the characteristic function of a relation (or set) and the relation (or set) itself seen as a set of tuples (or elements). The variable interpretation function maps variables to domain values, i.e., elements of D.

The semantics of the formulas and terms of the μ-calculus is captured by interpretation functions $\mathcal{D}_f$ and $\mathcal{D}_t$ that interpret a formula, respectively a term, with respect to a given structure $M = (D, I_P, I_D)$. Formally, the signatures of these interpretation functions are:

$$\mathcal{D}_f : \text{FORMULA} \times \mathcal{I}_P \times \mathcal{I}_D \longrightarrow B$$

$$\mathcal{D}_t : \text{TERM} \times \mathcal{I}_P \times \mathcal{I}_D \longrightarrow 2^{D^n}$$

where $\mathcal{I}_P$ is the set of all possible predicate interpretation functions I_P, $\mathcal{I}_D$ is the set of all variable interpretation functions I_D, and 2^{D^n} is the set of all possible n-place relations over D. The interpretations of formulas and terms will now be inductively defined along the rules of the syntax, assuming given interpretations for the variables ($\mathbf{I}_D$) and predicate symbols ($\mathbf{I}_P$). To stress that the first argument of $\mathcal{D}_f$ and $\mathcal{D}_t$ are syntactic constructs they are written enclosed in double quotes.

$$\begin{array}{ll}
\mathcal{D}_f(\text{"0"}, \mathbf{I}_P, \mathbf{I}_D) & = 0 \\
\mathcal{D}_f(\text{"1"}, \mathbf{I}_P, \mathbf{I}_D) & = 1 \\
\mathcal{D}_f(\text{"}(z_1 = z_2)\text{"}, \mathbf{I}_P, \mathbf{I}_D) & = (\mathbf{I}_D(z_1) = \mathbf{I}_D(z_2)) \\
\mathcal{D}_f(\text{"}\neg f\text{"}, \mathbf{I}_P, \mathbf{I}_D) & = \neg \mathcal{D}_f(\text{"f"}, \mathbf{I}_P, \mathbf{I}_D) \\
\mathcal{D}_f(\text{"}f \vee g\text{"}, \mathbf{I}_P, \mathbf{I}_D) & = \mathcal{D}_f(\text{"f"}, \mathbf{I}_P, \mathbf{I}_D) \vee \mathcal{D}_f(\text{"g"}, \mathbf{I}_P, \mathbf{I}_D) \\
\mathcal{D}_f(\text{"}\exists z. f\text{"}, \mathbf{I}_P, \mathbf{I}_D) & = \exists_{e \in D}. \mathcal{D}_f(\text{"f"}, \mathbf{I}_P, \mathbf{I}_D[z := e]) \\
\mathcal{D}_f(\text{"}R(z_1, \cdots, z_n)\text{"}, \mathbf{I}_P, \mathbf{I}_D) & = (\mathbf{I}_D(z_1), \cdots, \mathbf{I}_D(z_n)) \in \mathcal{D}_t(\text{"R"}, \mathbf{I}_P, \mathbf{I}_D)
\end{array}$$

$$\begin{array}{ll}
\mathcal{D}_t(\text{"X"}, \mathbf{I}_P, \mathbf{I}_D) & = \mathbf{I}_P(X) \\
\mathcal{D}_t(\text{"}\lambda z_1, \cdots, z_n. f\text{"}, \mathbf{I}_P, \mathbf{I}_D) & = \{ (e_1, \cdots, e_n) \mid \mathcal{D}_f(\text{"f"}, \mathbf{I}_P, \mathbf{I}_D[z_1 := e_1, \cdots, z_n := e_n]) \} \\
\mathcal{D}_t(\text{"}\mu X. R\text{"}, \mathbf{I}_P, \mathbf{I}_D) & = \text{lfp } \lambda Y \in 2^{D^n}. \mathcal{D}_t(\text{"R"}, \mathbf{I}_P[X := Y], \mathbf{I}_D)
\end{array}$$

Note 1: lfp means least fixed point. $\mu X. R$ denotes the set $Q \in 2^{D^n}$ such that:

1. $Q = \mathcal{D}_t(\text{"R"}, \mathbf{I}_P[X := Q], \mathbf{I}_D)$, i.e., Q is a fixed-point, and
2. $\forall_{P \subset Q}. P \neq \mathcal{D}_t(\text{"R"}, \mathbf{I}_P[X := P], \mathbf{I}_D)$, i.e., Q is a least fixed-point.

The requirement that R is monotone non-decreasing in X ensures the existence and uniqueness of the fixed-point.

Note 2: We use $\mathbf{I}[z_1 := e_1, \cdots, z_n := e_n]$ to denote the alteration (or update) of interpretation $\mathbf{I}$ to a new function that evaluates z_1 to e_1, z_2 to e_2, et cetera, leaving evaluations of any other variables intact. The alterations or 'assignments' are understood to be done simultaneously. The e's are understood to be domain values, e.g. $(e_1, e_2) \in D^2$.

Note 3: For readers familiar with denotational semantics (e.g. of a programming language), we like to point out that our notation of the interpretation functions $\mathcal{D}_f$ and $\mathcal{D}_t$ corresponds to the notation using double square brackets, as in:

$$\text{Eval}[[\neg f]]_{\mathbf{I}_P, \mathbf{I}_D} = \neg \text{Eval}[[f]]_{\mathbf{I}_P, \mathbf{I}_D}$$

Example 7.2 Again we investigate the transitive closure of the edge relation in a graph expressed by the μ-calculus term:

$$\mu Z. \lambda u, v. E(u, v) \vee \exists w. Z(u, w) \wedge Z(w, v)$$

To actually calculate its outcome, we need to fix a graph; or better, the graph establishes the model in which this term is to be interpreted. We identify the vertices with non-negative integers. Let us assume to have a graph of 4 vertices, then $D = \{0, 1, 2, 3\}$. The edge relation is denoted by the predicate symbol E and assume that its interpretation is:

$$I_P(E) = \{ (0,1), (0,2), (1,2), (2,3), (3,3)\}$$

We assume the variable interpretation to be empty since there are no free variables. Now we calculate the least fixed point term that denotes all paths in the graph:

$$\begin{aligned}&\mathcal{D}_t(\text{"}\mu Z.\, \lambda u,v.\, E(u,v) \vee \exists w.\, Z(u,w) \wedge Z(w,v)\text{"}, I_P, I_D)\\ &= \text{lfp } \lambda Y \in 2^{D^n}.\, \mathcal{D}_t(\text{"}\lambda u,v.\, E(u,v) \vee \exists w.\, Z(u,w) \wedge Z(w,v)\text{"}, I_P[Z := Y], I_D)\end{aligned}$$

This requires the calculation of:

$$\begin{aligned}&\mathcal{D}_t(\text{"}\lambda u,v.\, E(u,v) \vee \exists w.\, Z(u,w) \wedge Z(w,v)\text{"}, I_P[Z := Y], I_D)\\ &= \{ (e_1,e_2) \mid \mathcal{D}_f(\text{"}E(u,v) \vee \exists w.\, Z(u,w) \wedge Z(w,v)\text{"},\\ &\qquad I_P[Z := Y], I_D[z_1 := e_1, z_2 := e_2])\}\end{aligned}$$

However, we cannot proceed because we need to know Y. As we will shortly see, the least fixed-point calculation can be done by means of an iteration that starts with $Y = \emptyset$. Substituting this value gives:

$$\begin{aligned}&\{ (e_1,e_2) \mid \mathcal{D}_f(\text{"}E(u,v) \vee \exists w.\, Z(u,w) \wedge Z(w,v)\text{"},\\ &\qquad I_P[Z := \emptyset], I_D[z_1 := e_1, z_2 := e_2])\}\end{aligned}$$

Skipping a few tedious interpretation steps, we find that the set of pairs $I_P(E)$ results after the first iteration step in the least fixed-point calculation. In a later example we will carry out such a calculation in more detail.
□ example 7.2

In the sequel we restrict the domain D to the set of truth values B. n-ary terms then stand for subsets of B^n, which can be represented by their characteristic function in BDD form using dummy variables $d_1, \cdots, d_n$ as place-holders in order to be able to correctly instantiate (apply) a term.

7.4 Boolean μ-calculus

We extend the syntax of the μ-calculus to allow the use of:

1. Boolean variables as formulas,
2. Literals: a boolean variable is a positive literal; a negative literal is the negation of a boolean variable x denoted by x',
3. Universal quantification: $\forall z_1, z_2, \cdots, z_k.\, f$,
4. Logical connectives $\wedge$ (and), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence, replaces the symbol $=$), and $\oplus$ (exclusive-or) in formulas,
5. Application to arguments that are formulas,
6. 0 and 1 as generic constant terms (of zero arity) denoting the empty relation and the complete relation respectively,

7. A greatest fixed-point construct $\nu X . R$,
8. Logical operations on terms, using the connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$, and $\oplus$,
9. Literal predicate symbols, using P' to denote the negation of P.

We now introduce a convenient concrete syntax for our boolean μ-calculus defined in Backus-Naur notation (figure 7.1).

```
Formula ::= Formula_1 | Quantified_Formula .

Quantified_Formula ::= ( 'E' | 'A' ) { BV / ',' }+ '.' Formula .

Formula_1 ::= Formula_2 { ( '->' | '<->' | 'xor' ) Formula_2 } .

Formula_2 ::= Formula_3 { '+' Formula_3 } .

Formula_3 ::= Formula_4 { '&' Formula_4 } .

Formula_4 ::= { '~' } Atomic_Formula .

Atomic_Formula ::= Primitive_Formula | '(' Formula ')' .

Primitive_Formula ::= '0' | '1' | B_Var [ '''' ] | Application .

Application ::= Atomic_Term (  Primitive_Formula
                             | '(' { Formula / ',' }+ ')' ) .

Term ::= Term_1 | Abstraction | Fixed_Point .

Abstraction ::= 'L' { BV / ',' }+ '.' Formula .

Fixed_Point ::= ( 'mu' | 'nu' ) RV '.' Term .

Term_1 ::= Term_2 { ( '->' | '<->' | 'xor' ) Term_2 } .

Term_2 ::= Term_3 { '+' Term_3 } .

Term_3 ::= Term_4 { '&' Term_4 } .

Term_4 ::= { '~' } Atomic_Term .

Atomic_Term ::= '0' | '1' | RV [ '''' ] | '[' Term ']' .
```

Figure 7.1. Boolean μ-calculus concrete syntax in BNF.

The new notation for the connectives and punctuation symbols is made clear in table 7.2. We assume that the set of boolean variables BV and predicate symbols or relational variables RV are disjoint. Apart from the overloading of the 0 and 1 tokens and taking the above assumption into account the grammar is LL(1) and therefore unambiguous [Backh80]. It is straightforward to convert this grammar into an equivalent one, i.e., a grammar that generates the same language, but uses less meta-symbols (see figure 7.2).

Math. notation:	mu notation:	Meaning:
$\exists z_1, z_2, \cdots, z_k.$	`E z1, z2, ..., zk .`	Existential quant.
$\forall z_1, z_2, \cdots, z_k.$	`A z1, z2, ..., zk .`	Universal quant.
$\rightarrow$	`->`	Implication
$\leftrightarrow$	`<->`	Equivalence
$\oplus$	`xor`	Exclusive-or
$\vee$	`+`	(Inclusive-)or
$\wedge$	`&`	And
$\neg$	`~`	Not
$\lambda z_1, z_2, \cdots, z_k.$	`L z1, z2, ..., zk .`	Abstraction
$\mu X.$	`mu X .`	Least fixed-point
$\nu X.$	`nu X .`	Greatest fixed-point

Table 7.2. Notational correspondence.

```
F   ::= F1 | QF .
QF  ::= 'E' BVL '.' F | 'A' BVL '.' F .
F1  ::= F2 | F1 '->' F2 | F1 '<->' F2 | F1 'xor' F2 .
F2  ::= F3 | F2 '+' F3 .
F3  ::= F4 | F3 '&' F4 .
F4  ::= AF | '~' F4 .
AF  ::= PF | '(' F ')' .
PF  ::= '0' | '1' | BV | BV '''' | Ap .
Ap  ::= AT PF | AT '(' FL ')' .
FL  ::= F | FL ',' F .
T   ::= T1 | Ab | FP .
Ab  ::= 'L' BVL '.' F .
BVL ::= BV | BVL ',' BV .
FP  ::= 'mu' RV '.' T | 'nu' RV '.' T .
T1  ::= T2 | T1 '->' T2 | T1 '<->' T2 | T1 'xor' T2 .
T2  ::= T3 | T2 '+' T3 .
T3  ::= T4 | T3 '&' T4 .
T4  ::= AT | '~' T4 .
AT  ::= '0' | '1' | RV | RV '''' | '[' T ']' .
```

Figure 7.2. Boolean μ-calculus concrete syntax in Restricted-BNF.

The shorter names that we use for the non-terminal symbols should be obvious. We will denote the set of strings generated by a non-terminal symbol by the name of that non-terminal symbol; thus `F` denotes the universe of all formulas.

Also, we use the lowercase name to denote an arbitrary element of such a set of strings: `f` is a formula in `F`. We developed a computer program called `mu` based on the latter syntax. (The new syntax of figure 7.2 is directly suitable as input to the parser generator tool `yacc`.)

To ease the definition of the semantics of formulas and terms, some operators and constructs are seen as abbreviations of more elaborate constructs. Table 7.3 informally indicates the intended abbreviations. For these abbreviations we can define a transformation to strings of a simpler grammar (as in figure 7.3).

Construct:	Abbreviates:
`E z1, z2, ..., zk . f`	`E z1 . E z2 . ... E zk . f`
`A z1, z2, ..., zk . f`	`~(E z1, z2, ..., zk . ~(f))`
`G -> H`	`~(G) + H`
`G <-> H`	`(G -> H) & (H -> G)`
`G xor H`	`~(G <-> H)`
`G & H`	`~(~(G) + ~(H))`
`s'`	`~(s)`
`nu X . t`	`~(mu X . ~(t[~X/X]))`

Table 7.3. Abbreviations. `f` stands for an arbitrary formula; the `zi` are arbitrary variables; `G` and `H` are either both formulas or both terms; `s` stands for an arbitrary variable or predicate symbol; `X` is a predicate symbol and `t` a term, and `t[~X/X]` denotes the term that results after substituting `~X` for all free occurrences of `X` in `t`. Note that the correspondence is of a recursive nature.

This simple grammar is then the basis for our semantics definition.

```
F ::= 'E' BV '.' '(' F ')'
    | '(' F '+' F ')'
    | '~' '(' F ')'
    | '0' | '1' | BV
    | '[' T ']' '(' FL ')' .

FL ::= F | FL ',' F .

T ::= 'L' BVL '.' '(' F ')'
    | 'mu' RV '.' '[' T ']'
    | '[' T '+' T ']'
    | '~' '[' T ']'
    | '0' | '1' | RV .

BVL ::= BV | BVL ',' BV .
```

Figure 7.3. Boolean μ-calculus core syntax.

We define the alphabet Γ of the boolean μ-calculus language of figure 7.1 by:

Γ = V∪RV∪{`E`, `A`, `.`, `->`, `<->`, `xor`, `+`, `&`, `~`, `(`, `)`, `0`, `1`, `'`, `,`, `L`, `mu`, `nu`, `[`, `]`}

The alphabet Σ of the language generated by the grammar of figure 7.3 is a proper subset of Γ. We define a transformation $\mathcal{T}$ as a (partial) function from Γ^* to Σ^* such that every well-formed formula/term of the first language is translated into a well-formed formula/term of the second language. Here follows an inductive definition of $\mathcal{T}$:

```
𝒯( E bv.f )          =E bv.(𝒯( f ))
𝒯( E bvl, bv.f )     =E bv.(𝒯( E bvl.f ))
𝒯( A bvl.f )         =~(𝒯( E bvl.~(f) ))
𝒯( f1  -> f2 )       =(~(𝒯( f1 )) + 𝒯( f2 ))
𝒯( f1 <-> f2 )       =𝒯( (f1 -> f2) & (f2 -> f1) )
𝒯( f1 xor f2 )       =~(𝒯( f1 <-> f2 ))
𝒯( f2 + f3 )         =(𝒯( f2 ) + 𝒯( f3 ))
𝒯( f3 & f4 )         =~((~(𝒯( f3 )) + ~(𝒯( f4 ))))
𝒯( ~f4 )             =~(𝒯( f4 ))
𝒯( 0 )               =0
𝒯( 1 )               =1
𝒯( bv )              =bv
𝒯( bv' )             =~(bv)
𝒯( (f) )             =𝒯( f )
𝒯( at 0 )            =[𝒯( at )] (0)
𝒯( at 1 )            =[𝒯( at )] (1)
𝒯( at bv )           =[𝒯( at )] (bv)
𝒯( at bv' )          =[𝒯( at )] (~(bv))
𝒯( at ap )           =[𝒯( at )] (𝒯( ap ))
𝒯( at ( f1 ) )       =[𝒯( at )] (𝒯( f1 ))
𝒯( f1, f )           =𝒯( f1 ), 𝒯( f )
------------------------------------------------------------
𝒯( L bvl.f )         =L bvl.(𝒯( f ))
𝒯( mu rv.t )         =mu rv.[𝒯( t )]
𝒯( nu rv.t )         =~[mu rv.[~[𝒯( t[~rv/rv] )]]]
𝒯( t1  -> t2 )       =[~[𝒯( t1 )] + 𝒯( t2 )]
𝒯( t1 <-> t2 )       =𝒯( [t1 -> t2] & [t2 -> t1] )
𝒯( t1 xor t2 )       =~[𝒯( t1 <-> t2 )]
𝒯( t2 + t3 )         =[𝒯( t2 ) + 𝒯( t3 )]
𝒯( t3 & t4 )         =~[[~[𝒯( t3 )] + ~[𝒯( t4 )]]]
𝒯( ~t4 )             =~[𝒯( t4 )]
𝒯( 0 )               =0
𝒯( 1 )               =1
𝒯( rv )              =rv
𝒯( rv' )             =~[rv]
𝒯( [t] )             =𝒯( t )
```

The transformation $\mathcal{T}$ may be regarded as a parser specification: it transforms a string to a fully parenthesized form, cf. syntax tree.

Example 7.3 Careful inspection of the various cases in the definition of $\mathcal{T}$ will indeed show that:

```
𝒯( A u.[mu P.L u,v.N(u,v)+(E w.P(u,w)&P(w,v))](u,u) )=
~(E u.(~([mu P.[L u,v.(([N](u,v)+
    E w.(~((~([P](u,w))+~([P](w,v))))))))]](u,u))));
```

Compare this μ-formula to the μ-term exhibited in the previous examples then it should be clear that the above formula, when interpreted w.r.t. to a graph with N as its edge relation, states that each vertex is on a cycle of length at least 1.
□ example 7.3

We define the meaning of a formula/term of the extended language to be the meaning of the transformed formula/term. For the latter, we again define two functions $\mathcal{D}_f$ and $\mathcal{D}_t$ that capture the semantics of formulas and terms.

This time we shall define the meaning of boolean μ-calculus formulas and terms to be formulas of a simple propositional logic. Our goal is to represent the formulas of the propositional logic by BDDs. Figure 7.4 presents the syntax of the propositional logics PL and PLD. They are identical except for the fact that the logic PLD has an extra set of propositional variables: the so-called dummy variables, $d_i \in$ Dum. Dummy variables are solely used in the meaning of terms; a μ-calculus formula will never have a meaning in which dummy variables occur.

```
PL  ::= ( PL ∨ PL )
      | ¬ ( PL )
      | 0 | 1 | BV .

PLD ::= ( PLD ∨ PLD )
      | ¬ ( PLD )
      | 0 | 1 | BV | Dum .
```

Figure 7.4. The PL and PLD Propositional Logics Syntax.

The semantics functions are of the following type:

$\mathcal{D}_f$: F $\longrightarrow$ PL

$\mathcal{D}_t$: T $\longrightarrow$ PLD

where, as by convention mentioned earlier, F stands for the set of all well-formed formulas of the μ-calculus core syntax and T stands for terms. The variables that play a role in the definition of the semantics functions are collected in the set V $\cup$ RV. The interpretation for these variables is defined by a function:

I : V $\cup$ RV $\longrightarrow$ PLD, with

$\mathbf{I}(\texttt{bv}) = \texttt{bv}$, for all $\texttt{bv} \in \texttt{V}$,
$\mathbf{I}(\texttt{rv})$ = some element $\texttt{pld} \in \texttt{PLD}$ or undefined denoted by $-$,
for all $\texttt{rv} \in \texttt{RV}$.

Thus the only factor that influences the meaning of a formula/term is the interpretation of the predicate symbols. Figure 7.5 gives the complete inductive definition of the semantics functions $\mathcal{D}_f$ and $\mathcal{D}_t$. However, one important aspect has not been discussed so far: the rank or arity of the terms. We have implicitly assumed that in applications `[p] (g1)` the rank of the term `p` and the number of argument formulas in the list `g1` match. Case (F.7) is therefore only valid when a certain condition for the rank of `p` holds. This condition then guarantees that formulas are properly interpreted, i.e., they do not depend on any dummy variables. It can be shown that the weakest condition that suffices requires that the rank of `p` is less than or equal to the number of argument formulas in the application. The rank of a term can be inductively defined according the core syntax for terms. Bottom cases are formed by abstractions, relational variables and the constant terms `0` and `1`.

	$\mathbf{I}[\texttt{v} := \texttt{pld}]$	$= \lambda x.$ if $x = \texttt{v}$ then $\texttt{pld}$ else $\mathbf{I}(x)$
(F.1)	$\mathcal{D}_f(\texttt{E bv.(g)}, \mathbf{I})$	$= (\mathcal{D}_f(\texttt{g}, \mathbf{I})[0/\texttt{bv}] \vee \mathcal{D}_f(\texttt{g}, \mathbf{I})[1/\texttt{bv}])$
(F.2)	$\mathcal{D}_f(\texttt{(g + h)}, \mathbf{I})$	$= (\mathcal{D}_f(\texttt{g}, \mathbf{I}) \vee \mathcal{D}_f(\texttt{h}, \mathbf{I}))$
(F.3)	$\mathcal{D}_f(\texttt{\~{}(g)}, \mathbf{I})$	$= \neg(\mathcal{D}_f(\texttt{g}, \mathbf{I}))$
(F.4)	$\mathcal{D}_f(\texttt{0}, \mathbf{I})$	$= 0$
(F.5)	$\mathcal{D}_f(\texttt{1}, \mathbf{I})$	$= 1$
(F.6)	$\mathcal{D}_f(\texttt{bv}, \mathbf{I})$	$= \mathbf{I}(\texttt{bv})$
(F.7)	$\mathcal{D}_f(\texttt{[p] (g1, ..., gn)}, \mathbf{I})$	$= \mathcal{D}_t(\texttt{p}, \mathbf{I})[\mathcal{D}_f(\texttt{g1}, \mathbf{I})/d_1, \cdots, \mathcal{D}_f(\texttt{gn}, \mathbf{I})/d_n]$
(T.1)	$\mathcal{D}_t(\texttt{L bv.(g)}, \mathbf{I})$	$= \mathcal{D}_f(\texttt{g}, \mathbf{I})[d_1/\texttt{bv}]$
(T.2)	$\mathcal{D}_t(\texttt{L bvl, bv.(g)}, \mathbf{I})$	$= \mathcal{D}_t(\texttt{L bvl.(g)}, \mathbf{I})[d_{\lvert\texttt{bvl}\rvert+1}/\texttt{bv}]$
(T.3)	$\mathcal{D}_t(\texttt{mu rv.[p]}, \mathbf{I})$	$= \mathrm{lfp}(\texttt{rv}, \texttt{p}, \mathbf{I}[\texttt{rv} := 0])$
(T.4)	$\mathcal{D}_t(\texttt{[p + q]}, \mathbf{I})$	$= (\mathcal{D}_t(\texttt{p}, \mathbf{I}) \vee \mathcal{D}_t(\texttt{q}, \mathbf{I}))$
(T.5)	$\mathcal{D}_t(\texttt{\~{}[p]}, \mathbf{I})$	$= \neg(\mathcal{D}_t(\texttt{p}, \mathbf{I}))$
(T.6)	$\mathcal{D}_t(\texttt{0}, \mathbf{I})$	$= 0$
(T.7)	$\mathcal{D}_t(\texttt{1}, \mathbf{I})$	$= 1$
(T.8)	$\mathcal{D}_t(\texttt{rv}, \mathbf{I})$	$= \mathbf{I}(\texttt{rv})$
	$\mathrm{lfp}(\texttt{rv}, \texttt{p}, \mathbf{I})$	= if $(\mathcal{D}_t(\texttt{p}, \mathbf{I}) = \mathbf{I}(\texttt{rv}))$ then $\mathbf{I}(\texttt{rv})$ else $\mathrm{lfp}(\texttt{rv}, \texttt{p}, \mathbf{I}[\texttt{rv} := \mathcal{D}_t(\texttt{p}, \mathbf{I})])$

Figure 7.5. Semantics of boolean μ-calculus. `g`, `g1, ..., gn`, `h` $\in$ `F`, `bv` $\in$ `V`, $d_i \in$ Dum, `p`, `q` $\in$ `T`, `rv` $\in$ `RV`, and $\lvert\texttt{bvl}\rvert$ denotes the number of variables in a comma separated list of boolean variables.

For each term we define a rank via the function:

$$\mathrm{rank} : \mathtt{T} \longrightarrow \mathrm{N},$$

such that

$$\begin{array}{ll}
\mathrm{rank}(\mathtt{L\ bvl.(g)}) = |\mathtt{bvl}| & \text{(number of variables in the list)} \\
\mathrm{rank}(\mathtt{mu\ rv.[p]}) = \mathrm{rank}(\mathtt{p[0/rv]}) & \\
\mathrm{rank}(\mathtt{[p\ +\ q]}) = \max(\mathrm{rank}(\mathtt{p}), \mathrm{rank}(\mathtt{q})) & \\
\mathrm{rank}(\mathtt{\tilde{}[p]}) = \mathrm{rank}(\mathtt{p}) & \\
\mathrm{rank}(\mathtt{0}) = 0 & \\
\mathrm{rank}(\mathtt{1}) = 0 & \\
\mathrm{rank}(\mathtt{rv}) = \mathrm{r}(\mathtt{rv}) &
\end{array}$$

Note that in the last case r refers to the rank function for the predicate symbols which forms part of the definition of the calculus. Predicate symbols introduced by a fixed-point are never questioned for their rank since the second case substitutes 0 for them.

Example 7.4 Again consider the formula `f`:
```
A u.[mu P.L u,v.N(u,v)+(E w.P(u,w)&P(w,v))](u,u)
```
Expressed in our core syntax this formula reads:
```
~(E u.(~([mu P.[L u,v.(([N](u,v)+
    E w.(~((~([P](u,w))+~([P](w,v)))))))]](u,u))));
```
We choose as an interpretation:
$\mathbf{I} = \{ (\mathtt{u},\mathtt{u}), (\mathtt{v},\mathtt{v}), (\mathtt{w},\mathtt{w}), (\mathtt{N},\mathrm{d}_2), (\mathtt{P}, -) \}$, where N has rank 2. In fact we are overspecifying the interpretation: since the variables u, v, and w are all bound, and so is the predicate symbol P, values for them will always be supplied during the interpretation process of the formula and hence initial values are not needed.

Even for this simple example the calculation by hand of $\mathcal{D}_f(\mathtt{f})$ is quite elaborate. The calculation proceeds as follows (for brevity, subformulas and subterms will be denoted by numbers in angular brackets instead of repeating them in full):

$$\begin{array}{lll}
\mathcal{D}_f(\mathtt{f},\mathbf{I}) & = \mathcal{D}_f(\mathtt{<0>},\mathbf{I}) & \\
\mathcal{D}_f(\mathtt{<0>},\mathbf{I}) & = \mathcal{D}_f(\mathtt{\tilde{}(<1>)},\mathbf{I}) & = \neg(\mathcal{D}_f(\mathtt{<1>},\mathbf{I})) \\
\mathcal{D}_f(\mathtt{<1>},\mathbf{I}) & = \mathcal{D}_f(\mathtt{E\ u.(<2>)},\mathbf{I}) & = (\mathcal{D}_f(\mathtt{<2>},\mathbf{I})[0/\mathtt{u}] \vee \mathcal{D}_f(\mathtt{<2>},\mathbf{I})[1/\mathtt{u}]) \\
\mathcal{D}_f(\mathtt{<2>},\mathbf{I}) & = \mathcal{D}_f(\mathtt{\tilde{}(<3>)},\mathbf{I}) & = \neg(\mathcal{D}_f(\mathtt{<3>},\mathbf{I})) \\
\mathcal{D}_f(\mathtt{<3>},\mathbf{I}) & = \mathcal{D}_f(\mathtt{[<4>](u,u)},\mathbf{I}) & = \mathcal{D}_t(\mathtt{<4>},\mathbf{I})[\mathcal{D}_f(\mathtt{u},\mathbf{I})/\mathrm{d}_1, \mathcal{D}_f(\mathtt{u},\mathbf{I})/\mathrm{d}_2] \\
\mathcal{D}_t(\mathtt{<4>},\mathbf{I}) & = \mathcal{D}_t(\mathtt{mu\ P.[<5>]},\mathbf{I}) & = \mathrm{lfp}(\mathtt{P},\mathtt{<5>},\mathbf{I}[\mathtt{P} := 0])
\end{array}$$

In order to calculate the fixed-point term we might have to repeatedly calculate:

$$\begin{array}{lll}
\mathcal{D}_t(\mathtt{<5>},\mathbf{I}) & = \mathcal{D}_t(\mathtt{L\ u,v.(<6>)},\mathbf{I}) & = \mathcal{D}_t(\mathtt{L\ u.(<6>)},\mathbf{I})[\mathrm{d}_2/\mathtt{v}] \\
\mathcal{D}_t(\mathtt{L\ u.(<6>)},\mathbf{I}) & = \mathcal{D}_f(\mathtt{<6>},\mathbf{I})[\mathrm{d}_1/\mathtt{u}] & \\
\mathcal{D}_f(\mathtt{<6>},\mathbf{I}) & = \mathcal{D}_f(\mathtt{(<7>\ +\ <8>)},\mathbf{I}) & = (\mathcal{D}_f(\mathtt{<7>},\mathbf{I}) \vee \mathcal{D}_f(\mathtt{<8>},\mathbf{I})) \\
\mathcal{D}_f(\mathtt{<7>},\mathbf{I}) & = \mathcal{D}_f(\mathtt{[N](u,v)},\mathbf{I}) & = \mathcal{D}_t(\mathtt{N},\mathbf{I})[\mathcal{D}_f(\mathtt{u},\mathbf{I})/\mathrm{d}_1, \mathcal{D}_f(\mathtt{v},\mathbf{I})/\mathrm{d}_2] \\
\mathcal{D}_t(\mathtt{N},\mathbf{I}) & = \mathbf{I}(\mathtt{N}) & = \mathrm{d}_2 \\
\mathcal{D}_f(\mathtt{<8>},\mathbf{I}) & = \mathcal{D}_f(\mathtt{E\ w.(<9>)},\mathbf{I}) & = (\mathcal{D}_f(\mathtt{<9>},\mathbf{I})[0/\mathtt{w}] \vee \mathcal{D}_f(\mathtt{<9>},\mathbf{I})[1/\mathtt{w}]) \\
\mathcal{D}_f(\mathtt{<9>},\mathbf{I}) & = \mathcal{D}_f(\mathtt{\tilde{}(<10>)},\mathbf{I}) & = \neg(\mathcal{D}_f(\mathtt{<10>},\mathbf{I}))
\end{array}$$

$$\begin{array}{lll}
\mathcal{D}_f(\texttt{<10>},\mathbf{I}) & = \mathcal{D}_f(\texttt{(<11> + <12>)},\mathbf{I}) & = (\mathcal{D}_f(\texttt{<11>},\mathbf{I}) \vee \mathcal{D}_f(\texttt{<12>},\mathbf{I})) \\
\mathcal{D}_f(\texttt{<11>},\mathbf{I}) & = \mathcal{D}_f(\texttt{~(<13>)},\mathbf{I}) & = \neg\,(\mathcal{D}_f(\texttt{<13>},\mathbf{I})) \\
\mathcal{D}_f(\texttt{<13>},\mathbf{I}) & = \mathcal{D}_f(\texttt{[P](u,w)},\mathbf{I}) & = \mathcal{D}_t(\texttt{P},\mathbf{I})[\mathcal{D}_f(\texttt{u},\mathbf{I})/d_1, \mathcal{D}_f(\texttt{w},\mathbf{I})/d_2] \\
\mathcal{D}_t(\texttt{P},\mathbf{I}) & = \mathbf{I}(\texttt{P}) & \\
\mathcal{D}_f(\texttt{<12>},\mathbf{I}) & = \mathcal{D}_f(\texttt{~(<14>)},\mathbf{I}) & = \neg\,(\mathcal{D}_f(\texttt{<14>},\mathbf{I})) \\
\mathcal{D}_f(\texttt{<14>},\mathbf{I}) & = \mathcal{D}_f(\texttt{[P](w,v)},\mathbf{I}) & = \mathcal{D}_t(\texttt{P},\mathbf{I})[\mathcal{D}_f(\texttt{w},\mathbf{I})/d_1, \mathcal{D}_f(\texttt{v},\mathbf{I})/d_2]
\end{array}$$

Using $\mathbf{I}(\texttt{P}) = 0$, we find in a bottom-up way:

$$\begin{array}{llll}
\mathcal{D}_f(\texttt{<14>},\mathbf{I}) & = \mathcal{D}_f(\texttt{[P](w,v)},\mathbf{I}) & = 0[\texttt{w}/d_1,\texttt{v}/d_2] & = 0 \\
\mathcal{D}_f(\texttt{<12>},\mathbf{I}) & = \mathcal{D}_f(\texttt{~(<14>)},\mathbf{I}) & = \neg\,(0) & \\
\mathcal{D}_f(\texttt{<13>},\mathbf{I}) & = \mathcal{D}_f(\texttt{[P](u,w)},\mathbf{I}) & = 0[\texttt{u}/d_1,\texttt{w}/d_2] & = 0 \\
\mathcal{D}_f(\texttt{<11>},\mathbf{I}) & = \mathcal{D}_f(\texttt{~(<13>)},\mathbf{I}) & = \neg\,(0) & \\
\mathcal{D}_f(\texttt{<10>},\mathbf{I}) & = \mathcal{D}_f(\texttt{(<11> + <12>)},\mathbf{I}) & = (\neg\,(0) \vee \neg\,(0)) & \\
\mathcal{D}_f(\texttt{<9>},\mathbf{I}) & = \mathcal{D}_f(\texttt{~(<10>)},\mathbf{I}) & = \neg\,((\neg\,(0) \vee \neg\,(0))) & \\
\mathcal{D}_f(\texttt{<8>},\mathbf{I}) & = \mathcal{D}_f(\texttt{E w.(<9>)},\mathbf{I}) & & \\
 & \multicolumn{3}{l}{= (\neg\,((\neg\,(0) \vee \neg\,(0))) \vee \neg\,((\neg\,(0) \vee \neg\,(0))))} \\
\mathcal{D}_f(\texttt{<7>},\mathbf{I}) & = \mathcal{D}_f(\texttt{[N](u,v)},\mathbf{I}) & = d_2[\texttt{u}/d_1,\texttt{v}/d_2] & = \texttt{v} \\
\mathcal{D}_f(\texttt{<6>},\mathbf{I}) & = \mathcal{D}_f(\texttt{(<7> + <8>)},\mathbf{I}) & & \\
 & \multicolumn{3}{l}{= (\texttt{v} \vee (\neg\,((\neg\,(0) \vee \neg\,(0))) \vee \neg\,((\neg\,(0) \vee \neg\,(0)))))} \\
\mathcal{D}_t(\texttt{L u.(<6>)},\mathbf{I}) & \multicolumn{3}{l}{= (\texttt{v} \vee (\neg\,((\neg\,(0) \vee \neg\,(0))) \vee \neg\,((\neg\,(0) \vee \neg\,(0)))))[d_1/\texttt{u}]} \\
 & \multicolumn{3}{l}{= (\texttt{v} \vee (\neg\,((\neg\,(0) \vee \neg\,(0))) \vee \neg\,((\neg\,(0) \vee \neg\,(0)))))} \\
\mathcal{D}_t(\texttt{<5>},\mathbf{I}) & = \mathcal{D}_t(\texttt{L u,v.(<6>)}) & & \\
 & \multicolumn{3}{l}{= (\texttt{v} \vee (\neg\,((\neg\,(0) \vee \neg\,(0))) \vee \neg\,((\neg\,(0) \vee \neg\,(0)))))[d_2/\texttt{v}]} \\
 & \multicolumn{3}{l}{= (d_2 \vee (\neg\,((\neg\,(0) \vee \neg\,(0))) \vee \neg\,((\neg\,(0) \vee \neg\,(0)))))} \\
 & = d_2 & &
\end{array}$$

But $d_2 \neq \mathbf{I}(\texttt{P})$ so we have to recurse in the fixed-point calculation and now use $\mathbf{I}(\texttt{P}) = d_2$:

$$\begin{array}{llll}
\mathcal{D}_f(\texttt{<14>},\mathbf{I}) & = \mathcal{D}_f(\texttt{[P](w,v)},\mathbf{I}) & = d_2[\texttt{w}/d_1,\texttt{v}/d_2] & = \texttt{v} \\
\mathcal{D}_f(\texttt{<12>},\mathbf{I}) & = \mathcal{D}_f(\texttt{~(<14>)},\mathbf{I}) & = \neg\,(\texttt{v}) & \\
\mathcal{D}_f(\texttt{<13>},\mathbf{I}) & = \mathcal{D}_f(\texttt{[P](u,w)},\mathbf{I}) & = d_2[\texttt{u}/d_1,\texttt{w}/d_2] & = \texttt{w} \\
\mathcal{D}_f(\texttt{<11>},\mathbf{I}) & = \mathcal{D}_f(\texttt{~(<13>)},\mathbf{I}) & = \neg\,(\texttt{w}) & \\
\mathcal{D}_f(\texttt{<10>},\mathbf{I}) & = \mathcal{D}_f(\texttt{(<11> + <12>)},\mathbf{I}) & = (\neg\,(\texttt{w}) \vee \neg\,(\texttt{v})) & \\
\mathcal{D}_f(\texttt{<9>},\mathbf{I}) & = \mathcal{D}_f(\texttt{~(<10>)},\mathbf{I}) & = \neg\,((\neg\,(\texttt{w}) \vee \neg\,(\texttt{v}))) & \\
\mathcal{D}_f(\texttt{<8>},\mathbf{I}) & = \mathcal{D}_f(\texttt{E w.(<9>)},\mathbf{I}) & & \\
 & \multicolumn{3}{l}{= (\neg\,((\neg\,(0) \vee \neg\,(\texttt{v}))) \vee \neg\,((\neg\,(1) \vee \neg\,(\texttt{v}))))} \\
\mathcal{D}_f(\texttt{<7>},\mathbf{I}) & = \mathcal{D}_f(\texttt{[N](u,v)},\mathbf{I}) & = d_2[\texttt{u}/d_1,\texttt{v}/d_2] & = \texttt{v} \\
\mathcal{D}_f(\texttt{<6>},\mathbf{I}) & = \mathcal{D}_f(\texttt{(<7> + <8>)},\mathbf{I}) & & \\
 & \multicolumn{3}{l}{= (\texttt{v} \vee (\neg\,((\neg\,(0) \vee \neg\,(\texttt{v}))) \vee \neg\,((\neg\,(1) \vee \neg\,(\texttt{v})))))} \\
\mathcal{D}_t(\texttt{L u.(<6>)},\mathbf{I}) & \multicolumn{3}{l}{= (\texttt{v} \vee (\neg\,((\neg\,(0) \vee \neg\,(\texttt{v}))) \vee \neg\,((\neg\,(1) \vee \neg\,(\texttt{v})))))[d_1/\texttt{u}]} \\
 & \multicolumn{3}{l}{= (\texttt{v} \vee (\neg\,((\neg\,(0) \vee \neg\,(\texttt{v}))) \vee \neg\,((\neg\,(1) \vee \neg\,(\texttt{v})))))} \\
\mathcal{D}_t(\texttt{<5>},\mathbf{I}) & = \mathcal{D}_t(\texttt{L u,v.(<6>)}) & & \\
 & \multicolumn{3}{l}{= (\texttt{v} \vee (\neg\,((\neg\,(0) \vee \neg\,(\texttt{v}))) \vee \neg\,((\neg\,(1) \vee \neg\,(\texttt{v})))))[d_2/\texttt{v}]} \\
 & \multicolumn{3}{l}{= (d_2 \vee (\neg\,((\neg\,(0) \vee \neg\,(d_2))) \vee \neg\,((\neg\,(1) \vee \neg\,(d_2)))))} \\
 & = d_2 & &
\end{array}$$

Now we find $d_2 = \mathbf{I}(\texttt{P})$, so d_2 is the result of the fixed-point term. Substitution leads to:

$$\begin{array}{lll}
\mathcal{D}_t(\texttt{<4>},\mathbf{I}) = \mathcal{D}_t(\texttt{mu P.[<5>]}) & = \mathrm{lfp}(\texttt{P},\texttt{<5>},\mathbf{I}[\texttt{P} := 0]) & = d_2 \\
\mathcal{D}_f(\texttt{<3>},\mathbf{I}) = \mathcal{D}_f(\texttt{[<4>](u,u)},\mathbf{I}) & = d_2[\texttt{u}/d_1,\texttt{u}/d_2] & = \texttt{u} \\
\mathcal{D}_f(\texttt{<2>},\mathbf{I}) = \mathcal{D}_f(\texttt{~(<3>)},\mathbf{I}) & = \neg\,(\texttt{u}) & \\
\mathcal{D}_f(\texttt{<1>},\mathbf{I}) = \mathcal{D}_f(\texttt{E u.(<2>)},\mathbf{I}) & = (\neg\,(0) \vee \neg\,(1)) & \\
\mathcal{D}_f(\texttt{<0>},\mathbf{I}) = \mathcal{D}_f(\texttt{~(<1>)},\mathbf{I}) & = \neg\,((\neg\,(0) \vee \neg\,(1))) & \\
\mathcal{D}_f(\texttt{f},\mathbf{I}) \quad = \neg\,((\neg\,(0) \vee \neg\,(1))) & = 0 &
\end{array}$$

And we find that the formula f is false. Of course, we can only arrive at this result when we are able to simplify PL and PLD formulas and are able to test them for logical equivalence. Using BDDs to represent the formulas solves these problems, since BDD representations are canonical.
□ example 7.4

7.5 Interpreter for μ-calculus

We conclude this chapter with an exposition of the algorithms that form the heart of a simple computer program, called mu, for the boolean μ-calculus. The program is based on our BDD package. We start with introducing a function $\mathcal{B}$ that maps formulas of the logic PLD (and also the logic PL) to BDDs (figure 7.6).

$\mathcal{B}$: PLD $\longrightarrow$ BDD, with for all pld, pld1, pld2 $\in$ PLD,
and for all v $\in$ V $\cup$ Dum:

```
𝓑((pld1 ∨ pld2 )) = bdd_or(𝓑(pld1),𝓑(pld2))
𝓑(¬(pld))         = bdd_not(𝓑(pld))
𝓑(0)              = BDD_0
𝓑(1)              = BDD_1
𝓑(v)              = bdd_var(v)
```

Substitutions and test on equality are handled by:

```
pld1[pld2/v]       : bdd_subst(𝓑(pld1),𝓑(pld2),v)
pld1 = pld2        : bdd_equal(𝓑(pld1),𝓑(pld2))
```

Figure 7.6. Representing PLD formulas as BDDs.

Next, the semantics functions $\mathcal{D}_f$ and $\mathcal{D}_t$ are implemented. We assume that appropriate data structures have been defined to represent formulas F and terms T and the interpretation I. Algorithm 7.1 implements the least fixed-point lfp, algorithm 7.2 implements $\mathcal{D}_f$, and algorithm 7.3 implements $\mathcal{D}_t$.

```
BDD lfp(RV rv,T t,I i)
{
  if (bdd_equal (𝒟t(t,i),i(rv)))
    return i(rv);
  else
    return lfp(rv,t,i[rv := 𝒟t(t,i)]);
}
```

Algorithm 7.1. Implementation of the least fixed-point operator lfp.

```
BDD $\mathcal{D}_f$(F f,I i)
{
  switch (f) {
  case E bv.(g):
    return bdd_or(bdd_subst($\mathcal{D}_f$(g,i),0,bv),
                  bdd_subst($\mathcal{D}_f$(g,i),1,bv));
  case (g + h):
    return bdd_or($\mathcal{D}_f$(g,i),$\mathcal{D}_f$(h,i));
  case ~(g):
    return bdd_not($\mathcal{D}_f$(g,i));
  case 0:
    return BDD_0;
  case 1:
    return BDD_1;
  case bv:
    return bdd_var(bv);
  case [p](g1,g2,...,gn):
    assert(rank(p) <= n);
    R = $\mathcal{D}_t$(p,i);
    for (k:=1; k <= n; k++)
      R:=bdd_subst (R,$\mathcal{D}_f$(gk,i),$d_k$);
    return R;
  }
}
```

Algorithm 7.2. Implementation of the semantics function $\mathcal{D}_f$.

The mu program optionally allows boolean variables to be declared in a so-called domain statement at the beginning of the input; its syntax is:

```
   domain-statement : 'domain' = '{' bvl '}' ';'
```

We allow the user to define the (global) interpretation of predicate symbols by a let construct:

```
   let-statement : 'let' RV = T ';'
```

The meaning is that `let rv = t` modifies the interpretation **I** to become **I**[rv:= $\mathcal{D}_t$(t,**I**)].

Example 7.5 The μ-calculus formula of the previous example 7.4 is fully described by the following mu program input:

```
domain = { u,v,w };
let N = L u,v.v;
A u.[mu P.L u,v.N(u,v)+(E w.P(u,w)&P(w,v))](u,u);
```

As expected, the program correctly evaluates the above formula to the BDD value BDD_0 which denotes false.
□ example 7.5

```
BDD 𝒟_t (T t,I i)
{
  switch (t) {
  case L bv.(g):
    return bdd_subst(𝒟_f(g,i),d_1,bv);
  case L bvl,bv.(g):
    return bdd_subst(𝒟_t(L bvl.(g),i),d_{|bvl|+1},bv);
  case mu rv.[p]:
    assert(p monotone non-decreasing in rv);
    return lfp(rv,p,i[rv := BDD_0]);
  case [p + q]:
    return bdd_or(𝒟_t(p,i),𝒟_t(q,i));
  case ~[p]:
    return bdd_not(𝒟_t(p,i));
  case 0:
    return BDD_0;
  case 1:
    return BDD_1;
  case rv:
    assert(i(rv) != undefined);
    return i(rv);
  }
}
```

Algorithm 7.3. Implementation of the semantics function $\mathcal{D}_t$.

Part III

Programs and Examples

This is the last, but perhaps most interesting, part of the thesis. It again consists of three chapters. Chapter 8 explains many of the implementation details of our BDD package. Chapter 9 shows how BDDs can be set to work in a practical circuit verification tool. Chapter 10 is devoted to some examples of the use of the PTL program.

Chapter 8

The BDD Package

BDDs are a hot topic of interest in the CAD community. The many papers with the word BDD in their title, especially the ones about new applications, suggest that BDDs play an important role in an extensive range of tools.

The BDD package described in this chapter is based on the work of Karl Brace reported in [Brace90] and papers by Richard Rudell [Rudel93]. Apart from the usual logical operations on BDDs, the Eindhoven BDD package (as it is often referred to) includes a rich set of meta routines (e.g. quantification with respect to a set of variables, composition, conversion to sum-of-cubes), routines for statistics (e.g. size, number of minterms), support for development of new operations, and routines to visualize BDDs (e.g. for X-Windows). The package is available via anonymous ftp at `ftp.ics.ele.tue.nl` where it resides in the directory `pub/users/geert`.

Two methods for compacting a BDD graph are discussed in the next section. Section 2 discusses dynamic variable ordering.

8.1 Implementation issues

Complemented Edges
Consider complementing a BDD, i.e., given a BDD for a function f we would like to derive the BDD for the complemented function $\bar{f} = \neg f$. As we have seen in section 4.3 the "not" operation is accomplished by **ITE** (f, **0**, **1**). The effect of the ITE operation will be a BDD that is identical in structure to the original one except for the fact that the terminal nodes are exchanged. Figure 8.1 shows a sample BDD with its complement. It is possible to represent both functions within a single BDD by introducing so-called complemented edges. Such an edge carries a flag

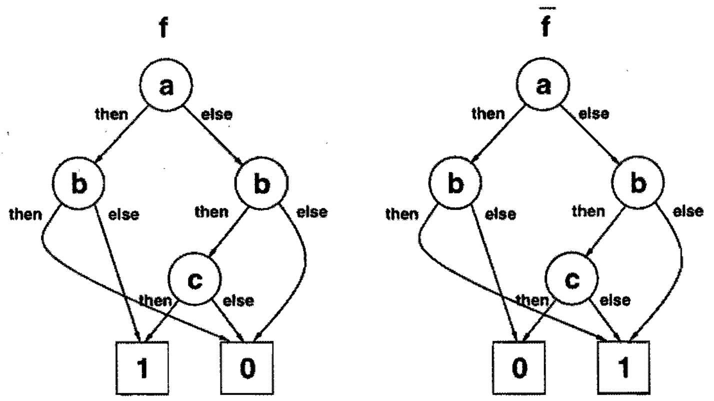


Figure 8.1. BDDs for complementary functions.

that indicates that the function associated with the vertex it points to is to be complemented. Regular edges have no such flag. Use of complemented edges often causes a savings in the number of vertices necessary to represent a given function. Clearly, there is no longer a need for two separate terminal vertices, just one, say the **1** vertex, suffices: the **0** function can be represented as a complemented edge to **1**. In general the complement of any function can easily be constructed by a complemented edge to the top-variable vertex for that function.

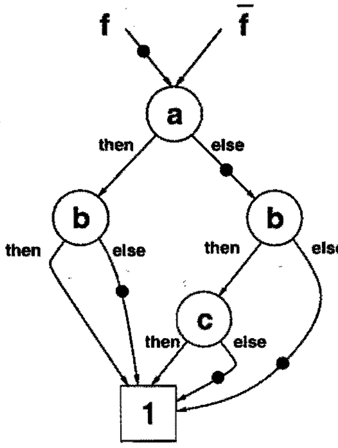


Figure 8.2. A function and its complement as a single BDD.

Care has to be taken however that introduction of complemented edges preserves the canonicity of the BDD. One way of achieving this is by restricting the places in the BDD where such edges may occur. It can be shown that only allowing complemented else-edges preserves canonicity. Figure 8.2 shows how the functions of figure 8.1 are represented as a single BDD using complemented edges. The latter are indicated by marking them with a solid dot.

The flag bit may be implemented in the pointer value itself. This trick is based on the fact that some bits of pointer values will always be fixed because of memory alignment requirements, e.g. on many computers a BDD node has an address that is a multiple of 4, so typically the 2 least significant bits of a BDD pointer value will be 0.

Inverted-input Edges
Some additional compaction of a BDD may be achieved by encoding the polarity of a variable as a flag on the edge leading to a BDD node. The presence of such an inverted-input flag indicates that the node's variable is to be interpreted negated, i.e., the roles of the then and else edges of that node are to be exchanged. In practice it turns out that the savings achieved by inverted-input edges are only minor compared to the savings of incorporating complemented edges. Moreover, use of inverted-input edges severely complicates dynamic variable ordering.

8.2 Dynamic variable ordering

Here we like to briefly summarize the issues involved in applying a dynamic variable ordering technique, i.e., changing the order of the variables during BDD construction. We like to stress that dynamic variable ordering is a very important addition to a BDD package, because it relieves the user of the burden of specifying a good ordering *a priori*, i.e., before he starts constructing and manipulating BDDs. We might define a good ordering as one that permits the function to be represented by a polynomially sized BDD, i.e., $\#\mathsf{nodes} = O(\mathsf{p}(\#\mathsf{vars}))$, where #nodes is the number of BDD nodes to represent the function given a variable ordering, #vars is the number of variables, and p some polynomial. For logic descriptions at the gate-level several static ordering algorithms have been proposed and shown to be successful for many circuits. For instance, for an n-bits binary adder with inputs $\mathsf{A}[\mathsf{n}:1]$ and $\mathsf{B}[\mathsf{n}:1]$ we find that the size of the BDD to represent the carry output is linear in n if we order the variables such that $\mathsf{A}[0] < \mathsf{B}[0] < \mathsf{A}[1] < \mathsf{B}[1] < \cdots < \mathsf{A}[\mathsf{n}] < \mathsf{B}[\mathsf{n}]$. However, for higher level specifications it becomes much harder to apply those methods short of first deriving a gate-level representation. (What is a good variable ordering for a circuit described as a VHDL process?) Also, even when a static ordering heuristic is used, it turns out that dynamic variable ordering often can substantially improve on the intermediate and final BDD sides. In particular when BDDs are used in the area of verification, it is our opinion that dynamic variable ordering becomes a mandatory prerequisite for successfully handling large circuits.

8.2.1 Principles

It is a known fact that the size of a (reduced ordered) BDD for a given boolean function may drastically change when a different ordering of the variables that label the BDD nodes is adopted. The problem with changing the order dynamically, is that one has to maintain canonicity. In an implementation, canonicity is achieved through a so-called unique table of BDD nodes: a node is identified by its pointer, and a new node is created only when it is not yet present in the unique table, otherwise the pointer stored in the table is returned. It would be very inefficient to construct an entire BDD for every different ordering that is tried. Therefore, dynamic variable ordering will be based on a succession of local modifications to the BDD, each of which can easily be made to preserve canonicity. An obvious local modification is the swapping of two consecutive variables.

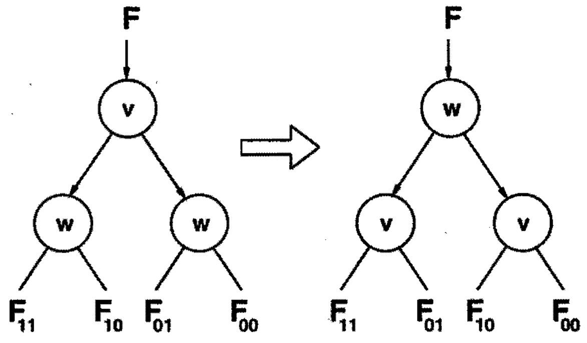


Figure 8.3. Effect of variable swap on BDD.

By repeatedly swapping neighbouring variables it is possible to generate every possible permutation. However, for practical purposes a full exploration of the 'variable orders space' cannot be tolerated; a simple local-search approach with limited hill-climbing is chosen instead. In this approach, each variable is tried at all possible positions in the order while the ranks of the other variables remain the same; we call this 'sifting the variable'. The search for the best position of that one variable may still lead to the construction of unacceptably large intermediate BDDs, hence the search is aborted as soon as some predefined BDD size limit is exceeded (we allow a size increase of at most 5%). It is easy to see that putting one variable at its best position (algorithm 8.1) takes $O(\#\text{vars} \cdot \#\text{nodes})$ time.

It makes sense to treat the variables in order of frequency: the variable with the most occurrences in the BDDs is sifted first. The rationale is that by changing the position of this variable the decrease in size of the BDDs will be the largest.

8.2.2 Implementation issues

A problem with dynamic variable ordering is to decide where and when to apply it. If done inside the recursive ITE construction routine, it might violate its invariants and cause the need for a restart of the top-level call. Rudell [Rudel93] is in favour of the latter suggestion, and argues that precisely because ITE is the major

```
best_rank:=rank:=orig_rank:=RANK(var);
best_size:=nr_nodes_alive();

/* Move 'var' down the order: */
for (next_rank:=rank+1; next_rank < max_rank; next_rank++) {
  swap_levels(rank, next_rank);
  rank:=next_rank;
  if (nr_nodes_alive() < best_size) {
    best_size:=nr_nodes_alive();
    best_rank:=rank;
  }
  else
  if (nr_nodes_alive() > 1.05 * best_size)
    break;
}

/* Back up to original rank: */
for (prev_rank:=rank-1; prev_rank >= orig_rank; rank:=prev_rank, prev_rank--)
  swap_levels(prev_rank, rank);

/* Up and back down to best rank require similar processing; omitted here. */
```

Algorithm 8.1. Sifting a single variable to its best position.

source for new nodes, dynamic ordering should be invoked right there. We, however, decided to allow dynamic ordering to take place outside recursive ITE calls only, but potentially after every top-level call. It remains to decide which criteria to define that trigger a call. Rudell uses as a measure an absolute bound on the total number of BDD nodes, that after each dynamic reordering is reset to twice the then existing number of nodes. We adopt a mixed approach using both a relative and absolute threshold. We were able to achieve results comparable with Rudell's [MetsA94]. The absolute threshold criterion we use is more or less the same as described above. A relative threshold is introduced to be able to anticipate sharp increases in BDD size as a function of the number of top-level ITE calls. Dynamic ordering will be triggered when the increase exceeds a factor of 2, i.e., a top-level ITE call results in twice the number of nodes as compared to the number of nodes prior to the call. The explanation is that the majority of BDD operations takes two operands, and that empirical evidence shows that the size of the result is of the order of the sum of the sizes of the operands; in the worst case it would be the product.

The whole point with dynamic variable ordering is that on one hand it is a very useful, and for some applications even vital, feature of a BDD package, but on the other hand it takes $O(\#\text{vars}^2 \cdot \#\text{nodes})$ time for just 1 invocation, and therefore should not be called upon too liberally, especially when many variables are involved. Clearly, there are a number of conflicting interests:

- Dynamic variable ordering should be done as soon as the current ordering is found to be rather poor.

- The 'quality' of an ordering can be assessed relative to the functions that are momentarily represented. There is usually no way to predict the sequence of future BDD operations on the currently represented functions.
- If the initial (or some intermediate) ordering is good, then the next call to dynamic ordering should be postponed as much as possible.
- When the functions to be represented are such that no good ordering exists, dynamic ordering should be refrained from completely.

Our solution is to fix the total time spend in one call to some reasonable constant, say 10 (cpu) seconds, and use a more sophisticated trigger-criterion as explained above. As our experiments point out there is no such thing as one medicine that cures all. Some tuning of the dynamic variable parameters may often give better results. The effect of dynamic variable ordering is illustrated for a 16-bit data input/output, 4-bit control rotator circuit in figure 8.4 and figure 8.5 (see also table 8.1).

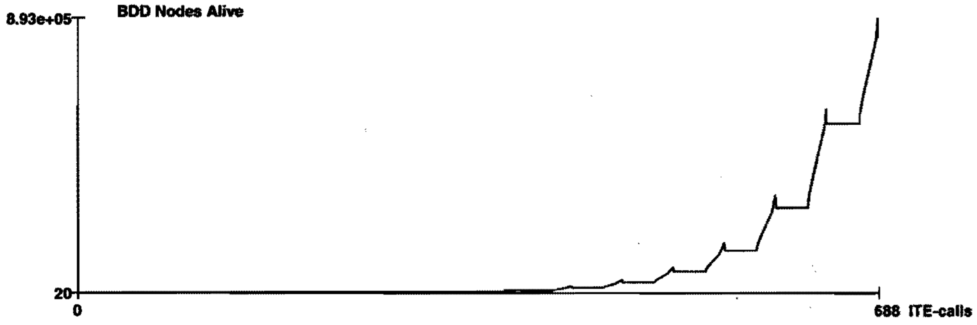


Figure 8.4. #nodes as function of top-level ITE calls; no dynamic ordering. Both axes are linearly scaled.

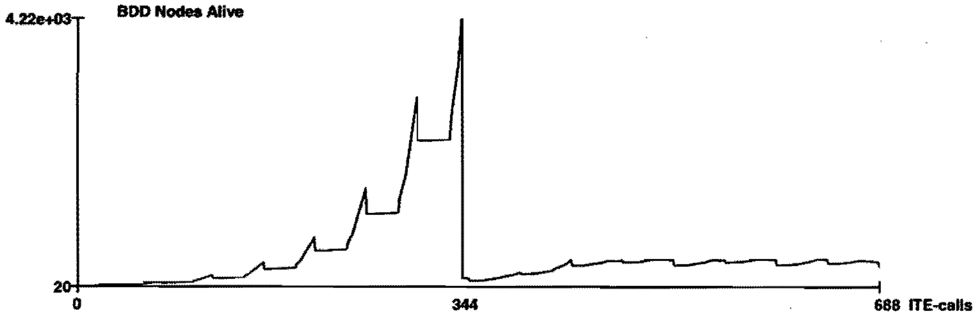


Figure 8.5. #nodes as function of top-level ITE calls; with dynamic ordering. Both axes are linearly scaled.

Figure 8.4 shows that the circuit requires a million BDD nodes which is achieved after 688 top-level ITE calls. With dynamic variable ordering (figure 8.5), the peak number of BDD nodes is slightly more than 4000. The sharp decrease in number of nodes occurring after 344 ITE-calls is the result of a single dynamic variable reordering invocation.

8.2.3 Examples and results

Table 8.1 indicates the effect of dynamic variable ordering on some typical benchmarks, all except the rotator taken from [Kropf94]. #nodes is the size of the final shared BDDs for all output functions. The runtime is in seconds on a HP9000/s755. The 'Good' and 'Bad' orderings are obtained manually based on intuition, we don't claim them to be the best, resp. worst; 'Dynamic' means dynamic variable ordering is on during BDD construction, 'Bad' is taken as initial ordering, and at the end dynamic ordering is applied exhaustively until no more gain is obtained. Note that for the multiplier dynamic ordering gets stuck in a local minimum and finds an even worse order than our Bad one. The results for Min_Max include BDDs for both the regular outputs and the next-state functions. We see that what we think as good can sometimes be improved upon.

Circuit	Good		Bad		Bad+Dynamic	
	#nodes	secs	#nodes	secs	#nodes	secs
16-bit rotator	81	<1	1081328	56	81	1
8-bit adder	36	<1	751	<1	36	<1
16-bit adder	76	<1	196575	16	123	1
32-bit adder	156	<1	>1000000	80	452	4
32-bit alu	8869	<1	>1000000	83.4	4341	8.2
64-bit alu	17829	<1	>1000000	81.4	9487	47.2
128-bit alu	35749	1.9	>1000000	79.1	18086	149.6
256-bit alu	71598	4.0	>1000000	82.2	44870	697.9
8-bit Min_Max	890	<1	79007	6	883	3
16-bit Min_Max	3310	<1	>1000000	50	3295	16
32-bit Min_Max	12566	2	>1000000	39	39265	86
12-bit multiplier	605883	255	1324674	340	1494828	2500

Table 8.1. Effect of dynamic variable ordering.

Chapter 9

Application of BDDs in a Hardware Description Language

9.1 Introduction

In this chapter we address the application of BDD-based combinational circuits equivalence checking in the context of a Hardware Description Language (HDL). The idea is to discuss several important aspects of the design of a combinational circuit verification tool. Such a tool has actually been developed by the author as part of IBM's BSN (Boolean Specification Networks) project. We will focus on how to deal with typical HDL constructs; the language issues, in particular the syntax and semantics, are of secondary concern. The HDL we will use in our examples is modelled after the proprietary BSN language. Syntactically, the HDL is similar to the C programming language.

For the moment we only consider the combinational subset of the HDL, i.e., we assume that the HDL descriptions do not contain memory elements and we also assume the absence of loops. We presuppose that each well-formed HDL description may be interpreted as defining a set of boolean functions over the primary inputs, i.e., the description associates a boolean function with each of its primary outputs. This means that the semantics is such that any circuit described in the language can be automatically transformed to a combinational gate-level description in the same language. Details can be found in the M.Sc. thesis of [Schuu94]. Although no formal semantics has been defined for our HDL, we are confident that the intended meaning of the syntactic constructs is sufficiently explained in the accompanying text and by the examples.

Our HDL allows parameterized, hierarchical descriptions of combinational circuits, see figure 9.1. The leaves of the hierarchy are formed by instances of behavioural module definitions. These may be combined to form structural module definitions (the interior nodes). The root is a bound instance of a module for which all parameters are given definite values. Parameters are typically used to define modules in a generic way, e.g. an n-bits adder will have n as its parameter.

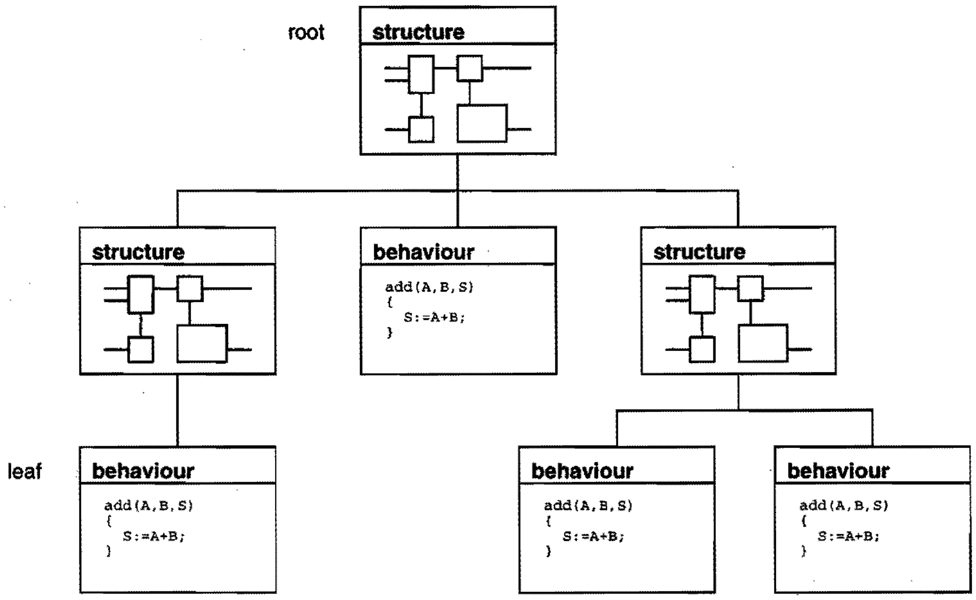


Figure 9.1. Typical hierarchy tree of a design.

A structural module definition contains statements on how its instances are to be connected. Structural module definitions may be recursive, and instantiation statements may be conditional with respect to the module's parameters.

We start this chapter with a presentation of the concrete syntax of the language. A formal definition of its semantics is beyond the scope of this thesis. Instead, we rely on the reader's familiarity with similar hardware description languages; where necessary an informal explanation will be given. Examples are included to help understand and appreciate certain constructs. Section 9.3 discusses a number of useful syntactical transformations. These transformations result in a gate-level description of a circuit. An interpretation for the HDL in terms of BDDs is presented in section 9.4. Section 9.5 gives an overview of an approach to verify the equivalence of two designs written in the HDL. Section 9.6 discusses several techniques that are employed to cope with cases for which the BDDs turn out to become too big.

9.2 A sample HDL

Figures 9.2 through 9.4 present the concrete syntax of the HDL which we will use throughout this chapter. For lack of a better name we will simply name this language HDL.

```
circuit-description ::= { module-definition }+ circuit-instance .

module-definition ::= b-module-definition | s-module-definition .

b-module-definition ::= ’behaviour’ module-interface b-block .

s-module-definition ::= ’structure’ module-interface s-block .

module-interface ::= name [ formal-params ] ’(’ pin-dcls ’)’ .

formal-params ::= ’[[’ { name / ’,’ }+ ’]]’ .

pin-dcls ::= { ( ’input’ | ’output’ ) pins / ’;’ }+ .

pins ::= { pin / ’,’ }+ .

circuit-instance ::= ’circuit’ instance ’;’ .
```

Figure 9.2. HDL grammar rules part I: circuit/module interface.

```
s-block ::= ’{’ [ net-dcls ] { s-statement } ’}’ .

net-dcls ::= { ’net’ nets ’;’ }+ .

nets ::= { net / ’,’ }+ .

s-statement ::= instance-call
              | guard s-statement
              | iterator s-statement
              | s-block .

instance-call ::= instance ’(’ port-exprs ’)’ ’;’ .

instance ::= [ name ’:’ ] module-name [ actual-params ] .

actual-params ::= { const-expr / ’,’ }+ .

port-exprs ::= { simple-expr / ’,’ }+ .

simple-expr ::= { ( facility | const-expr ) / ’.’ }+ .

facility ::= ( pin-name | net-name | local-name | ’–’ ) [ subscript ] .

guard ::= ’[[’ cond-expr ’]]’ .

iterator ::= ’(’ name ’:’ const-expr ’..’ const-expr ’)’ .
```

Figure 9.3. HDL grammar rules part II: structure.

HDL, though concise in syntax, is sufficiently powerful to stand as a model for many of its real-life counter-parts like Verilog and VHDL. Note that the description of structure and behaviour is clearly separated. The statements of a structural module specify connections of instances via nets. Their order has no meaning. Statements of a behavioural module are interpreted sequentially and obey the so-called single-assignment rule: a local or output pin must be assigned a value exactly once. Locals, nets, and pins represent physical entities; they are not variables in a programming language. Furthermore, locals and output pins must be assigned a value before they occur in a right-hand side expression of an assignment statement (define-before-use). These conditions ensure that a behavioural module is combinational.

```
b-block ::= '{' [ local-dcls ] { b-statement } '}' .

local-dcls ::= { 'local' locals ';' }+ .

locals ::= { local / ',' }+ .

b-statement ::= assignment-stat
              | iterator b-statement
              | b-block .

assignment-stat ::= lhs '=' expression ';' .

lhs ::= { facility / '.' }+ .
```

Figure 9.4. HDL grammar rules part III: behaviour.

A number of syntactic categories have been left unspecified. Their precise definition is of no concern, and their intention should be clear from the examples and the next brief discussion:

name

Some identifier.

pin, local, net

Pins, nets, and locals are declared by stating their name optionally followed by a bit-range specification. If no range is present, the object is declared scalar and is 1-bit wide; otherwise the range specification looks like '[DH:DL]' with the requirement DH ≥ DL and then the object is declared 1-dimensional with bit-width DH-DL+1. DH and DL are the object's declared bounds. In the context of arithmetic operations, the bit with smallest index (DL) is considered the least significant bit.

pin-name, local-name, net-name, –

The identifier of the declared object or the generic sink object '–'. The sink is used to explicitly indicate that one is not interested in a certain value, e.g. one might like to ignore the carry bit resulting from an addition:

```
- . S[4:1] = A[4:1] + B[4:1];
```

Objects declared as 1-dimensional may be subscripted. A subscript looks like '[H:L]' with H ≥ L; [N] is equivalent to [N:N]. Absence of a subscript

Repertoire of Operators & Predefined Functions	
Symbol/Notation	Meaning
lg2	ceiling of base 2 logarithm (const-expr operand)
decode(E)	full decode
reverse(E)	reverses bits of E
extend(E,n)	copies MSB of E till size is n
trunc(E,n)	removes MSB of E till size is n
min(E1,E2)	minimum of E1 and E2
max(E1,E2)	maximum of E1 and E2
**	exponentiation (const-expr operands)
*	multiplication (const-expr operands)
/	division (const-expr operands)
%	modulo (const-expr operands)
+	addition with carry
−	subtraction with borrow
<	less than
<=	less than or equal
==	equality
!=	inequality
>=	greater than or equal
>	greater than
~	bitwise complement (prefix)
'	bitwise complement (postfix)
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<->	bitwise EQUIV (XNOR)
->	bitwise IMPLY
.	concatenation
<<	left shift (shift distance is const-expr)
>>	right shift (shift distance is const-expr)
rol	left rotate (rotate distance is const-expr)
ror	right rotate (rotate distance is const-expr)
C ? T : E	conditional expression, if C then T else E
op (i:L..H) E	iteration expression; op may be &, |, <->, ^, +
op E	op reduction over all bits of E; op may be &, |, <->, ^, +

Table 9.1. Typical elements of expressions in HDL.

for an object means: use the complete object; for a 1-dimensional object this is equivalent to a subscript of [DH:DL]. The sink '–' may also be subscripted and then is of size H-L+1, otherwise it has size 1.

module-name

Identifier of a previously defined module.

const-expr

Constant expression, i.e., an expression such that when all appropriate iteration variables and parameters are bound to a constant value, the expression can be evaluated to a constant value.

cond-expr

Conditional expression, a constant expression that evaluates to 0 or 1.

expression

Expressions are defined in the usual way, allowing infix notation for operators and a number of predefined functions. Parentheses may be used for clarity and to overrule operator precedence. Primitives are facilities and numbers. Numbers may be expressed in various radii and are in principle unbounded. Table 9.1 lists the operators and some built-in functions. The requirement that operands have matching bit-widths for various binary operators is relaxed: the operands are right-aligned (i.e., aligned at their least-significant side) and the smaller operand is padded with zero bits on the left (i.e., at its most-significant side). In assignments we will also assume a similar treatment for the right-hand side expression: it will be truncated if it is too wide, or padded with zeroes in case it is too small, to match the width of the left-hand side.

Figure 9.5 shows an example of an 8-bit parity checker.

```
/* b = a[1] xor a[0] */
behaviour xor2(input a[1:0]; output b)
{
  b = ^a;
}

structure Parity_Tree[[N]](input A[2**N:1]; output B)
{
  [[ N == 1 ]]
    xor2(A, B);

  [[ N > 1 ]] {
    net tmp[2:1];

     left[N] : Parity_Tree[[N-1]](A[2** N    :2**(N-1)+1], tmp[2]);
    right[N] : Parity_Tree[[N-1]](A[2**(N-1):           1], tmp[1]);
    xor2(tmp, B);
  }
}

circuit parity_8 : Parity_Tree[[3]];
```

Figure 9.5. 8-bit parity circuit in HDL.

The behavioural module xor2 consist of a single statement, assigning to b the result of xor-ing all bits of a, using the exclusive-or operator ^ as a reduction operator. The structural module is generic in the parameter N and uses recursion.

So, Parity_Tree[[1]] results in an instance-call to the behavioural module xor2. For N > 1, Parity_Tree[[N]] is constructed from two instances of Parity_Tree[[N-1]]. Each is supplied with half the bits of input A, and their results are combined using an xor2 instance. Figure 9.6 shows a schematic of this circuit.

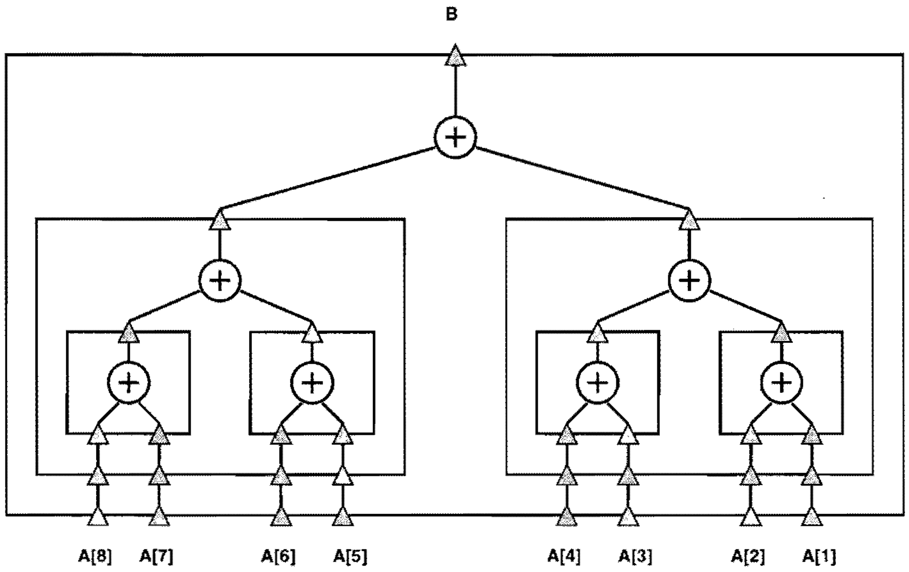


Figure 9.6. Parity_Tree[[3]] schematic. The ⊕-nodes stand for xor2 instances.

9.3 Source-level transformations

It should be obvious that many constructs of our sample HDL are predominantly intended to ease the effort of circuit specification: parameterized modules, iteration, and recursion do not add to the expressive power of the language. The same functionality can always be specified without them. This observation directly leads to a number of transformations on the HDL text that produce a more primitive description while preserving its meaning.

Transformation	Effect
1. Binding	Determining actual values for formal parameters and evaluating constant expressions
2. Unfolding	Unrolling iterations
3. Unwinding	Making recursive calls explicit
4. Flattening	Substituting structure contents for instance calls
5. Mapping	Converting behaviour to structure

Table 9.2. Summary of transformations and their effect.

Transformations 1, 2, and 3 of table 9.2 are best performed simultaneously. This will lead to a circuit description where all constant-expressions are reduced to numbers. Instantiation of generic modules and recursion unwinding generally create new module definitions and instance-calls that need to be properly named. Our rule is: `obj[[n]]` with `n` bound to the actual value `k` becomes the new object identified by `obj#k`. Figure 9.7 shows the example circuit after applying the first 3 transformations.

```
behaviour xor2(input a[1:0]; output b)
{
  b = ^a;
}

structure Parity_Tree#1(input A[2:1]; output B)
{
  I_1 : xor2(A, B);
}

structure Parity_Tree#2(input A[4:1]; output B)
{
  net tmp[1:2];

   left[2] : Parity_Tree#1(A[4:3], tmp[2]);
  right[2] : Parity_Tree#1(A[2:1], tmp[1]);
  I_2 : xor2(tmp, B);
}
```

```
structure Parity_Tree#3(input A[8:1]; output B)
{
  net tmp[1:2];

   left[3] : Parity_Tree#2(A[8:5], tmp[2]);
  right[3] : Parity_Tree#2(A[4:1], tmp[1]);
  I_2 : xor2(tmp, B);
}

circuit parity_8 : Parity_Tree#3;
```

Figure 9.7. After binding, unfolding iterations, and unwinding recursion.

Another useful transformation is the substitution of structural module instances by their content. This is called flattening. To avoid name clashes, some objects (typically nets and instances) must be renamed. The circuit description for our example after flattening is shown in figure 9.8.

A more challenging transformation is mapping behavioural modules onto a network of primitive modules. Our goal is to convert a b-module definition into a semantically equivalent s-module definition. This can be done by attaching a meaning to the operators present in the b-module body (b-block) in terms of a set of primitive module definitions. This conversion is best implemented using a syntax directed approach.

```
behaviour xor2(input a[1:0]; output b)
{
  b = ^a;
}

structure Parity_Tree#3(input A[8:1]; output B)
{
  net N_0[2:1], N_1[2:1], N_2[2:1];

  I_0 : xor2(A[8:7], N_1[2]);
  I_1 : xor2(A[6:5], N_1[1]);
  I_2 : xor2(N_1, N_0[2]);
  I_3 : xor2(A[4:3], N_2[2]);
  I_4 : xor2(A[2:1], N_2[1]);
  I_5 : xor2(N_2, N_0[1]);
  I_6 : xor2(N_0, B);
}

circuit parity_8 : Parity_Tree#3;
```

Figure 9.8. Flattened circuit.

We assume that transformations 1, 2, and 3 are already applied. So the only possible b-statement is the assignment-stat, and the only possible s-statement is the instance-call. Also, iteration expressions will be absent. The statements of the b-module are processed sequentially. Declarations of locals become net declarations. An assignment statement results in a number of instance-calls to certain primitive modules. There will be one such call for each operator and predefined function occurring in the right-hand side expression of the assignment statement. Table 9.3 below lists the primitive modules defined for some operators and predefined functions.

Operator	Primitive module
& E	`$AndReduce`
^ E	`$XorReduce`
decode(E)	`$Decode`
*	`$Multiply`
+	`$Add`
<	`$Less`
~	`$Not`
&	`$And`
|	`$Or`
...	...

Table 9.3. 'Technology' mapping.

The primitive modules reside in a separate file, the so-called technology file. Typically, a primitive module is a structural module that is parameterized with respect to the bit-width of its input and output pins and consists of a network of basic gates.

The left-hand side of an assignment statement is converted to a port expression for the output pin of a call to the special `$Connect` module (figure 9.9). The right-hand side is converted to a port expression for the input pin of that connect module.

```
behaviour $Connect[[N]](input A[N:1]; output Y[N:1])
{
  Y = A;
}
```

Figure 9.9. Generic description of connect module in HDL.

Table 9.4 summarizes the conversion steps.

Behaviour	Structure	Remark
b-module-definition	s-module-definition	
b-block	s-block	
local-dcls	net-dcls	
assignment-stat	{ instance-call }+	among them `$Connect`
lhs	port-expr	output of `$Connect`
expression	port-expr	input of `$Connect`

Table 9.4. Conversion of b-module to s-module.

Here is an example to illustrate the mapping transformation:

```
behaviour M(input A[3:0], B; output Y[2:1])
{
  local t[3:0];

  t = B.A[2:0];
  Y.-[1:0] = ~t & extend(A[3] | B, 4);
}
```

The behavioural module `M` accomplishes the following: first the scalar input `B` is concatenated with the 3 least significant bits of `A`. Then, all 4 bits of `t` are complemented and bitwise AND-ed with the 4-bit wide extension of the OR of bit `A[3]` and `B` (`extend` duplicates the MSB of its first argument to produce a result as wide as the second argument indicates). The 2 most significant bits of the final expression are assigned to `Y` and the remaining 2 least significant bits are ignored, i.e., left unconnected (this is what `-[1:0]` means).

When converted to a network of basic gates we obtain the following structural module. Note that additional nets (N_0, N_1, and N_2) are introduced to interconnect the generated instance-calls.

```
structure M(input A[3:0], B; output Y[2:1])
{
  net t[3:0], N_0[3:0], N_1, N_2[3:0];

  I_0 : $Connect[[4]](B.A[2:0], t);
  I_1 : $Not[[4]](t, N_0);
  I_2 : $Or[[1]](A[3], B, N_1);
  I_3 : $And[[4]](N_0, N_1.N_1.N_1.N_1, N_2);
  I_4 : $Connect[[4]](N_2, Y.-[1:0]);
}
```

The s-module M is also depicted in figure 9.10. Often it is possible to avoid the generation of spurious $Connect instances.

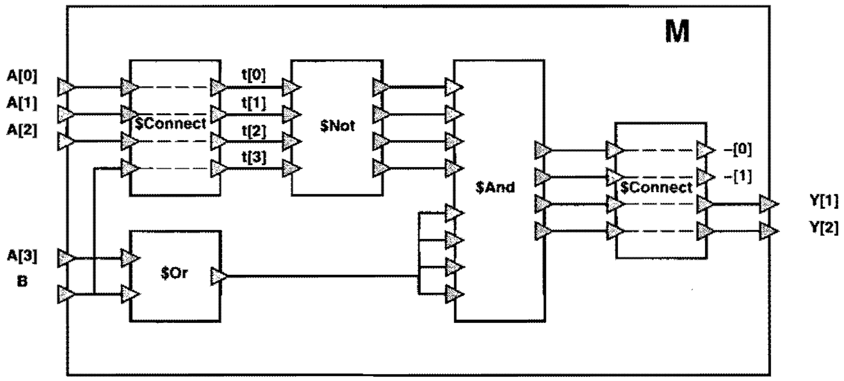


Figure 9.10. The mapped example module M.

The module definitions in the technology file are themselves expressed in HDL. It is not too difficult to express all the necessary $-primitives in terms of a small set of basic gates. Then a second application of binding, unfolding, and unwinding converts the circuit to a true gate-level description. As an example, the gate-level implementation for the xor-reduction operator is given in figure 9.11. Figure 9.12 shows the gate-level description of the Parity_Tree example in terms of 2-input XOR gates.

```
/* Y = ^A; N >= 1 */
structure $XorReduce[[N]](input A[N:1]; output Y)
{
  [[ N = 1 ]]
    CONNECT(A, Y);

  [[ N = 2 ]]
    XOR(A[1], A[2], Y);

  [[ N >= 3 ]] {
    net T[N-1:2];

    XOR(A[1], A[2], T[2]);
    (i : 3..N-1)
      XOR(A[i], T[i-1], T[i]);
    XOR(A[N], T[N-1], Y);
  }
}
```

Figure 9.11. Generic description for mapping xor-reductor to XOR gates.

```
structure Parity_Tree#3(input A[8:1]; output B)
{
  net N_0[2:1], N_1[2:1], N_2[2:1];

  I_0 : XOR(A[8], A[7], N_1[2]);
  I_1 : XOR(A[6], A[5], N_1[1]);
  I_2 : XOR(N_1[2], N_1[1], N_0[2]);
  I_3 : XOR(A[4], A[3], N_2[2]);
  I_4 : XOR(A[2], A[1], N_2[1]);
  I_5 : XOR(N_2[2], N_2[1], N_0[1]);
  I_6 : XOR(N_0[2], N_0[1], B);
}

circuit parity_8 : Parity_Tree#3;
```

Figure 9.12. Bound, unfolded, unwound, flattened, and mapped.

9.4 BDD interpretation of behavioural modules

The transformations discussed in the previous section make it possible to derive a boolean network from an HDL circuit description. Each basic gate-level HDL element has its corresponding BDD operation as shown in table 9.5.

BDDs for the primary outputs of the circuit are calculated by applying the BDD operations for each element in a proper topological order to the BDDs for intermediate nets, starting with the BDDs for each primary input and any constant net values (0 or 1 for false resp. true). The terms "topological sorting" and "rank ordering" will be used interchangeably.

HDL basic element	BDD operation
CONNECT	bdd_assign
NOT	bdd_not
AND	bdd_and
NAND	bdd_nand
OR	bdd_or
NOR	bdd_nor
XOR	bdd_xor
XNOR	bdd_xnor
input A	bdd_create_var("A")
0	bdd_0
1	bdd_1

Table 9.5. HDL basic element and corresponding BDD operation.

This process is explained by means of the 4-bit adder circuit of figure 9.13.

```
behaviour add[[N]](input a[N-1:0], b[N-1:0]; output cout, s[N-1:0])
{
  cout.s = a + b;
}
circuit adder_4 : add[[4]];
```

Figure 9.13. Specification of a 4-bit adder.

After the transformations, the description of figure 9.14 results. The instances are shown in rank order: I_0 is first, then I_1, et cetera. The BDDs for the outputs cout, s[3] till s[0] are (automatically) drawn in figure 9.16. Instead of explicitly mapping the circuit onto a set of basic gates and then deriving the BDDs, it is also possible to directly interpret the HDL operations by BDD (vector) operations. Also, the requirement that the circuit first be flattened may be dropped when the circuit is rank-orderable, i.e., there are no apparent loops in the structural modules. The expressions occurring in a behavioural module are interpreted by a (virtual, i.e., software simulated) BDD-vector stack machine. The stack organization is depicted in figure 9.15.

Each item on the BDD-vector stack is a vector of BDDs. For uniformity, items of size 0 are allowed. Operations on the stacked items pop 0 or more top items off the stack and possibly push a result back on the stack. Of course many optimisations are possible, e.g. for some operations the actual popping and pushing may be avoided by directly operating on the elements of the stacked vector. Algorithm 9.1 shows a glimpse of the main interpreter routine; many details have been left out.

```
structure add#4(input a[3:0], b[3:0]; output cout, s[3:0])
{
  net N_0[4:2], N_1, N_2, N_3, N_4, N_5, N_6,
      N_7, N_8, N_9, N_10, N_11, N_12;

  I_0 : XOR(b[0], 0, N_11);
  I_1 : AND(b[0], 0, N_10);
  I_2 : XOR(N_11, a[0], s[0]);
  I_3 : AND(a[0], N_11, N_12);
  I_4 : OR(N_12, N_10, N_0[2]);
  I_5 : XOR(b[1], N_0[2], N_8);
  I_6 : AND(b[1], N_0[2], N_7);
  I_7 : XOR(N_8, a[1], s[1]);
  I_8 : AND(a[1], N_8, N_9);
  I_9 : OR(N_9, N_7, N_0[3]);
  I_10 : XOR(b[2], N_0[3], N_5);
  I_11 : AND(b[2], N_0[3], N_4);
  I_12 : XOR(N_5, a[2], s[2]);
  I_13 : AND(a[2], N_5, N_6);
  I_14 : OR(N_6, N_4, N_0[4]);
  I_15 : XOR(b[3], N_0[4], N_2);
  I_16 : AND(b[3], N_0[4], N_1);
  I_17 : XOR(N_2, a[3], s[3]);
  I_18 : AND(a[3], N_2, N_3);
  I_19 : OR(N_3, N_1, cout);
}
```

Figure 9.14. Gate-level 4-bit adder (instances are in rank-order).

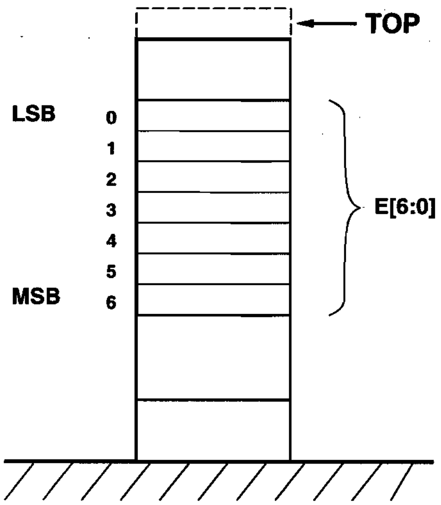


Figure 9.15. BDD-vector stack.

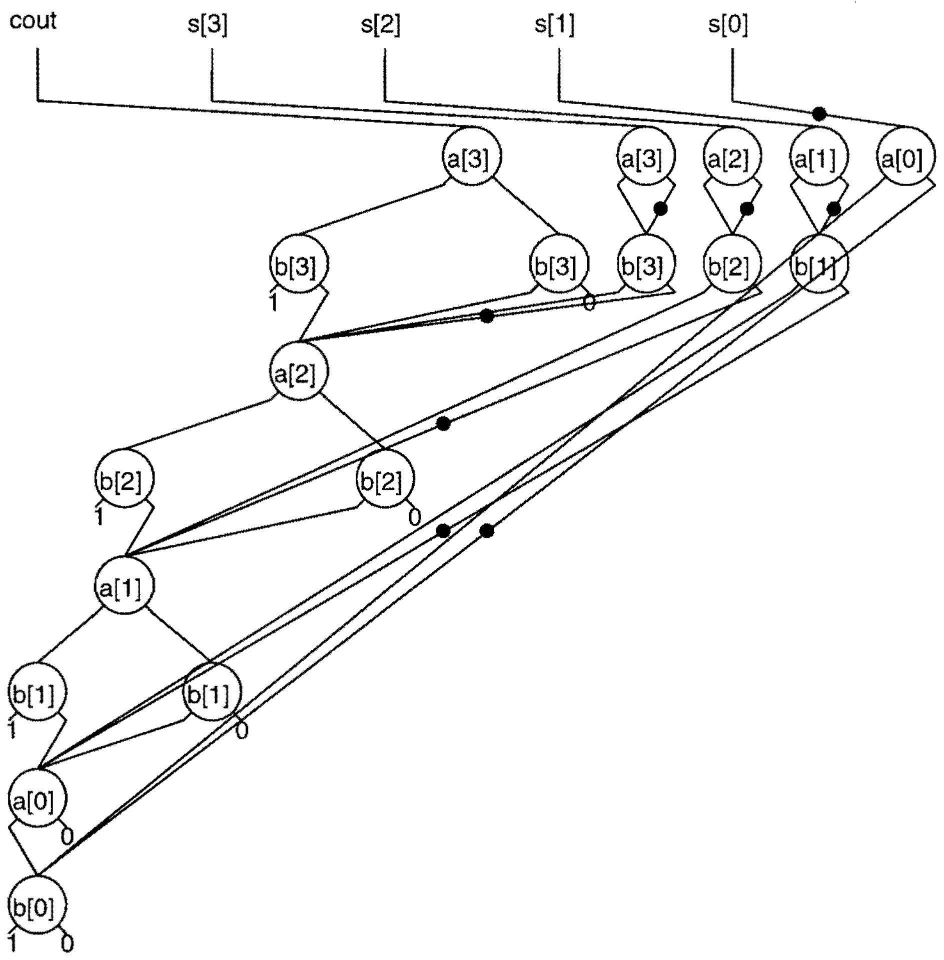


Figure 9.16. BDDs for `adder_4` (a dot on an edge means complementation).

```
void bdds_apply(Opcode op)
{
  switch (op) {
  case BDDS_NOT_OP:
    A:=bdds_pop();
    for (i:=0; i < size(A); i++)
      R[i]:=bdd_not(A[i]);
    bdds_push(R);
    break;

  case BDDS_SUB_OP:
  case BDDS_ADD_OP:
    B:=bdds_pop();
    A:=bdds_pop();

    Ci:=bdd_0();
    for (i:=0; i < size(A); i++) {
      t1  :=bdd_xor(B[i], Ci);
      S[i]:=bdd_xor(t1, A[i]);
      if (op = BDDS_SUB_OP) A[i]:=bdd_not(A[i]);
      t2:=bdd_and(A[i], t1);
      t3:=bdd_and(B[i], Ci);
      Ci:=bdd_or(t1, t2);
    }
    S[i]:=Ci;
    bdds_push(S);
    break;
    ...
  } /*switch*/
}
```

Algorithm 9.1. Sketch of HDL interpreter.

There are a number of options in the organization of processing a circuit to derive its BDDs:

1. The circuit is first fully flattened and the instances are rank ordered. Then one proceeds as mentioned in the beginning of this section.

   Discussion: This approach shifts the burden of handling hierarchy and scope levels from the BDD interpreter to the transformations. It presupposes that the BDD interpreter has knowledge of the set of basic gates (technology file).

2. The circuit is not, or only partially flattened, and each structural module is locally rank ordered. Then all behavioural modules are processed and BDDs for their outputs in terms of their inputs are determined. By means of BDD composition the BDDs for the whole circuit are found.

   Discussion: The BDD interpreter must be able to handle hierarchy. It is to be expected that this method will require a lot of memory to store the intermediate BDDs for all behavioural modules simultaneously. Also the compositions are rather time consuming and have the drawback that no dynamic variable ordering may be done in the meantime.

3. Same as above, but now each behavioural module is processed only at the moment its BDDs are required by an instance (demand driven), and the BDD calculations start with the BDDs found for the inputs of the module. In this way no *a posteriori* composition is necessary.

   Discussion: This is probably the most cost-effective method. Demand-driven (or lazy evaluation) is usually a good scheme to adopt. At any time, only the BDDs for the partially processed circuit need to be stored, any intermediate results may be freed. No compositions occur so dynamic variable ordering can be fully exploited. The drawback is that a behavioural module will be processed as many times as there are instances of it. However, typically the inputs BDDs for it will be different in each invocation anyway, and if not, the memory function present in the BDD package will often avoid the unnecessary recalculation of known BDDs.

Table 9.6 clearly demonstrates the superiority of the 3-rd method when compared with the 1-st method. The 2-nd method has not been implemented. The example circuit is `add[[N]]` and a good variable order was manually supplied. For a fair comparison, the listed runtimes for deriving BDDs for the mapped circuit (method 1 in column 3) do not include the mapping itself (which is reported separately in column 4).

N	Method 3	Method 1	Mapping	#BDD nodes
8	0.0	0.0	0.2	39
16	0.0	0.1	0.2	79
32	0.0	0.2	0.2	159
64	0.0	0.5	0.3	319
128	0.0	1.1	0.5	639
256	0.1	2.9	1.0	1279
512	0.1	8.0	2.1	2559
1024	0.2	29.5	4.7	5119

Table 9.6. Performance (runtime in secs) of method 3 versus method 1.

9.5 The HDL verifier

An HDL parser, the transformations, and the BDD interpreter are implemented to form one program. Actually, the HDL presented here is only a subset from the one that is implemented; many interesting features have not been discussed. The program expects two HDL files as input, the first will be considered the implementation (imp), the second the specification (spec). This distinction (and thus asymmetry) is necessary to correctly interpret don't care conditions, which will not be discussed here. Information about the correspondence relation between the implementation and specification designs' primary signals may be separately specified in a so-called signal correspondence file.

The designs will be verified by constructing a BDD representation for all signals, starting at the primary inputs, and comparing the corresponding primary outputs for equivalence. Ideally the two designs under test should be processed simultaneously. This is feasible for gate-level designs but not easy to implement for hierarchical designs especially when the levels of abstraction of specification and implementation differ. Currently, we therefore choose to handle the spec and imp design sequentially. First the spec design is traversed in a depth-first manner from primary inputs to primary outputs. Afterwards, the imp design is treated similarly; the same traversal routines are used. While traversing, BDDs are constructed for each bit of a primary input, an internal signal, and a primary output. Initialization takes care of uniquely assigning a fresh BDD variable to each corresponding pair of primary inputs.

After the BDDs are derived for both circuits, the outputs are checked bit for bit. If all of them match, the verification is successful and "Ok" is printed; otherwise the conflicting outputs are reported and for each a test vector is given that when applied to the respective inputs of the circuits exhibits the discrepancy. For example, taking the mapped 4-bit adder (figure 9.14) as implementation and erroneously using a NAND gate for I_8, the program reports:

```
Outputs 'cout' mismatch.
a[0] = 0;
a[2] = 1;
b[2] = 0;
a[3] = 1;
b[3] = 0;
Outputs 's[2]' mismatch.
a[0] = 0;
Outputs 's[3]' mismatch.
a[0] = 0;
a[2] = 1;
b[2] = 0;
```

Besides this, the program offers the following features:

- Showing the true support of outputs (and locals).
- Conversion of a complete circuit to a HDL behavioural module in sum-of-products or a factored form.
- Conversion to espresso input, running espresso, and reinterpreting its output in terms of a HDL behavioural module.

9.6 Dealing with large circuits

A standard technique applied by many CAD-tools when facing a large design is to partition the design into a number of smaller blocks that are dealt with separately. There are several ways to obtain a desired partition. Often the design is described in a hierarchical way and then the natural approach would be to divide the design along the boundaries dictated by the hierarchy. In the absence of a clear coarse-grain structure in the description, one could resort to some suitable partitioning algorithm or simply leave the problem for the designer to solve. The latter, of course, is usually met with obvious reluctance. For a verification tool we have the additional requirement that the partitioning of the two designs under comparison should be consistent in the sense that similar blocks are verified against each other. Not surprisingly, when verifying descriptions of differing levels of abstraction, e.g. a flat gate-level circuit against a high-level arithmetic description, this requirement will be very hard to fulfill. Next we will explain a method to solve this problem in a practically acceptable way.

Whatever method is used to cut the design, the effect will always be the breaking of certain connections and the conceptual introduction of new primary input-output pairs. The spot where a net is broken will be called a cutpoint. More precisely, when we put a cutpoint on a net, the cut will be located right after the driver of that net (see figure 9.17).

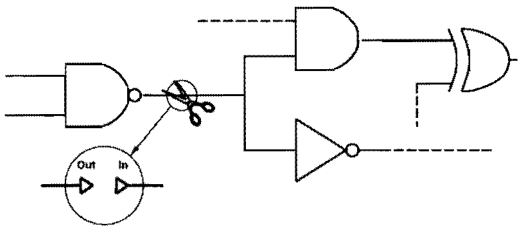


Figure 9.17. Location of a cutpoint and its effect.

In a gate-level design the driver for a net is usually an output port of a gate or a primary input pin. However, breaking nets driven by primary inputs doesn't make much sense. A consistent partition consists of a number of spec cutpoints together with their imp correspondences. Effectuating the partition results in the designs being split up in a number of chunks of logic, such that for each spec block there exists precisely one corresponding imp block and vice versa. The verification problem is thereby greatly simplified: only the corresponding (much smaller) blocks need to be compared. The problem is that we may be confronted with so-called false negatives, i.e., primary outputs are found to miscompare when in fact they are identical. We will return to false negatives in section 9.6.4.

We could hope for a good designer to supply the necessary cutpoints. This seems to be a reasonable assumption when both the spec and imp design are handcrafted; the designer surely will have an intimate knowledge of his design (in particular regarding its structure and functionality) and probably will use the same names for corresponding signals. (In [Burch91] a set of good cutpoints for a class of multiplier circuits is indicated, however no automatic method to derive them is suggested.)

A common complaint about automatic logic synthesis is that it obscures much of the original internal structure, partly because synthesis tools have the habit of messing up the internal net names. So, although a designer might very well be able to come up with a set of useful cutpoints in the spec, he will have problems to figure out their correspondences in the imp. Moreover, the person that verifies the design need not be the designer himself, and hence, has little knowledge about the precise functioning of the circuit and the meaning of the signals.

9.6.1 Cutpoints and BDDs

As mentioned before, we derive BDDs for both designs sequentially; first the spec design is traversed then the imp design. The difference in treatment lies in the nature of certain actions on cutpoint signals that are encountered. Initialization takes care of uniquely assigning a fresh BDD variable to each corresponding pair of primary inputs and to each corresponding pair of cutpoints.

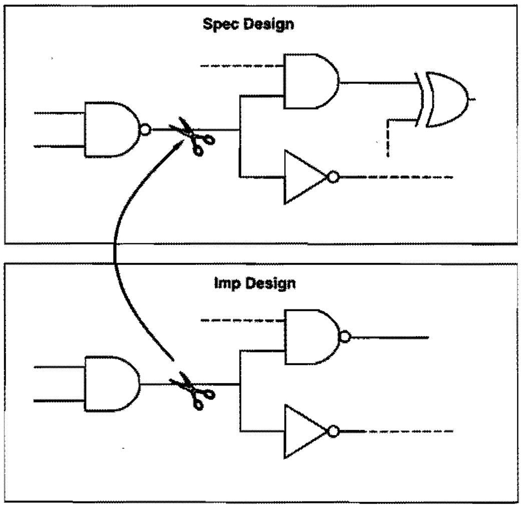


Figure 9.18. Corresponding cutpoints.

The BDDs for the spec will thus be expressed over the combined sets of primary input and cutpoint variables. During traversal of the imp, each cutpoint BDD will be compared against the corresponding spec cutpoint BDD. Figure 9.18 sketches such a situation.

If all corresponding cutpoints are indeed equivalent and also all primary outputs compare, the designs are declared functionally equivalent. With BDDs it becomes trivially easy to take polarity faults into account. So, without further notice, comparisons will be understood to be performed *modulo complementation*. In fact, when a pair of supposedly equivalent cutpoints miscompares due to opposite polarities, we may decide to correct this by complementing the BDD variable for the cutpoint in the imp design, assuming that the signals are not at fault but the specified cutpoint correspondence is erroneous. Actually, there is room for improvement; it is theoretically possible to also detect polarity faults in the primary input correspondences. Polarity faults are a major source of design errors, and it is therefore important to detect and report them explicitly. Another advantage of using BDDs is the possibility to detect dependencies in miscompares and as a result avoid spurious error messages: a miscompare whose difference (exclusive-or of the BDDs associated with the signals being compared) depends on cutpoint variables that themselves proved to be miscompares should be flagged as such.

9.6.2 Hunting for correspondences

Attaching a BDD to every bit of every signal in the spec design enables an on-the-fly signal correspondence hunt during imp design processing. Since BDDs are a canonical representation of Boolean functions and because our BDD package uses sharing of nodes, two signals that carry the same logical function will have identical BDDs. Therefore by simply marking the BDD associated with a spec cutpoint and testing every imp signal BDD for this mark, we may disclose some of the sought-after cutpoint correspondences. For a proper resumption of the verification process, the thus discovered imp cutpoint should of course be effectuated, i.e., the imp signal is cut and the newly created input must be assigned the BDD variable already associated with the spec cutpoint. It is tempting to throw away both the BDDs calculated for the corresponding cutpoints' output sides. Unfortunately, they need to be kept at hand because resolution of false negatives might refer to them. Assuming an intelligent and judicious choice of spec cutpoints and assuming that the spec design description is at a somewhat higher level of abstraction than the imp design description, it still might happen that not all spec cutpoints have a corresponding signal in the imp. It is conceivable that in such a case the imp design BDDs will become unacceptably big and to no avail: the domain of BDD variables for the spec and the imp is no longer the same and hence comparing BDDs becomes senseless. One remedy would be to have the verification program report a list of the discovered correspondences and abort as soon as some imp BDD gets bigger than a certain limit. It is not easy to decide whether the BDD size explodes because of the circuit's inherent complex functionality or because some cutpoint correspondences were missed.

9.6.3 Cutpoint guessing

Now that we have a way of tracking down cutpoint correspondences, we might as well consider automatically generating them in the first place, perhaps in addition to some provided by the designer. The success of cutpoint guessing highly depends on the amount of information we use in our decision to cutpoint a signal. Many heuristics come to mind: if the spec description is hierarchical and each instantiated sub-circuit is of moderate complexity with respect to BDD size, it would make sense to cutpoint all top-level interconnect between the instances; if few clues about the structure of the spec are present, then perhaps random cutpointing might prove successful; one could define an 'importance' measure for signals that reflects the likelihood of a spec signal to be present in the imp design, for instance based on the fan-out count of the signal.

The simple scheme we adopted in our program is based on BDDs. From our point of view the size of the BDDs is the ultimate measure for a successful verification run. We therefore strive to introduce as many cutpoints as necessary to control the BDD sizes. Note, however, that it is fallacious to argue that introducing too many cutpoints doesn't hurt. It does! Take the extreme case where all spec signals are cutpointed. Granted, initially, i.e., close to the primary inputs, we will find many correspondences in the imp, but quickly we will miss a few and from then on no imp signal will ever correspond to a spec signal anymore. Truthfully, we have no definite answer to the cutpointing problem.

In our experiments we use the number of variables in the support of the Boolean function associated with a signal as cutpointing criterion. Also, for practical considerations, we only allow named signals to be cutpointed. In other words, we will not attempt to cutpoint intermediate BDDs that result from evaluating subexpressions in behavioural descriptions. Given the moderate expressive power of the behavioural block descriptions (e.g. there is no multiplication operator defined) and some observations of designers' practice (most expressions tend to fit within an 80 character line; local variables are used abundantly) this seems not to be a serious restriction.

9.6.4 Resolving false negatives

The sole purpose of cutpointing is to introduce new (virtual) primary inputs and thereby increasing the chances that the BDDs constructed during verification are kept reasonably small. Our method for automatic cutpoint generation described above, will guarantee small BDDs for all spec design signals. But clearly the cutpoint inputs are not real primary inputs: they are names that stand for Boolean functions over the real primary inputs. In that sense, a cutpoint BDD variable may not be treated as an independent variable. A comparison between two signals should therefore take this interdependency into account: if the exclusive-or of the BDDs is the 0 BDD, the signals are truly equivalent; otherwise the non-0 difference needs further investigation: it's a potential false negative. This is resolved by resubstituting the functions for the cutpoint variables into the non-0 difference, the penalty being a possible increase in the size of the BDD for the difference. It is obvious that we should treat the cutpoint variables in reverse topological order. Often, not all cutpoint variables need be considered; if indeed the signals are equivalent, usually only a small number of resubstitutions (say less than 10) are needed to establish the falsehood of the miscompare. On the other hand, when the signals do actually miscompare this can only be decided after resubstituting all the cutpoint variables present in the support of the difference. Because of this behaviour of false negatives it is a good idea to have a program option to turn off the resolution of false negatives.

9.6.5 Experiments

Several experiments have been performed to validate the ideas expressed above. They were all run on a 99MHz HP9000/735 machine. The circuits are characterized in table table 9.7. "PI", "PO", and "REG" respectively, are the number of primary input, primary output, and register bits; the "INT" columns list the number of internal signals. The first 9 circuits are the toughest (according to Rudell) in the ISCAS'85 benchmark set. The 5 others are industrial designs. The "IMPs" for the ISCAS circuits are the non-redundant ones known under suffix "nr". Note that the ISCAS circuits are at gate-level, and spec and imp are very close in terms of internal signals, unlike the industry circuits that clearly have a high-level spec and a low-level imp. For all tables below with the exception of table 9.10, if a circuit is not listed this means that it ran out of memory (> 120Mb). Table 9.8 and table 9.9 are included for reference: they indicate which circuits can be handled without cutpoints, both with (+DVO) and without using dynamic variable ordering (-DVO). Although +DVO definitely strengthens BDD-based verification (at the expense of an increase in runtime for most of the circuits), the notorious 16-bit multiplier c6288 and the larger industrial designs cannot be coped with.

Spec cutpoints were available for the industrial designs (none were necessary for alu32) and their effectiveness is indicated in table 9.10. It is interesting to report that often only a few signals were specified; the number of cuts is a bit-count, whereas the spec designs use internals signals declared as bitvectors. Table 9.11 lists the results when a number of "(Gen)" cutpoints are generated automatically (which takes time as listed in the last column). All ISCAS benchmarks including c6288 are easily handled. No good cutpoints could be automatically found for the industrial designs alu16 and idct.

Circuit	PI	PO	REG	INT SPEC	INT IMP
c432	36	7	0	153	150
c499	41	32	0	170	170
c1355	41	32	0	514	514
c1908	33	25	0	855	853
c2670	233	139	0	1130	898
c3540	50	22	0	1647	1598
c5315	178	123	0	2184	2175
c6288	32	32	0	2384	2367
c7552	207	108	0	3405	3290
mul9	18	18	0	375	648
mul16	32	32	0	955	2056
alu16	77	40	1	452	1178
alu32	72	34	0	763	1179
idct	38	40	2674	6089	26098

Table 9.7. Some characteristics of the test circuits.

Circuit	Peak BDD	Peak Mem (Kb)	Time (s)
c432	7865	407	0.6
c499	74885	2303	5.7
c1355	199158	5776	10.8
c1908	157956	4866	19.8
c3540	565476	15607	31.4
mul9	234341	6622	24.0
alu32	249635	7074	3:16.6

Table 9.8. Classical method: no cutpoints; -DVO.

Circuit	Peak BDD	Peak Mem (Kb)	Time (s)
c432	5934	414	1.4
c499	48235	1452	29.2
c1355	130037	4280	1:16.3
c1908	32714	1066	22.4
c2670	22278	839	23.7
c3540	192268	7093	1:39.1
c5315	11109	560	16.8
c7552	190653	5626	2:15.2
mul9	96046	2647	48.9
alu32	20690	772	14.0

Table 9.9. Classical method: no cutpoints; +DVO.

Circuit	Cuts	Peak BDD	Peak Mem (Kb)	Time (s)
mul9	50	32012	1138	20.8
mul16	136	28927	982	3:09.2
alu16	35	4865	345	9.5
idct	103	385258	41945	5:24.4

Table 9.10. Designer supplied only Spec cutpoints; +DVO.

Circuit	Cuts (Gen)	Peak BDD	Peak Mem (Kb)	Time (s) (Gen)
c432	11 (13)	4097	338	0.4 (0.2)
c499	16 (24)	797	271	0.3 (0.3)
c1355	32 (40)	2085	272	0.7 (0.7)
c1908	57 (66)	4553	344	2.3 (2.1)
c2670	86 (135)	286915	8861	53.8 (7.0)
c3540	143 (185)	28326	968	27.0 (4.8)
c5315	239 (301)	9626	511	27.8 (14.1)
c6288	479 (479)	8824	495	8.9 (7.1)
c7552	407 (469)	13959	648	20.3 (9.6)
mul9	0 (48)	96046	2647	48.9 (58.3)
alu16	70 (116)	33662	1167	36.6 (15.7)
alu32	66 (164)	10286	488	11.3 (29.2)

Table 9.11. Automatic cutpoint generation; +DVO.

Chapter 10

The PTL Program

10.1 Introduction

In this chapter three problems are studied and solved with the use of the ptl program. The heart of this program is a satisfiability checker for propositional linear-time temporal logic as presented in chapter 6. Some other features of the program will briefly be mentioned. The problems under study are a simple elevator, a logic game, and a synchronous bus arbiter.

10.2 A 2-story elevator

An elevator moves up and down between two floors. There is a push button on each floor that initiates a request for the elevator to come to that floor, pick up the person, and move to the other floor (figure 10.1).

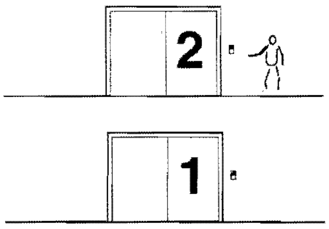


Figure 10.1. A 2-story elevator.

Stepping in and out, and also the opening and closing of doors is not taken into consideration. Actually, there will be no notion of a person riding the elevator.

We like to specify this system in LTL and then use this specification to derive some interesting properties of the elevator system and eventually prove that a simple controller conforms with it. (Although this elevator system looks almost too simple to merit any real-life application, there does actually exist such an elevator in the lobby of the EE building of Eindhoven University.)

The system is modelled using the following internal state variables:

V_1:	elevator is on the first floor,
V_2:	elevator is on the second floor,
O_1:	a request to bring the elevator to the first floor and then carry a person up to the second floor is pending,
O_2:	a request to bring the elevator to the second floor and then carry a person down to the first floor is pending.

And the external control variables and their intended meaning are:

K_1:	someone pushes the button on the first floor,
K_2:	someone pushes the button on the second floor.

Obviously, a good specification at least ensures that when a button is pressed a request is initiated and that a request will cause the elevator to perform a particular movement. To keep the specification as general as possible we shall avoid fixing any time delays between the occurrence of certain events: we determine the order of events without telling exactly when they happen. But keep in mind that any specification is by definition of a subjective nature and one cannot argue whether a specification is correct in itself. The least we can do is to make sure a specification is not self-contradictory which would render it useless.

Beginning of specification.

This first part of the specification states conditions that must hold at any time, hence the "always" (also called henceforth) temporal operator:
$\square$ (

The elevator is either on the first floor or on the second floor:
$(V_1 \leftrightarrow V_2')$
Remark: we could have used only 1 variable to denote the position of the elevator because in this case we only have 2 (binary) floors; in general this does not hold.

When there are no requests, the elevator does not move:
$(O_1' O_2' V_1 \rightarrow \circ V_1)$
$(O_1' O_2' V_2 \rightarrow \circ V_2)$
Remark: no superfluous movement.

Next come two requirements that capture the operational behaviour of the elevator.

A request will stand till and including the moment of arrival of the elevator on that floor, which is bound to happen sometime:
$(O_1V_2 \rightarrow \circ\ ((O_1V_2)\ \mathsf{U}\ (O_1V_1)))$
$(O_2V_1 \rightarrow \circ\ ((O_2V_1)\ \mathsf{U}\ (O_2V_2)))$
Remark: pushing the button calls the elevator to your floor. We must retain the request since the movement of the elevator is not completed yet.

When there is a request on a floor and the elevator happens to be on that floor, it need only move to the other floor and when arriving there the request may or may not extinguish depending on whether the button is pushed again:
$(O_1V_1 \rightarrow \circ\ ((O_1V_1)\ \mathsf{U}\ V_2))$
$(O_2V_2 \rightarrow \circ\ ((O_2V_2)\ \mathsf{U}\ V_1))$
Remark: step in the elevator and it brings you to the other floor.

The request is reset upon completion of the elevator's task:
$(O_1V_1 \circ V_2 \rightarrow \circ\ (O_1'\ \mathsf{U}_w\ K_1))$
$(O_2V_2 \circ V_1 \rightarrow \circ\ (O_2'\ \mathsf{U}_w\ K_2))$
Remark: the premiss states the completion of the elevator's task, so it makes sense to immediately clear the request at least (weak-)until a button is pressed again in order to service a next customer.

Pressing a button immediately causes a request:
$(K_1 \rightarrow O_1)$
$(K_2 \rightarrow O_2)$
Remark: this is a matter of taste: we like to have instant service.

This concludes the first part of the specification:
)

Now we state what must hold initially:

In the beginning there will be no requests (weak-)until a button is pressed:
$(O_1'\ \mathsf{U}_w\ K_1)$
$(O_2'\ \mathsf{U}_w\ K_2)$
Remark: this ensures that requests cannot occur spontaneously.

The initial state of the system:

Remark: not specified, i.e., the elevator is either on the first or on the second floor.

End of specification.

Figure 10.2 summarizes the above specification in the LTL syntax as typed on a computer.

```
[] (
                                (V1 <-> V2')
(O1' O2' V1 -> @V1)                    (O1' O2' V2 -> @V2)
(O1 V2 -> @((O1 V2) U (O1 V1)))        (O2 V1 -> @((O2 V1) U (O2 V2)))
(O1 V1 -> @((O1 V1) U V2))             (O2 V2 -> @((O2 V2) U V1))
(O1 V1 @V2 -> @(O1' Uw K1))            (O2 V2 @V1 -> @(O2' Uw K2))
(K1 -> O1)                             (K2 -> O2)
)
(O1' Uw K1)                            (O2' Uw K2)
```

Figure 10.2. Elevator spec in ptl syntax. Note the symmetry between the clauses for both floors.

We will subject the elevator specification to a number of tests. Unless stated otherwise, in each case the description as in figure 10.2 is assumed to be logically AND-ed with a description of the external behaviour of the system; in other words the environment consisting of persons pushings the buttons on both floors. This complete description is then checked to logically imply one or more consequences. Each case starts with a informal statement of the test.

Test 0: The specification is satisfiable, i.e., not self-contradictory.

Test 1: From the fact that no button is ever pressed it follows that no requests will ever occur and consequently the elevator never moves:
$\Box K_1' \Box K_2' \rightarrow \Box O_1' \Box O_2' \, (\Box V_1 \vee \Box V_2)$

Test 2: A button press on floor 1 happening once implies that the elevator must eventually end up on the second floor and stay there:
$(K_1' \; \mathrm{U} \; (K_1 \circ \Box K_1')) \Box K_2' \rightarrow V_2 \; \mathrm{U} \; (V_1 \circ (V_1 \; \mathrm{U} \; \Box V_2))$

Test 3: When both buttons are once pressed simultaneously, the elevator makes a full move, i.e., either down and up again, or up and down again:
$$\begin{array}{l} (K_1' K_2') \; \mathrm{U} \; (K_1 K_2 \circ \Box (K_1' K_2')) \\ \rightarrow \\ (\quad V_1 \circ (V_1 \; \mathrm{U} \; (V_2 \circ (V_2 \; \mathrm{U} \; \Box V_1))) \\ \;\mathrm{xor}\, V_2 \circ (V_2 \; \mathrm{U} \; (V_1 \circ (V_1 \; \mathrm{U} \; \Box V_2)))) \end{array}$$

Test 4: When button 1 is pressed infinitely often, the elevator must move infinitely often too:
$\Box \Diamond K_1 \; \Box K_2' \rightarrow \Box \Diamond (V_1 \circ (V_1 \; \mathrm{U} \; (V_2 \circ (V_2 \; \mathrm{U} \; V_1))))$

Test 5: Every request is honoured:
$\Box \, (O_1 \rightarrow \Diamond (V_1 \circ V_2)) \, \Box \, (O_2 \rightarrow \Diamond (V_2 \circ V_1))$
In this case the spec is supposed to imply (→) this formula.

Test 6: It is possible that pressing a button more than once causes less movements of the elevator; in other words some calls get lost. In fact we press button 1 twice and then assert that the elevator makes only 1 movement:
$K_1' \; \mathrm{U} \; (K_1 \circ (K_1' \; \mathrm{U} \; (K_1 \circ \Box K_1'))) \Box K_2' \rightarrow V_2 \; \mathrm{U} \; (V_1 \circ (V_1 \; \mathrm{U} \; \Box V_2))$

Clearly, this merely needs to be tested for satisfiability.

The results of running the ptl program on the seven test cases are collected in table 10.1 below.

Test	Result	Time (secs)
0	True	< 0.1
1	True	< 0.1
2	True	< 0.1
3	True	< 0.1
4	True	< 0.1
5	True	0.4
6	True	0.2
Spec ← FSM	True	< 0.1

Table 10.1. ptl results and runtimes (HP9000/755, 76 MIPS).

It seems that the given specification pretty much captures our informal concept of how the simple 2-story elevator is supposed to behave. It would be more interesting to see whether a proposed controller circuit complies with the formal specification. Here we shall describe the controller by a Moore-type state machine and prove that its behaviour is contained in (is a subset of) the specification. To make things easier, we first suggest a simplification to the specification: note that the request variables O_1 and O_2 are fully determined by the button variables K_1, K_2 and the elevator position variables V_1 and V_2, thus they are superfluous. The specification without explicitly mentioning requests is stated in figure 10.3. Of course, we can automatically check that the old spec of figure 10.2 indeed implies the new one (and not the other way around). This means that any behaviour that makes the old spec true also makes the new spec true.

```
[](
(V1 <-> V2')
(K1  -> V2 U (V1 @(V1 U V2)))
(K2  -> V1 U (V2 @(V2 U V1)))
)
((V1 Uw (V1 (K1 V K2))) V (V2 Uw (V2 (K1 V K2))))
```

Figure 10.3. Revised elevator spec in plt syntax.

A state diagram of the proposed controller is depicted in figure 10.4.

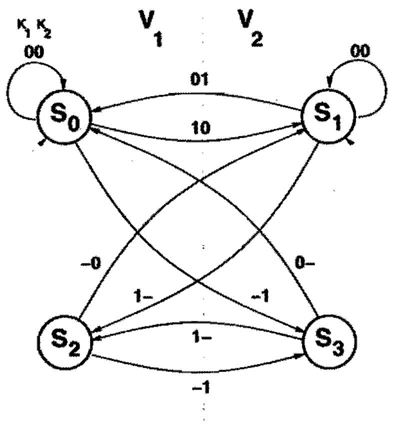


Figure 10.4. Elevator controller state diagram. The dashed line indicates the symmetry in the diagram with respect to initial states and output function.

An ptl description of the diagram is given in figure 10.5. The results of testing whether this description implies the new spec is also mentioned in table 10.1. The implication does not hold the other way around, i.e., the descriptions are not equivalent.

```
[](
/* Read: next state will be Si iff now ... or now ... */
(@S0 <-> (S0 K1' K2') V (S1 K1' K2 ) V (S3 K1'))
(@S1 <-> (S0 K1  K2') V (S1 K1' K2') V (S2 K2'))
(@S2 <-> (S1 K1) V (S3 K1))
(@S3 <-> (S0 K2) V (S2 K2))
)

/* Initial state: */
( S0  S1' S2' S3' V S0' S1  S2' S3' )

/* Output function: */
[](V1 <-> S0 V S2)  [](V2 <-> S1 V S3)
```

Figure 10.5. Moore machine for elevator controller in ptl.

More eleborate LTL models of elevator designs can be found in [WoodW89] and [HaleR87].

10.3 Chinese ring puzzle

Here we look at a variant of the well-known Chinese Ring puzzle [Keist]. We will show how the puzzle can be cast as a reachability problem on a state-space and use the ptl program to solve it.

A jewelry box has a lock with 7 binary dials or knobs; each knob is in one of two possible positions: open (coded as 0 or false) or closed (coded as 1 or true). The knobs are numbered from left to right, see figure 10.6 that shows the front panel.

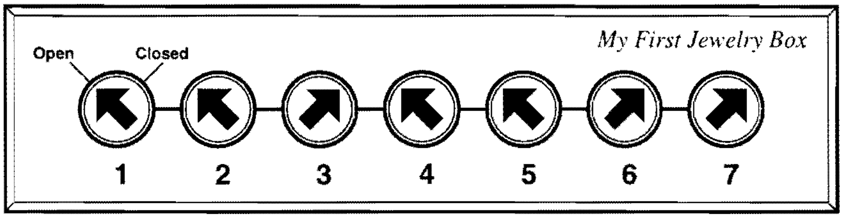


Figure 10.6. Jewelry box lock control panel.

Due to the mechanics of the lock, the knobs cannot be turned independently:

1. Knob number 1 (the most left knob) can always be turned.
2. If one doesn't choose to turn the first knob then the only knob that can be turned is the one directly following the first closed knob as seen from the left. (This is knob 4 in figure 10.6).
3. If the last knob is the only one in the closed position then 2) does not apply, and the only choice left is to turn the first knob.

The objective is to open the box, i.e., setting all knobs to their open position. Initially, all knobs are in their closed position. (The fastest solution for the instance with 7 knobs takes 85 turns.)

The puzzle may be described by a finite automaton over a 2 symbol alphabet and states set $S_0, \cdots, S_{127}$, or equivalently by a Moore type sequential machine with the state transition table of table 10.2. Its single output, `Open`, is true in state `0b0000000` and false in all other states.

first	K[1:7]	@K[1:7]
1	0b-------	K xor 0b1000000
0	0b1------	K xor 0b0100000
0	0b01-----	K xor 0b0010000
0	0b001----	K xor 0b0001000
0	0b0001---	K xor 0b0000100
0	0b00001--	K xor 0b0000010
0	0b000001-	K xor 0b0000001
0	0b000000-	K

Table 10.2. Knob settings as function of whether or not `first` knob is turned.

The above state transition table can be automatically converted to an LTL formula. This, together with the initial state and the output function, gives the description of figure 10.7.

```
[](
(@K[1] <-> first  K[1]' V first' K[1])
(@K[2] <-> first  K[2]  V first' K[1] K[2]' V K[1]' K[2])
(@K[3] <-> first  K[3]  V K[3] K[1] V K[3] K[2]'
        V  first' K[3]' K[1]' K[2])
(@K[4] <-> first K[4] V K[4] K[1] V K[4] K[2]
        V  K[4] K[3]' V first' K[4]' K[3] K[1]' K[2]')
(@K[5] <-> first K[5] V K[5] K[3] V K[5] K[1]
        V  K[5] K[2] V K[5] K[4]'
        V  first' K[5]' K[4] K[3]' K[1]' K[2]')
(@K[6] <-> first K[6] V K[6] K[4] V K[6] K[3]
        V  K[6] K[1] V K[6] K[2] V K[6] K[5]'
        V  first' K[6]' K[5] K[4]' K[3]' K[1]' K[2]')
(@K[7] <-> first K[7] V K[7] K[5] V K[7] K[4]
        V  K[7] K[3] V K[7] K[1] V K[7] K[2]
        V  K[7] K[6]'
        V  first' K[7]' K[6] K[5]' K[4]' K[3]' K[1]' K[2]')
)

K[1] K[2] K[3] K[4] K[5] K[6] K[7]

[](Open <-> K[1]' K[2]' K[3]' K[4]' K[5]' K[6]' K[7]')
```

Figure 10.7. Puzzle described as ptl formula.

To obtain a solution to the puzzle we use a special option of the ptl program that will cause a valid model to be generated in a format suitable for a wave-form viewer program [Buurm]. If we merely ask for a model that satisfies the above LTL description then obviously it is not guaranteed that it will contain the 'open' state. We force a model with a solution by AND-ing the formula with `<>Open` and then asking for a satisfying model. The results are shown in figure 10.8. We see that by alternatingly choosing to turn the first knob and to turn the only other possibility (case 2), a solution is obtained after 85 turns, which happens to be the fastest way to arrive at one. In general it is not guaranteed that the ptl program finds the 'smallest' model, but heuristics are included to aim at it.

10.4 Synchronous bus arbiter

The next example is taken from [McMil93]. We will first give an informal specification of the arbiter, then translate that into a behavioural module of the HDL introduced in chapter 9, and finally a structural implementation is suggested.

The purpose of the arbiter is to exclusively acknowledge one of the requests for some shared resource. We will assume synchronous behaviour, i.e., requests and acknowledgements are to be observed at discrete points in time imposed by a global clock. We consider an n-input, n-output arbiter ($n \geq 1$): the request inputs are req[0], req[1], $\cdots$, req[n-1], the ackowledge outputs are ack[0], ack[1], $\cdots$,

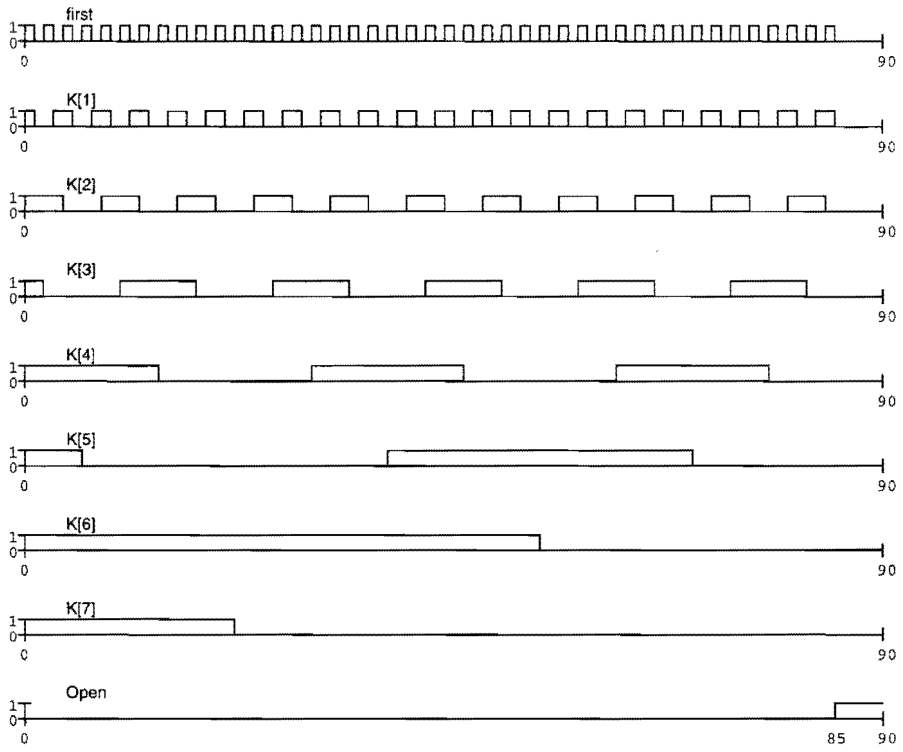


Figure 10.8. Timing diagrams showing a solution of the puzzle.

ack[n-1]. See also figure 10.9. The rules by which the arbiter operates are:

Rule 1: In case of a single request, that request is acknowledged immediately.

Rule 2: In case of multiple requests, the one with lowest index is acknowledged immediately.

To avoid a low indexed request to continuously take priority over a higher indexed one (starvation), the next rule is added:

Rule 3: Persistent requests will be served on a round-robin basis.

In other words, a request is acknowledged if there are no requests of higher priority (= with lower index) and there is no persistent request in need of service other than the request itself. So far we haven't exactly defined what persistence entails. Of course, there is no unique definition. In [McMil93] the choice is made to call a request persistent if it is raised for a duration of at least n clock cycles. More precisely, McMillan's arbiter uses a shift-register (T in figure 10.10) in which a single token (a 1 bit) is circulated one position ahead per clock cycle; a request that cannot be directly acknowledged but 'holds' the token is guaranteed to be acknowledged n clock cycles from now, i.e., when the token reappears. In the worst case this means that a request has to persist for $n-1$ cycles to get the

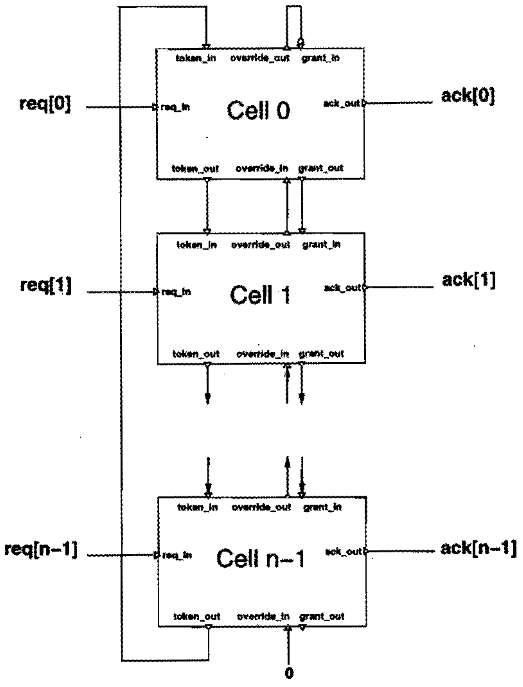


Figure 10.9. An n-input/n-output arbiter.

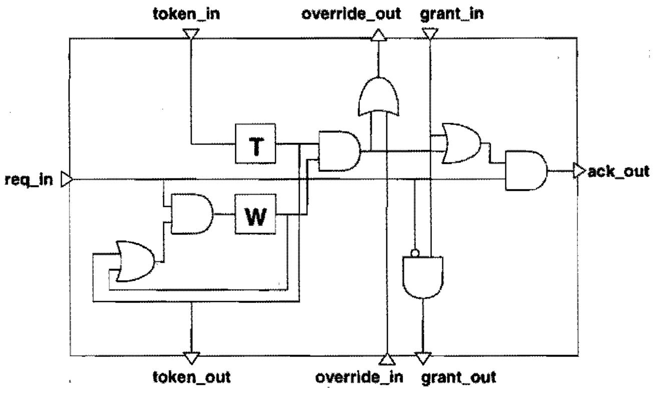


Figure 10.10. A single cell of the arbiter circuit.

token plus another n cycles to finally get acknowledged. In an implementation we therefore need to remember the fact that a request coincides with the presence of the token. For this purpose the register W[i] is introduced: it will be set in the cycle following the concurrence of a request and the token, and remains set as long as the request is raised. The predicate persists indicates the cycle in which the persistent request may be acknowledged. McMillan's arbiter may now be formally defined by (using the temporal next-time operator @ to refer to the next clock cycle):

$\forall_{0 \le i < n}$ @T[(i + 1)%n] = T[i] (where % denotes *modulo*), and

$\forall_{0 \le i < n}$ @W[i] = req[i] $\wedge$ (W[i] $\vee$ T[i]), and

$\forall_{0 \le i < n}$ persists (i) = W[i] $\wedge$ T[i], and

$\forall_{0 \le i < n}$ ack[i] = req[i] $\wedge$ [($\forall_{0 \le j < i}$ $\neg$ req[j]) $\wedge$ ($\forall_{0 \le j \ne i < n}$ $\neg$ persists (j)) $\vee$ persists (i)]

This 'high-level' definition can be shown to be equivalent to the circuit depicted in figure 10.9, which is composed of n cells of the kind drawn in figure 10.10. For a 4 cell version, it takes the ptl program just a few seconds to do so. More interesting is the observation that McMillan's circuit allows situations where requests are present although none is immediately acknowledged. We use the ptl program to find out whether the following property is implied by the (4-cell) circuit:

```
[](    (req[0] V req[1] V req[2] V req[3])
   -> (ack[0] V ack[1] V ack[2] V ack[3]))
```

It is not! Using the counter-example option we obtain the timing diagrams of figure 10.11. The reason for this behaviour lies in the definition of persists: clearly it may happen that persists (1) holds, but at the same time the request req[1] is absent but an another one, viz. req[2], is present. It seems more natural to let persists also depend on the request signal. Therefore we suggest the following modification:

$\forall_{0 \le i < n}$ persists (i) = req[i] $\wedge$ W[i] $\wedge$ T[i], and therefore

$\forall_{0 \le i < n}$ ack[i] = req[i] $\wedge$ ($\forall_{0 \le j < i}$ $\neg$ req[j]) $\wedge$ ($\forall_{0 \le j \ne i < n}$ $\neg$ persists (j)) $\vee$ persists (i)

Using $\neg$ req[i] $\Rightarrow$ $\neg$ persists (i) the latter simplifies to:

$\forall_{0 \le i < n}$ ack[i] = req[i] $\wedge$ ($\forall_{0 \le j < i}$ $\neg$ req[j]) $\wedge$ ($\forall_{i < j < n}$ $\neg$ persists (j)) $\vee$ persists (i)

Figure 10.12 shows a straightforward implementation (without the registers) of the modified arbiter specification in our sample HDL. For clarity, the persists predicate is separately defined; presupposing the existence of a macro preprocessor.

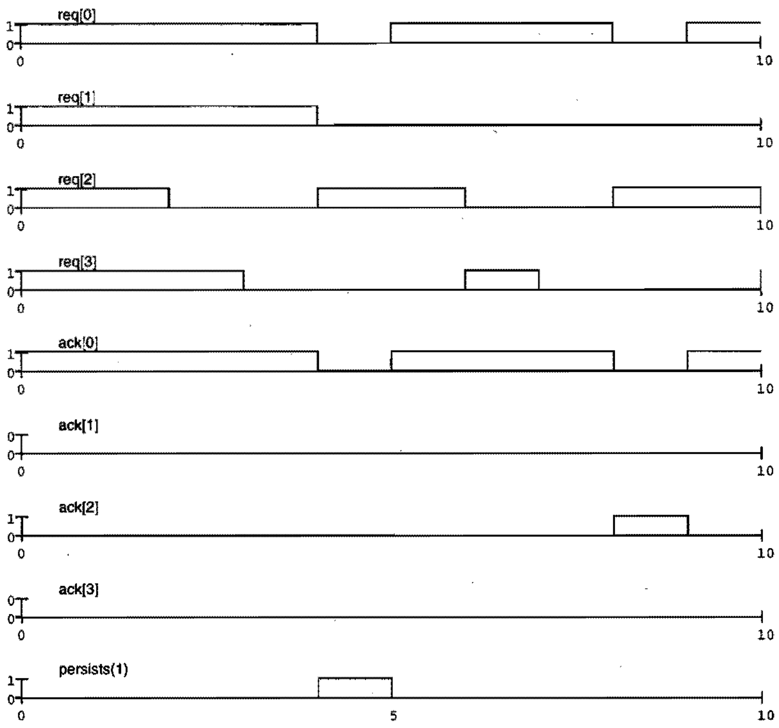


Figure 10.11. No ack signal in the 5th cycle but req[2] is high.

```
#define persists(i)(req[i] & W[i] & T[i])

behaviour arbiter[[n]](input  req[n-1:0], T[n-1:0], W[n-1:0];
                       output @T[n-1:0], @W[n-1:0], ack[n-1:0])
{
  (i:0..n-1) {
    @T[(i+1) % n] = T[i];

    @W[i] = req[i] & (W[i] | T[i]);

    ack[i] = req[i] & (& (j:  0..i-1) ^req[j])
                    & (& (j:i+1..n-1) ^persists(j))
           | persists(i);
  }
}
```

Figure 10.12. Implementation of the arbiter in our HDL.

We can implement our proposed new specification at the logic level by modifying the AND gate that combines the T and W register outputs to include req_in as a third input. Of course, the ptl program was used to verify the correctness of this change.

McMillan suggests to verify whether the following properties hold for the (properly initialized) arbiter:

1. No two acknowledge outputs are asserted simultaneously, or equivalently, at any time at most 1 acknowledge is asserted. This is a typical safety property.
2. Every persistent request is eventually acknowledged, i.e., starvation is excluded. This is a liveness property.
3. Acknowledge is not asserted without request. This again may be seen as a safety property.

McMillan uses his CTL model checker SMV to automatically verify them. The desired properties may as well be formulated in LTL:

1. $[\,](\exists!_{0\le i<n}\ \texttt{ack[i]}\ \vee\ \forall_{0\le i<n}\ \texttt{ack[i]'})$
2. $\forall_{0\le i<n}\ \texttt{[] <> (req[i] -> ack[i])}$
3. $\forall_{0\le i<n}\ \texttt{[] (ack[i] -> req[i])}$

The last requirement is trivially true; one glance at the logic of a cell suffices. The first requirement is of course supposed to be certified by the priority and round-robin token scheme, but still it is not quite obvious. The second requirement turns out to be the hardest to verify. The phrasing of this second property might be questioned: it doesn't seem to naturally follow from the informal description given above. This might be considered a genuine and justifiable reason for objecting the use of temporal logics in hardware verification: often the clear informal intentions turn into incomprehensible formulas. In this case, given the notion of the persists predicate, it would be more obvious to restate property 2 as:

$\forall_{0\le i<n}\ \texttt{[] (persists(i) -> <>ack[i]).}$

Literally spelled out: always, if a request persists it will eventually be acknowledged. Unfortunatly, with the original definition of persists this cannot be fulfilled. With our new definition however the property holds for both the original circuit and the modified one.

10.4.1 Reachability analysis

Our current implementation of the ptl program constructs its model graph explicitly. For an n-state sequential machine this means that the model graph will have n vertices. To have a fair comparison with McMillan's results as reported in [McMil93], we use the reachable state-space algorithm of chapter 5 which is available as a predefined term in the μ-calculus program. The results reported in table 10.3 are achieved with the use of our BDD package with dynamic variable ordering switched on. (The times are measured on a HP9000/s755 workstation).

The number of BDD nodes representing the set of initial states S_0 and also the number of nodes of the reachable states set (Reachability) are the theoretical lower bounds (and upper bounds, because they are symmetrical functions). McMillan proves that the runtime progresses as the square of n (given a good variable ordering). Our results confirm this. For large n ($n > 28$) the performance detoriates probably because dynamic variable ordering destroys the initially good ordering and too much memory is needed to construct the next-state relation.

n	#states	Tot	S_0		Next-state		Reachability	
	$(n \cdot 2^n)$	sec	msec	#nodes	msec	#nodes	msec	#nodes
8	2048	1.5	20	24	1480	60	50	15
9	4608	2.0	40	27	1920	65	60	17
10	10240	2.6	60	30	2480	75	100	19
11	22528	3.2	60	33	3000	70	160	21
12	49152	4.9	70	36	4680	90	170	23
13	106496	7.3	80	39	6970	95	250	25
14	229376	10.6	110	42	10120	105	430	27
15	491520	9.8	110	45	9370	110	340	29
16	1048576	12.5	140	48	11940	80	450	31
24	$402 \cdot 10^6$	52.7	310	72	41640	120	10810	47
28	$7 \cdot 10^9$	66.2	390	84	59930	285	6100	55
32	$137 \cdot 10^9$	-	480	96	Killed >120M		-	-

Table 10.3. Results of arbiter's reachable states calculation.

10.5 Discussion

This chapter intended to illustrate the practical use of temporal logic in describing digital hardware and proving its correctness. We also showed the possibilities of a prototype LTL satisfiability checker program to aid in the analysis of specifications and implementations, and to eventually derive 'better' circuits. We like to stress that verification should not only be considered as an 'after-the-fact' means for approval of a design. Indeed, such a viewpoint has its merits, but it also underestimates the capabilities verification tools can offer during the design process. With the growing complexity of the designs we believe that particularly this role of verification, i.e., as a companion to synthesis, should get more attention and ultimately might find its niche in standard design methodologies.

In retrospect, we feel that it is precisely the latter application that is most suitable for temporal logics based verification tools. More often than not, manual intervention is required to get a certain property tested, simply because many simplifications and optimizations cannot (yet) be automatically detected and derived by the tool.

Chapter 11

Conclusions

11.1 Contributions and achievements

The research goal set forward at the start of this thesis work was to investigate various logics with respect to their application to the functional verification of digital circuits. We have approached this goal by studying some typical verification problems for combinational and sequential circuits. The main contribution of this thesis may be formulated as the development of practical software tools for several verification problems by using mathematical models and the presentation of the theory for reasoning about those models. This approach is reflected in the structure of this thesis: part I discusses verification problems in the area of combinational circuits; part II focuses on sequential circuit verification, and part III presents practical examples of systems that have been developed and discusses details of the implementations. Also, part III examines a number of test cases that exhibit the typical modelling of problems in terms of the investigated logics and shows how they are solved by the presented tools.

Part I and II share a similar structure: firstly, the type of circuit we are dealing with is explained and several verification problems are formulated; secondly, the syntax and semantics of a logic that is particularly well-suited to express these verification problems are introduced; and, thirdly, algorithms to automate the reasoning process are derived. Our contribution has been to exemplify the steps necessary to formalize a logic in terms of its syntax and semantics, to express certain verification related decision problems in that logic, and lastly, to show how such problems are solved by means of efficient algorithms. We hope that our engineering oriented approach, in contrast to a more mathematically inclined approach, makes the subject matter more easily comprehensible.

The following discussion makes the contributions of this dissertation more explicit.

- The general problem of combinational circuit verification as addressed in chapter 2 led to some interesting results for permutator circuits. We claim that the size of a BDD for such circuits is of the order $n \cdot {}^2\log n$, where n is the number of single bit input signals. We also show how such a circuit can be designed for any value of $n \geq 1$, not just for n that are powers of 2. The design is based on the structure of a sorting (butterfly) network.
- The algorithm that converts a propositional formula into its disjunctive normal form, as presented in chapter 3, is well-known. Finding a minimum DNF representation is an NP-hard problem. In our implementation we optimize the result by a simple containment check, i.e., conjunctions (cubes) that are implied by others are removed. Although this would normally require an $O(n^2)$ algorithm, we use a modified merge-sort ($O(n \cdot {}^2\log n)$) algorithm that in practice turns out to give very good results. This DNF conversion algorithm is used in the LTL satisfiability checker described in chapter 6.
- The BDD package that is discussed in chapter 4 and chapter 8 is generally recognized as one of the better engineered packages available today. In a recent comparative study conducted by Bwolen Yang [YangB98] the Eindhoven BDD package performed very satisfactorily. The package is part of Philips' YATC verification tool and several IBM proprietary tools such as Verity [Küehl95] and the BSN suite of verification tools. As mentioned before, the BDD package forms the heart of the verification programs discussed in this thesis.
- The observation that strongly connected components play a crucial role in checking satisfiability of LTL formulas is already made in [Venka87]. We believe that theorem 6.1 (chapter 6) that explicitly emphasizes this role is quite new. Also, our definition of elementary formulas and the treatment of eventualities deviates from other approaches and can be shown to lead to a more efficient implementation. Venkatesh proposed to replace eventualities by auxiliary propositional variables and then add additional terms to the formula. Clearly, this would increase the number of propositional variables and hence exponentially increases the number of states in the model graph.
- In section 6.8 we show how various types of finite state machines can be expressed in LTL. This is valuable knowledge for any engineer using an LTL satisfiability checker. It also formed the basis for a algorithm that we developed to automatically generate an LTL formula from a given hardware description in the BSN language.
- The treatise on μ-calculus in chapter 7 serves to show how a rather powerful formal system can be reduced to a small set of primitive operations (namely, propositional logic with a least-fixed point operator). The μ-interpreter that

we sketch, is directly derived from its formal semantics, and therefore its correctness may be said to be achieved by construction.

- Chapter 9 discusses the development of a combinational circuit equivalence checker based on a modern high-level hardware description language. This work is derived from IBM's BSN project for which a similar tool was developed around 1992. The application of BDDs in such a program was rather novel at that time. We believe the concept of single-sided cutpoints and the automation of cutpoint generation to be new as well.

It should not be surprising that a major effort of the thesis work went into the development of the programs discussed in part III. Unfortunately there is little room in a thesis to stress the importance of the availability of reliable and efficient software tools for verification. Most of the tools have been made available in the public domain. Hence, numerous people both in industry and academia have obtained copies. This has provided us with much feedback which resulted in many bug fixes and improvements.

11.2 Directions for future research

In this thesis we have addressed three major verification problems: combinational circuit equivalence, also known as boolean equivalence, sequential circuit equivalence, and model/property checking. These problems are all well-understood and have obvious applications in digital circuit verification. Many researchers have studied these problems and quite a number of tools have been developed. Judged by the optimistic press releases of companies that offer formal verification tools, one might get the impression that all is solved and further research be futile. The current state of the art can be roughly characterized by the following data:

- Boolean equivalence checking is successfully applied by all major companies in the electronics and semiconductor industry as a standard step in their chip design flow. Designs up to 50,000 gates can typically be handled. Larger designs can be handled provided that the design is sufficiently partitioned. Some verifiers are able to exploit the hierarchy present in a design.

- In industry there seems to be much less need for equivalence checking of sequential circuits. In most design practices it is traditional to consider large caches and (off-chip) memories as separate entities, even from a simulation point of view. Moreover, the locations of registers, flip-flops, and latches are often frozen at an early stage. Hence, the sequential verification problem reduces to a combinational one. The role of sequential verification is confined to relatively small control-dominated designs. Tools in this area can typically handle up to a couple of hundred single-bit memory elements.

- Model checking has as yet not found a wide-spread use in industry. Several companies are active in this field, but this work is still much research related.

> Of the tools proposed to solve any of the three problems we consider here, a model checking tool will undoubtedly have the steepest learning curve for a designer. This is largely due to the immaturity of current user-interfaces and the unfamiliarity with formal methods and temporal logics. Model checkers are typically deployed to analyze communication protocols and safety critical systems. Usually the actual data that is transported in such a system is of little or no concern which allows for a large reduction in the state space. Also, other aspects of the system can often be (manually) replaced by simpler models. Hence the popularity of the Spin tool [Holzm91] that uses an explicit representation of the state-space, and the symbolic model checker SMV [McMil93].

It should be obvious that the aforementioned verification problems are closely related. Most tools nowadays heavily rely on a BDD package. In a combinational equivalence checker the BDDs represent the boolean functions of the circuit; in sequential equivalence checking and model checking the BDDs are used to represent state transitions and sets of states. With BDDs the bottleneck is often not computation time but memory size. Any breakthroughs in tool performance are likely to result from improvements in the implementation of the BDDs or an altogether different representation of the fundamental objects (such as logic functions and sets), or a combination of both. Some research in this area has already been reported on. It is suggested to integrate various verification engines into a single tool. Each engine is optimized to efficiently solve a particular class of problems. A global strategy is applied to analyze the problem and decide on what engine to invoke. Typical engines are a BDD based verifier, a logic simulator, an ATPG (automatic test pattern generation) or recursive learning based module, and a satisfiability checker. Particularly algorithms for satisfiability enjoy a renewed interest.

If we consider brute-force, push-button, verification tools, as opposed to tools that require quite a bit of user intelligence to get them on their way (e.g. theorem provers), it is clear that they are severely limited by the time and space complexity of the problems they try to solve. Even small problem instances might turn out infeasible to solve. Future work should therefore concentrate on methods of compositional verification, in which a design is partitioned in manageable pieces that may be processed independently. More attention also needs to be paid to automating abstraction and reduction methods. The idea here is to tailor the input data to the verification task. Partial order reduction (as e.g. employed in Spin) is a good example of such a technique: execution behaviours that are indistinguishable with respect to the property to be verified are treated as a single case. Abstraction aims at selectively ignoring irrelevant details in a design, of course without violating the verification outcome. Although the possibility of *false negatives* might in some cases be acceptable, *false positives* cannot be tolerated.

Although the CAD vendors and specialized formal verification companies that offer verification tools cannot (yet) fully live up to their promises, the fact that such tools are now commercially available acknowledges their importance and generates a strong impetus to the research community.

Probably the greatest challenge for research in formal verification is to keep up with Moore's law: a verification tool that today can handle a complete microprocessor, will in a year be required to handle a system on a chip.

References

[Backh80] Backhouse, R.C., *Syntax of Programming Languages: Theory and Practice,* Prentice-Hall, 1980.

[Baren84] Barendregt, H.P., *The Lambda Calculus,* North-Holland, 1984.

[Brace90] Brace, Karl S., Rudell, Richard L., and Bryant, Randal E., "Efficient Implementation of a BDD Package," *Proc. 27-th ACM/IEEE Design Automation Conference,* pp. 40-45, Orlando, Florida, June 24-28, 1990.

[Breid89] Breidegard, Björn, "Lion Cage Example," *Private communication,* Lund University, Sweden, October, 1989.

[Bryan86] Bryant, Randal E., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers,* vol. C-35, no. 8, pp. 677-691, August 1986.

[Burch91] Burch, J.R., Clarke, E.M., and McMillan, K.L., "Symbolic Model Checking: 10^{20} States and Beyond," *International Workshop on Formal Methods in VLSI Design,* 1991.

[Burch91] Burch, Jerry R., "Using BDDs to Verify Multipliers," *Proceedings of the 28th ACM/IEEE Design Automation Conference,* pp. 408-412, San Francisco, June 1991.

[Burch94] Burch, Jerry R., Clarke, Edmund M., Long, David E., McMillan, Kenneth L., and Dill, David L., "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Trans. on Computers Aided Design of Integrated Circuits and Systems,* vol. 13, no. 4, pp. 401-424, April 1994.

[Buurm] Buurman, H.W., J.W.G. Fleurkens, et al., "xplog - an interactive postprocessor for simulation results," *Manual page (1es), Technical University Eindhoven.*

[Corme90] Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L., "Chapter 28: Sorting Networks," in *Introduction to Algorithms,* MIT Press, 1990.

[Dijks76] Dijkstra, E.W., "Programming Methodologies: Their Objectives and Their Nature," in *Structured Programming,* Infotech Int. Ltd., 1976.

[EijkC96] Eijk, C.A.J. van and Jess, J.A.G., "Exploiting Functional Dependencies in Finite State Machine Verification," *Proc. of the European Design and Test Conference,* pp. 9-14, 1996.

[Emers90] Emerson, E.A., "Chapter 16: Temporal and Modal Logic," in *Handbook of Theoretical Computer Science,* ed. Jan van Leeuwen, vol. B: Formal Models and Semantics, pp. 996-1072, Elsevier Science Publishers B.V., 1990.

[Galli87] Gallier, Jean H., *Logic for Computer Science: Foundations of Automatic Theorem Proving,* John Wiley & Sons, 1987.

[Garey79] Garey, Michael R. and Johnson, David S., *Computers and Intractability, A Guide to the Theory of NP-Completeness,* W.H. Freeman, 1979.

[HaleR87] Hale, Roger, "Using Temporal Logic for Prototyping: The Design of a Lift Controller," *Proc. Colloquium on Temporal Logic in Specification,* pp. 375-408, Springer-Verlag, Altrincham, UK, April 8-10, 1987.

[Hoare85] Hoare, C.A.R., *Communicating Sequential Processes,* Prentice-Hall, 1985.

[Holzm91] Holzmann, Gerard J., *Design and Validation of Computer Protocols,* Prentice Hall, 1991.

[Janss86] Janssen, G.L.J.M., "Network Description & Modelling Language - NDML," in *The Integrated Circuit Design Book,* ed. P. DeWilde, pp. 4.60-4.108, Delft University Press, 1986.

[Janss86] Janssen, G.L.J.M., "Circuit Description," in *Circuit Analysis, Simulation and Design,* ed. A.E. Ruehli, Advances in CAD for VLSI, vol. 3, Part 1, pp. 5-43, North-Holland, 1986.

[Janss89] Janssen, G.L.J.M., "Circuit Modelling and Animated Interactive Simulation in Escher+," *Proceedings SCS European Simulation Multiconference, Simulation applied to manufactoring, energy and environmental studies and electronics and computer engineering,* pp. 265-270, Rome, 7-9 June, 1989.

[Janss90] Janssen, Geert L.J.M., "Hardware Verification using Temporal Logic: A Practical View," in *Formal VLSI Correctness Verification, VLSI Design Methods-II,* ed. L.J.M. Claesen, pp. 159-168, Elsevier Science Publishers B.V. (North-Holland), IFIP, 1990.

[Janss92] Janssen, Geert L.J.M., "Progress in Verification with PTL," *Third Periodic Progress Report, BRA 3281 ASCIS, Part 2,* March 19, 1992.

[Janss92] Janssen, Geert L.J.M., "Verification of Finite State Machines using Temporal Logic," *Third Periodic Progress Report, BRA 3281 ASCIS, Part 2,* March 19, 1992.

[Karpl89] Karplus, Kevin, "Using If-then-else DAGs for Multi-Level Logic Minimization," *Advanced Research in VLSI, Proceedings of the Decennial Caltech Conference on VLSI,* pp. 101-117, MIT Press, Cambridge, MA, Pasedena, CA, March 1989.

[Keist] Keister, William, "SPIN-OUT," *(C) Copyright Binary Arts, 1987, 1991.*

[Kropf94] Kropf, Thomas, "Benchmark-Circuits for Hardware-Verification," *2-nd Conference on Theorem Proving in Circuit Design,* September 1994.

[Küehl95] Küehlmann, Andreas, Srinivasan, Arvind, and LaPotin, David P., "Verity - A Formal Verification Program for Custom CMOS Circuits," *IBM Jounal of Research and Development,* vol. 39, no. 1/2, pp. 149-166, 1995.

[Malik93] Malik, Sharad, "Analysis of Cyclic Combinational Circuits," *Proceedings ICCAD'93,* pp. 618-625, Santa Clara, Ca, Nov. 7-11, 1993.

[Manna81] Manna, Z. and Pnueli, A., "Verification of concurrent programs: the temporal framework," in *The Correctness Problem in Computer Science,* ed. Robert S. Boyer, J. Strother Moore, International Lecture Series in Computer Science, Academic Press, New York, 1981.

[McMil93] McMillan, Kenneth L., *Symbolic Model Checking,* Kluwer Academic Publishers, 1993.

[MetsA94] Mets, Arjen A., "Dynamic variable ordering for BDD minimization," *Student report Eindhoven University, Dept. of Electrical Engineering,* January 1994.

[Mohnk93] Mohnke, Janett and Malik, Sharad, "Permutation and phase independent Boolean comparison," *INTEGRATION, the VLSI journal,* vol. 16, pp. 109-129, 1993.

[Niess88] Niessen, Cees, Berkel, C.H. (Kees) van, Rem, Martin, and Saeijs, Ronald W.J.J., "VLSI Programming and Silicon Compilation; A Novel Approach from Philips Research," *IEEE Int. Conf. on Computer Design: VLSI in Computers & Processors,* pp. 150-151, Rye Brook, October 3-5, 1988.

[Peyto87] Peyton-Jones, Simon L., *The Implementation of Functional Programming Languages,* Prentice Hall, 1987.

[Phili89] Philipson, Lars, "A Challenge for Formal Verification of CMOS Logic," *Private communication,* September 3, 1989.

[Raner93] Ranerup, K., Philipson, L., Madsen, J., Oleson, O., and Janssen, G.L.J.M., "Controller synthesis and verification, in," in *Application-Driven*

Architecture Synthesis, ed. Francky Catthoor, Lars Svensson, pp. 211-232, Kluwer Academic Publishers, 1993.

[Rudel93] Rudell, Richard, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Proc. ICCAD'93*, 1993.

[Rudel93] Rudell, Richard, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Workshop Notes International Workshop on Logic Synthesis*, Granlibakken Conference Center, Tahoe City, CA., May 23-26, 1993.

[Schuu94] Schuurmans, N.J.A., "Transforming Behaviour to Structure in an Industrial HDL Environment," *Master Thesis Eindhoven University, Dept. of Electrical Engineering*, August 1994.

[Sistl85] Sistla, A.P. and Clarke, E.M., "The Complexity of Propositional Linear Temporal Logics," *Journal of the ACM*, vol. 32, no. 3, pp. 733-749, July 1985.

[Tarja72] Tarjan, Robert E., "Depth-First Search and Linear Graph Algorithms," *SIAM Journal Comput.*, vol. 1, no. 2, pp. 146-160, June 1972.

[Tarsk55] Tarski, A., "A lattice-theoretical fixpoint theorem and its applications," *Pacific Journal of Mathematics*, vol. 5, pp. 285-309, 1955.

[Venka87] Venkatesh, G., "Modeling and Verification of Digital Systems using Temporal Logic," *Proceedings of the IFIP WG 10.2 8-th International Conference on Computer Hardware Description Languages and their Applications*, Amsterdam, The Netherlands, 27-29 April 1987.

[Warsh62] Warshall, S., "A Theorem on Boolean Matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11-12, Jan. 1962.

[WolfW90] Wolf, Wayne, "An Algorithm for Nearly-Minimal Collapsing of Finite-State Machine Networks," *Proc. IEEE Int. Conf. on Computer-Aided Design*, pp. 80-83, Santa Clara, CA, 1990.

[Wolpe83] Wolper, Pierre, "Temporal Logic Can Be More Expressive," *Information and Control*, no. 56, pp. 72-99, 1983.

[WoodW89] Wood, William G., "Specification of the Elevator Problem Using Temporal Logic," *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 12-14, 1989.

[YangB98] Yang, Bwolen, Bryant, Randall E., O'Hallaron, David R., Biere, Armin, Coudert, Olivier, Janssen, Geert, Ranjan, Rajeev, and Somenzi, Fabio, "A Study of BDD Performance in Model Checking," *Proc. 2-nd Int. Conf. on Formal Methods in Computer-Aided Design*, Palo Alto, CA, November 3-6, 1998.

Biography

Geert Janssen was born on April 12 1956 in Oss, a provincial town near 's–Hertogenbosch, the Netherlands. He attended Titus Brandsma Lyceum to obtain the Atheneum-B diploma in 1974. In 1981 he graduated from Eindhoven University, then called *Technische Hogeschool*, with an *ingenieur* (M.Sc.) degree in Electrical Engineering. Exactly 1 day before the ceremony he was called for to fulfill his military duties in Appingendam, Groningen. Following 6 weeks of basic training, he was appointed assistant teacher at the Military Academy in Breda. After a honorary discharge in the rank of second lieutenant, he joined Philips Telecommunications Industries in Hilversum. There he worked as a software engineer on the development of a new operating system for telephony exchange applications. Just before the establishment of a joint venture with AT&T, Mr. Janssen was offered a Ph.D. position in the group of Prof. Jess at Eindhoven University. Three years later, in 1986, he became a *universitair docent* (assistant professor). This caused him to be more and more involved in teaching and organizational work next to his research for a doctorate. This work was interrupted in the years 1990-1997 for temporary leave during the summer months spent at IBM's T.J. Watson Research Center in Yorktown Heights, USA. His main professional interests are formal methods for hardware design, synthesis, and verification; language theory and programming languages; and algorithms for symbolic computation with applications in Computer Aided Design.

Stellingen

behorende bij het proefschrift

Logics for Digital Circuit Verification:
Theory, Algorithms, and Applications

van Geert Janssen

Technische Universiteit Eindhoven, februari 1999

1. Formele verificatiemethoden voor digitale hardware worden vooralsnog door de industrie sceptisch bekeken. Echter, deze methoden dwingen een ontwerper precieze en ondubbelzinnige specificaties voor zijn ontwerp vast te leggen wat allerlei voordelen heeft naast het primair beoogde doel van formele verwerking. [dit proefschrift].

2. De synchrone arbiter schakeling die McMillan als voorbeeld gebruikt in zijn dissertatie [Symbolic Model Checking, Kluwer, 1993] is niet optimaal ontworpen. [dit proefschrift, hoofdstuk 10].

3. Stelling 4 bij het proefschrift van Gjalt de Jong [Proefschrift Technische Universiteit Eindhoven, 1993] luidend *"De effectiviteit van Binary Decision Diagrams wordt overschat."* moet gezien worden als een te voorbarige en te pessimistische uitspraak, die nu slechts enkele jaren later door de praktijk is achterhaald. Zie de recente studie naar het gebruik van BDDs voor Model Checking [Bwolen Yang, FMCAD'98].

4. Het werk verricht door medewerkers van SRI International (Menlo Park, CA) waarbij de theorem prover PVS wordt uitgebreid met specifieke (en daardoor ook meer efficiënte) redeneermodules, o.a. door het toepassen van het in dit proefschrift beschreven BDD-pakket en μ-calculus programma, geeft een oplossing aan hoe verificatie op consistente wijze over een groot deel van het ontwerptraject kan worden toegepast. [Cyrluk e.a., TPCD'95].

5. Blijkbaar is niet iedereen overtuigd van de noodzaak van het deadlock-vrij zijn van een systeem. Zo laat bijvoorbeeld de wegenverkeerswet de situatie toe dat bij een gelijkwaardige kruising 4 auto's de kruising tegelijkertijd naderen met als gevolg dat elke auto op een andere moet wachten.

6. In wiskundige verhandelingen zou het gebruik van het uitroepteken als leesteken verboden moeten worden! De redenen hier voor zijn er minstens 2! Ten eerste is het beledigend de lezer middels het uitroepteken er op te wijzen dat iets belangrijk is en/of vanzelf spreekt; ten tweede is verwarring met het symbool voor faculteit van een natuurlijk getal niet te voorkomen.

7. Voor de opleiding tot Elektrotechnisch Ingenieur is kennis van en ervaring met mathematisch modelleren van problemen en oplossingsmethoden, in het bijzonder middels graafmodellen en algoritmen, onontbeerlijk. Het dient dan te worden toegejuicht dat een nieuw curriculum (5jr, 1995) voor de faculteit Elektrotechniek van de TU Eindhoven hiervoor inderdaad een, zij het bescheiden, plaats heeft ingeruimd.

8. De moeilijkheden die veel techniekstudenten ervaren bij het aanleren van een programmeertaal zijn grotendeels te wijten aan hun slechte taalvaardigheid.

9. Als zelfs de goden hun eigen taal niet beheersen wat kunnen we dan verwachten van de gewone sterveling?
[N. Wirth, "Data Structures and Algorithms", Scientific American, sep. 1984],
[N. Wirth, "Hardware Compilation: Translating Programs into Circuits", IEEE Computer, juni 1998].

10. Het toenemende aanbod van omvangrijke software pakketten voor PC's, en de daardoor noodzakelijke expansie van PC's naar steeds grotere en snellere systemen, doet vermoeden dat de aandacht voor compacte datastructuren en efficiënte algoritmen danig aan het afnemen is. Hiermee wordt ook de waardering voor het vakmanschap van een programmeur ondermijnd.

11. De slechtste implementatie van een eindige automaat is waarschijnlijk de CWS handdoekautomaat die je in de meeste toiletten van de Technische Universiteit aantreft. Mijn ervaring is dat óf de automaat bevindt zich in een niet-'resettable' toestand (de handdoek zit vast), óf hij is aan het einde van zijn 'tape'.

Logics for Digital Circuit Verification
Theory, Algorithms, and Applications

presents the results of investigating various logics with respect to their application to verification of digital hardware. The approach highlights both the end-user aspects and the implementor's aspects. The thesis is structured in 3 parts. Part I discusses verification problems in the area of combinational circuits; part II focuses on sequential circuit verification, and part III presents the software tools that have been developed and discusses details of their implementations. Also, part III contains a number of test cases that exhibit the typical modelling of problems in terms of the investigated logics and shows how they are solved by the presented tools:

- *bdd - boolean function manipulation*
- *ptl - temporal logic satisfiability checker*
- *mu - propositional μ-calculus tool*
- *bsn2veri - circuit equivalence checker*
- *bsn2mc - Fair-CTL model checker*

This thesis focuses on techniques for hardware verification. The approach is formal, i.e., mathematical theories will be presented that form the basis for modelling the hardware and reasoning about its behaviour. The work concentrates on decidable theories, for which algorithms exist that can be used to prove certain properties of the circuit. Central to this thesis are the application of the theory and the development of efficient algorithms.

ISBN 90 386 1560 4