Equivalence checking of multi-level logic functions by means of multi-level tautology

Nijsten, A.J.

Award date:
1988

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
DESIGN AUTOMATION GROUP

EQUIVALENCE CHECKING OF MULTI-LEVEL LOGIC FUNCTIONS

BY MEANS OF MULTI-LEVEL TAUTOLOGY.

A.J. Nijsten

Master thesis
reporting on graduation work
performed from 11.01.88 to 24.08.88
by order of prof. dr. ing. J.A.G. Jess
and supervised by ir. M.R.C.M. Berkelaar.

Abstract.

This report presents a method of testing whether two multi-level logic circuit descriptions are equivalent or not. Existing functional equivalence tools operate only on two-level descriptions. Since multi-level descriptions are frequently encountered, the only means of checking their equivalence is to flatten them to two-level descriptions first. To avoid this inefficient flattening operation, an algorithm which operates directly on multi-level descriptions is developed.

The algorithm is based on tautology checking of the h function composed of the two functions which need to be compared. A divide and conquer technique is used to solve the tautology question. The h function is evaluated to select the "best" Shannon expansion variable. This expansion is performed repeatedly, until the the resulting functions are small enough to answer the tautology question.

Experimental results show that the criteria used to guide the choice of this "best" expansion variable influence the speed of the algorithm considerably. Several criteria are discussed and tested. The best criterion found so far is based on the concept of the probability of an intermediate function of this h function going to a fixed value after expansion. Since this fixed value is used in other intermediate functions, priority is given to forcing the ones with largest fan out to a fixed value. Experimental data shows that the algorithm is reasonably fast, but it can't compete with an existing two-level comparison technique if two-level functions are checked.

Finally some tests are performed, with semi-flattening of the multi-level logic functions. Two simple criteria to determine which intermediate function has to be substituted are tested. The obtained results look very promising.

CONTENTS

LIST OF FIGURES

Introduction.

This report presents a method of testing whether two multi-level logic circuit descriptions are functionally equivalent. While developing software which performs transformations on sets of logic functions (e.g. software for minimisation, decomposition or technology mapping of logic functions) one must be able to compare the input descriptions with the output, in order to verify the correctness of the used software or algorithm.

Unless the described logic circuit is a PLA, the description used will not be a in a sum of product form. Rather multi-level circuit descriptions will be used.

At the moment the only powerful algorithms available for equivalence checking, operate on sum of product descriptions. Thus when the need arises to compare two multi-level logic descriptions, the only means of doing so is to transform each multi-level representation to a sum of product (two-level) form and then apply a two-level comparison technique to these flattened descriptions.

The main disadvantage of this technique is: flattening can be very expensive in terms of CPU time, because this process involves repeated application of the complementation and intersection operation. Moreover, the resulting two-level descriptions can become very large, i.e. can contain many cubes. As a result the flattening operation can be very time consuming, as well as the two-level comparison of the large flattened functions.

To avoid this inefficiency, a method to operate directly on the multi-level descriptions has to be developed. A possible approach, restricted to completely specified logic functions, based on multi-level tautology checking is described in this report.

1. Basic definitions.

In order to describe the algorithm in the next chapter properly, the basic terms and principles used frequently throughout the rest of this report need to be discussed first. Most of the definitions are adopted from references [1] and [2]. Although a multi-level tautology algorithm is discussed, we start this chapter with definitions and descriptions of two-level boolean functions, because a multi-level logic function can be thought of as composed of a set of linked two-level descriptions.

1.1 Representation of logic functions.

Strictly speaking, a logic function is a transformation which maps the input set B = {0,1} to the output set Y = {0,1,2}. See equation 1.

$$f : B^n \rightarrow Y^m \tag{1.1}$$

It represents a logic function mapping the n input variables $x_1,...,x_n$ to the m output variables $y_1,...,y_m$.

Note that the output may also assume the don't-care value 2. If Y = {0,1}, i.e. the function doesn't assume don't-care values for any input combination, the function is a completely specified logic function otherwise it is an incompletely specified logic function.

The notation f_i $(1 \leq i \leq m)$ is used to represent the i^{th} component of f, which is just one of the m outputs. The name boolean function is reserved for such a single component f_i which is a multiple input, single output logic function.

Many representations of a logic function are possible, the most straight forward one is the truth table. In this table, the output values are listed for every possible combination of the inputs. Because this isn't a very efficient description, the algebraic representation together with a description suited for calculations in a program will be used. We will focus on the sum of product representation (1.2), since it can be used to represent any logic function as a set of boolean functions and it is easy a form to be described by a data-structure for calculations in a program.

Example:

$$\begin{aligned} f_1 &= \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_2x_3 + \bar{x}_1\bar{x}_2x_3 + x_1x_3 \\ f_2 &= x_2x_3 + \bar{x}_1x_2 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3 \end{aligned} \tag{1.2}$$

To transform the sum of product form to this data-structure, the concept **cube** is introduced. A cube is equivalent to a complete product term in a sum of product representation. It can be specified by a row vector $\mathbf{c} = [c_1,...,c_n]$ where:

$$c_i = \begin{cases} 0 & \text{if } x_i \textit{ appears complemented in the product term.} \\ 1 & \text{if } x_i \textit{ appears uncomplemented in the product term.} \\ 2 & \text{if } x_i \textit{ doesn't appear in the product term.} \end{cases} \tag{1.3}$$

$1 \leq i \leq n.$

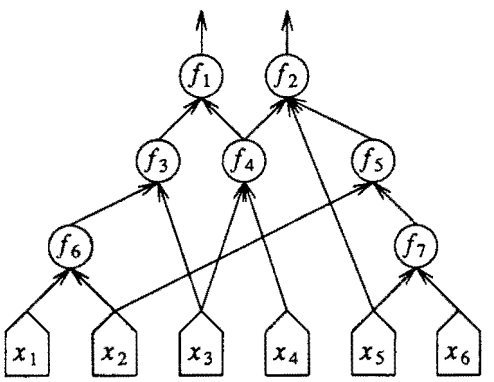
n Stands for the number of different literals in the sum of product equation.

This implies that we can represent every cube in a boolean function (in sum of product form) by a row vector **c**. By stacking all these row vectors, we get a matrix which represents the complete boolean function. For example, the matrix of f_1 see (1.2) is pictured in figure 1. This matrix is called a **matrix representation** of a two-level single output logic function. Note, that we need a matrix for every component f_i, to describe a multiple output logic function completely.

$$F = \begin{bmatrix} 0\,0\,0 \\ 2\,0\,1 \\ 0\,0\,1 \\ 1\,2\,1 \end{bmatrix}$$

Figure 1. Matrix representation of a sum of product logic function

Now the multi-level logic functions can be described, which are the primary object of our multi-level equivalence checking. Such a multi-level function can be pictured as in figure 2. In this figure we can distinguish three basic elements, **Primary Inputs**, **Primary Outputs** and **Intermediate Functions**. We define **PI** as the set of primary inputs, **PO** as the set of primary outputs and **IF** as the set of intermediate functions.



Primary outputs PO = $\{f_1, f_2\}$

Intermediate functions
IF = $\{f_3, \ldots, f_7\}$

Primary inputs
PI = $\{x_1, \ldots, x_6\}$

Figure 2. Graphic representation of a multi-level logic function.

Basically, a multi-level logic function maps the primary inputs to the primary outputs via the intermediate functions.

Every intermediate function is a completely specified two-level boolean function described as a sum of products in algebraic or matrix representation. The inputs to these intermediate functions are primary inputs or outputs of other intermediate functions. The output of them is propagated towards the inputs of other intermediate functions or is the primary output of the total multi-level logic function.

Thus, in order to describe a multi-level logic function completely, we need to know the PI and PO set and to specify all the intermediate functions $f_1, \ldots, f_l$ with their interconnections.

As mentioned before, an intermediate function which is a two-level boolean function, can be described efficiently by a matrix. Such an intermediate function can have primary inputs and intermediate

inputs, which are outputs of other intermediate functions. As a result, we can divide the cubes in intermediate functions in two sets (1.4).

$$c = [Pi_1,..,Pi_n,Io_1,..,Io_m] \tag{1.4}$$

where:

$$Pi_i = \begin{cases} 0 \text{ if } \textit{primary input i appears complemented in the product term.} \\ 1 \text{ if } \textit{primary input i appears uncomplemented in the product term.} \\ 2 \text{ if } \textit{primary input i doesn't appear in the product term.} \end{cases}$$

$$1 \le i \le n$$

$$Io_i = \begin{cases} 0 \text{ if } \textit{output of intermediate j appears complemented in the product term.} \\ 1 \text{ if } \textit{output of intermediate j appears uncomplemented in the product term.} \\ 2 \text{ if } \textit{output of intermediate j doesn't appear in the product term.} \end{cases}$$

$$1 \le j \le m$$

Every Io_i represents a complete intermediate function in an other intermediate function.

A complete single output multi-level logic function is described by associating two numbers with every intermediate function f_i in FI, for every intermediate function f_j which accepts f_i's output as an input. These new modified matrices are stacked to form the complete function described by the **modified matrix representation**. The first number of these two, named **Link** is the number j of the intermediate function f_j which accepts the output of f_i as input. The second one named **Index** is the position of a literal in a cube of f_j. This literal represents the complete intermediate function f_i. I.e. index is the position of one of the elements Io_j in (1.4). Also see the example in figure 3.

Finally we define the **fan out** of an intermediate function f_i as the number of intermediate functions f_j which accept f_i's output as an input.

1.2 Operations on logic functions.

Operations on logic functions, used later on, will be defined in this section. Since the matrix representation is used primarily, the operations are defined on this type of description.

The **empty cube**, although not really a cube by our definition, is introduced as a matter of convenience. A normal cube is represented by a row vector with a certain number of elements. If this set of elements is an empty set, the cube is called an empty cube, denoted by the symbol ϕ.

$$y_1 = (x_1 + x_2)\,\overline{(\bar{x}_3 x_4)} + \overline{(x_2 x_4)}\,\overline{(x_3 + x_4)}$$

PI = $\{x_1, x_2, x_3, x_4\}$
PO = $\{y_1\}$
IF = $\{f_1 f_2 f_3 f_4 f_5 f_6 f_7\}$

where:

$f_1 = x_1 + x_2$ $\qquad$ $f_5 = f_1 \bar{f}_2$
$f_2 = \bar{x}_3 x_4$ $\qquad$ $f_6 = \bar{f}_3 \bar{f}_4$
$f_3 = x_2 x_4$ $\qquad$ $f_7 = f_5 + f_6 = y_1$
$f_4 = x_3 + x_4$

There are 4 primary inputs and 6 real intermediate functions, thus 4 + 6 = 10 literals per cube. This results in a modified matrix notation as listed below.

$x_1\ x_2\ x_3\ x_4\ f_1\ f_2\ f_3\ f_4\ f_5\ f_6$

$f_7 =$ 2 2 2 2 2 2 2 2 1 2
2 2 2 2 2 2 2 2 2 1
link = undefined, primary output
index = undefined

$f_6 =$ 2 2 2 2 2 2 0 0 2 2
link = 7
index = 10

$f_5 =$ 2 2 2 2 1 0 2 2 2 2
link = 7
index = 9

$f_4 =$ 2 2 1 2 2 2 2 2 2 2
2 2 2 1 2 2 2 2 2 2
link = 6
index = 8

$f_3 =$ 2 1 2 1 2 2 2 2 2 2
link = 6
index = 7

$f_2 =$ 2 2 0 1 2 2 2 2 2 2
link = 5
index = 6

$f_1 =$ 1 2 2 2 2 2 2 2 2 2
2 1 2 2 2 2 2 2 2 2
link = 5
index = 5

Figure 3. Example: Multi-level function represented by a modified matrix.

The **complement** of a cube $\mathbf{c} = [c_1,...,c_n]$ is a matrix **M** with cubes m_1 through m_n. Where $m_i = [2,..,\bar{c}_i,..,2]$, $\bar{c}_i = 1$ if $c_i = 0$, $\bar{c}_i = 0$ if $c_i = 1$. m_i is an empty cube if $c_i = 2$.

The **intersection** or product of two cubes $\mathbf{c} = [c_1,...,c_n]$ and $\mathbf{d} = [d_1,...,d_n]$, written as $c \cap d$, is a cube **e** $= [e_1,...,e_n]$. The entries e_i are obtained from the entries of **c** and **d** according to the diagram in figure 4.

	$\cap$	d_i: 0	1	2	
	0	0	ϕ	0	
c_i	1	ϕ	1	1	$1 \le i \le n$
	2	0	1	2	

Figure 4. Diagram for the intersection of two cubes.

If there is an index i such that $c_i = 1$ and $d_i = 0$ or vise versa, the resulting cube **e** is said to be the **empty cube.**

The **union** or sum of two cubes $\mathbf{c} = [c_i,...,c_n]$ and $\mathbf{d} = [d_1,...,d_n]$ is the matrix **M** as described in (1.5). If cube c as well as d is an empty cube, matrix **M** reduces to an empty cube **M.**

$$M = \begin{bmatrix} c_1 & \cdots & c_n \\ d_1 & \cdots & d_n \end{bmatrix} \tag{1.5}$$

Although strictly speaking, tautology isn't an operation, it is defined in this section. A completely specified logic function f is a **tautology** if its outputs are 1 for all possible combinations of the input. I.e. Y = {1}, see equation (1.1).

The Shannon expansion and Shannon cofactor are basic concepts for our equivalence algorithm. The **cofactor** of a cube $\mathbf{c} = [c_1,...,c_n]$ with respect to a cube $\mathbf{p} = [p_1,...,p_n]$ is a cube $c_p = [c_{p_1},...,c_{p_n}]$ with its components as defined in (1.6). This implies that after cofactorisation, the cube c_p is independent of all literals in cube **p.**

$$c_{p_k} = \begin{cases} \phi & \text{if } c \cap p = \phi \\ 2 & \text{if } p_k = 0 \text{ or } p_k = 1 \\ c_k & \text{if } p_k = 2 \end{cases} \tag{1.6}$$

$$1 \le k \le n$$

The cofactor of a two-level logic function f with respect to cube **p** is defined as a two-level logic function f_p. This function f_p consists of all the cubes of f cofactored with respect to **p** as described in (1.6).

In our algorithms we will calculate the cofactor with respect to a cube with only one literal in position j. This special cube is denoted by v_j. The complement of this cube, the j^{th} entry replaced by a 0 if the original one was a 1, or vise versa is denoted by $\bar{v}_j$.

For a moment we return to the algebraic notation. The **Shannon expansion** of a two-level logic function f is (1.7):

$$f = x_j f_{x_j} + \bar{x}_j f_{\bar{x}_j} \tag{1.7}$$

and f_{x_j} is called the cofactor of f with respect to the variable x_j. Note that this definition can be translated directly to a matrix representation by replacing x_j by v_j and f, f_{x_j} by matrix representations of these functions. As a result f_{x_j} is independent of x_j.

Because multi-level logic functions are the main object of this report, a multi-level cofactor needs to be defined as well. Given a modified matrix (multi-level logic function) **F** with the sets PI = $\{x_1,..,x_j,..,x_n\}$ and IF = $\{f_1,...,f_m\}$. The cofactor of **F** with respect to v_j $(i \le j \le n)$[1] is a modified matrix F_{v_j} with all f_i elements of the set IF, cofactored with respect to v_j. Thus F_{v_j} has the sets PI = $\{x_1,..,x_{j-1},x_{j+1},..,x_n\}$ and IF = $\{f_{1_{x_j}},...,f_{m_{x_j}}\}$.

Proof: We use two properties of the cofactor proven in [1]. Let f and g be algebraic representations of completely specified boolean functions. Then the following operations commute:

1. Intersection of two functions and cofactor operations, i.e.
   $(f \cdot g)_{x_j} = (f_{x_j} \cdot g_{x_j})$

2. Complementation of a function and cofactor operations, i.e.
   $\overline{(f)}_{x_j} = \overline{(f_{x_j})}$

As stated above, the cofactor of the modified matrix is only calculated with respect to one of the elements of the PI set. According to the descriptions in the previous paragraph, every intermediate function $f_i \in$ FI consists of one or more cubes of the type listed in equation (1.4). As a result the operation F_{v_j} only effects the part $Pi_1,...,Pi_n$ of every cube, the part $Io_1,...,Io_m$ remains unchanged. The intermediate functions represented by Io_1 to Io_m are also cofactored with respect to v_j.

According to the intersection table of figure 4, cube **c** can be rewritten as in (1.8).

$$c = [Pi_1,...,Pi_m] \cap [Io_1,...,2] \cap \cdots \cap [2,...,Io_m] \tag{1.8}$$

If we translate this to the algebraic representation, we get (1.9) in which f^p only depends on primary inputs and g_1 through g_m each represents an element of the IF set.

$$c = f^p \, g_1 \, g_2 \, ... \, g_m \tag{1.9}$$

If we apply property 2 of the cofactor operation to (1.9) we get (1.10),

$$c_{v_j} = (f^p \, g_1 \, g_2 \, \cdots \, g_m)_{x_j} = f_{x_j}{}^p \, g_{1_{x_j}} \, \cdots \, g_{m_{x_j}} \tag{1.10}$$

which can be translated back to (1.11).

---

1. Note: v_j has n + m entries, but an entry unequal to 2 can only occur in the range from 1 to n.

$$c_{v_j} = [Pi_1,..,Pi_{j-1},Pi_{j+1},\ldots,Pi_n,Io_{1_{v_j}},\ldots,Io_{m_{v_j}}] \tag{1.11}$$

Note that this multi-level cofactorisation is equivalent to the normal Shannon cofactorisation.

2. A multi-level equivalence algorithm.

In this chapter an algorithm to check the equivalence of two completely specified single output logic functions f and g is described. Unless stated otherwise, every function in this section is assumed to be of this type. The discussed algorithm is based on an idea in reference [3].

2.1 Basic concepts.

The function h, created according to equation (2.1), plays a central role in the algorithm. By inspection of this functions truth table it can be seen that $h = 1$ if and only if $f = g$.

$$h = f \cdot g + \bar{f} \cdot \bar{g} \tag{2.1}$$

Thus, checking whether function h is a tautology, results in testing the equivalence of function f and g.

If functions f and g are both multi-level logic functions, each specified by their PI, PO and IF sets, then h is a multi-level logic function. The PI set of h is the union of the PI sets of f and g. The IF set is the union of three sets; the IF sets of f and g and a set $Q = \{q_1, q_2, q_3\}$. The functions in Q represent:

$$\begin{aligned} q_1 &= f \cdot g \\ q_2 &= \bar{f} \cdot \bar{g} \\ q_3 &= q_1 + q_2 \end{aligned} \tag{2.2}$$

Proposition 1.

A basic proposition in the context of tautology checking is [1]: Let $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$ be the Shannon expansion of a completely specified logic function f. Then $f = 1$ (i.e. f is a tautology) if and only if $f_{x_i} = 1$ and $f_{\bar{x}_i} = 1$.

Proof: We first proof the "if" and then the "only" part of this "if and only if" proposition.

If part, if f_{x_i} and $f_{\bar{x}_i}$ are tautologies, the values of their outputs are 1 for all values of the inputs. Hence $x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i} = x_i + \bar{x}_i = 1$.

Only part, suppose one of the cofactors of f, say f_{x_i}, is not a tautology. Then there exists an input combination so that one of the outputs of f_{x_i}, say y_j is 0. Since f_{x_i} is independent of x_i, an input combination with $x_i = 1$ can be selected so that the output of y_j of $x_i f_{x_i}$ is 0. For this input combination, the output of $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$ corresponding to y_j must also be 0 and therefore f is not a tautology.

Before the algorithm is discussed, the property of the cofactor listed in (2.3) has to be proved.

$$f_{ab} = f_{ba} \tag{2.3}$$

Proof:

$$f_{ab} = (f_a)_b = ab\, f_{ab} + a\bar{b}\, f_{a\bar{b}} + \bar{a}b\, f_{\bar{a}b} + \bar{a}\bar{b}\, f_{\bar{a}\bar{b}} \tag{2.4}$$

$$f_{ba} = (f_b)_a = ab\, f_{ba} + \bar{a}b\, f_{b\bar{a}} + a\bar{b}\, f_{\bar{b}a} + \bar{a}\bar{b}\, f_{\bar{b}\bar{a}} \tag{2.5}$$

Since f_{ab} and f_{ba} are both independent of a and b, equating the terms on the right hand side of the equal sign in (2.4) and (2.5) yields (2.3).

Finally it is made plausible why both functions f and g need to be completely specified. Suppose f is an incompletely specified function, which isn't specified for input vector x, but maps x to 0 when applied as input. If for example this function is reduced to an equivalent function f' with a minimum number of cubes, it may proof efficient to assign output 1 to the unspecified input x. If we use the tautology check of (2.1), to test the equivalence of f and f', function h returns a 0 for input x. Thus, although the functions may be equivalent for all specified inputs, the test fails.

2.2 A solution to the tautology problem.

By introduction of equation (2.1) the question of multi-level equivalence of two logic functions f and g is reduced to checking the tautology of a single multi-level logic function h.

The proposition in the previous section regarding Shannon expansion and tautology forms the basis for the proposed solution. Since the Shannon expansion offers the possibility to divide a logic function into two smaller functions, it provides the basis for a divide and conquer strategy for answering the tautology question.

If the cofactor with respect to some variables in the PI set of h is calculated recursively on each subsequent cofactor of h, a binary tree as in figure (5) is created. At each node of this tree, one out of three actions can be taken:

- The cofactor is found to be a tautology and the recursion can be terminated on this branch of the tree.
- The cofactor is found not to be a tautology, hence on basis of proposition 1, h is not a tautology and f and g are not equivalent.
- Neither of these two is possible, again perform the Shannon expansion on this node with respect to a new element of the PI set of h.

This is also the basis for a powerful two-level tautology algorithm described in [4].

Because h is a multi-level function, the multi-level cofactor is applied to h. This implies that all elements of the IF set are cofactored in the usual two-level way with respect to a variable x_j in the PI set. As can be seen from the definition of the multi-level cofactor, this variable x_j is removed from every intermediate function. This may modify these cofactored intermediate functions in such a way that they can be simplified to some constant value (0 or 1). If so, each intermediate variable Io_i (1.4) representing such a simplified function can be replaced by this value, keeping in mind that $Io_i = 1$ represents direct substitution and $Io_i = 0$ complemented substitution of the constant. All these modified intermediate functions in their turn may also simplify to a logic constant. These values are also substituted. In the end the substitution may result in $h = 0$ or $h = 1$. If $h = 1$ the function at this node of the cofactor tree is a tautology, in case of a 0 the node is no tautology and hence the functions f and g are not equivalent. If a node is a tautology, the original function h at the root of the binary cofactor tree is a tautology on the sub-set of the cofactored variables, i.e. $f = g$ on that sub-set. When the union of these sub-sets equals the PI set of h, h is a tautology and f=g. With this in mind the three previously listed actions at the nodes modify to:

- The cofactor is found to be a tautology, because the output of h equals 1. The recursion can be terminated on this branch of the tree.

- The cofactor and as a result the complete function h is found not to be a tautology, because h equals 0 at this node. The complete equivalence test can be terminated.
- After cofactorisation the output of h remains undefined, perform the Shannon expansion again at this node.

for the multi-level case.

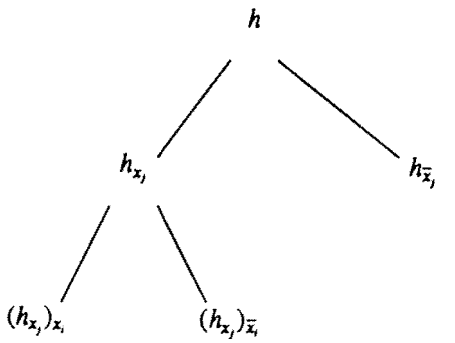


Figure 5. A part of a binary cofactoring tree.

This can be translated directly to a multi-level equivalence checking algorithm. It is listed in a C like language in figure 6.

As can be seen from figure 6, all functions in the algorithm are precisely defined, except SELECT. According to equation (2.3), one is completely free to choose the order in which a function is cofactored with respect to all elements in the primary input set. As is shown in appendix A, the cofactorisation order can have a large influence on the number of nodes in the binary cofactor tree and hence on the time, the algorithm needs to determine the equivalence of the two functions.

A method to solve the simplify problem is discussed in the next paragraph. SELECT is discussed in the next chapter.

2.2.1 A simplify algorithm.

Simplify is a function which has to determine if an intermediate function in the modified matrix representation evaluates to a constant value 0 or 1.

An intermediate function consists of one or more cubes. Bear in mind that the empty cube is also a cube. If an intermediate function is made out of just one cube, simplification is quite simple. If this cube is an empty cube, the intermediate evaluates to 0, if all entries are 2's, it evaluates to 1.

Before intermediate functions with more than one cube are discussed, the concept of a **cover** is introduced. A cube $\mathbf{c} = [c_1,...,c_n]$ is said to cover a cube $\mathbf{d} = [d_1,...,d_n]$, written as $\mathbf{c} \supset \mathbf{d}$, if each entry of $\mathbf{c}$ **contains** the corresponding entry of $\mathbf{d}$. An entry c_j contains an entry d_j ($c_j \supset d_j$) according to the table in figure 7.

```
/* Create function h according to (2.1). */
h = CREATE_H (f, g)
output = TAUTOLOGY (h)
if (output == 1)
  f and g are equivalent.
else
  f and g are not the same.

/* The function tautology, returns a 1 if the function is a tautology. */
/* It returns a 0 if it is no tautology. */
TAUTOLOGY (f)

/* Select a variable x_i from the PI set of f for the Shannon expansion */
x_i = SELECT (f)

/* calculate cofactor with respect to x_i */
f_{x_i} = COFACTOR (f, x_i)

/* simplify cofactored function and test if output is a constant */
output_0 = 2
output_1 = 2
f_{x_i} = SIMPLIFY (f_{x_i}, &output_0)
if (output_0 == 0)
  return (0)
if (output_0 == 2)
  output_0 = TAUTOLOGY (f_{x_i})

/* move to other branch of binary cofactor tree */
if (output_0 != 0)
  f_{x̄_i} = COFACTOR (f, x̄_i)
  output_1 = 2
  f_{x̄_i} = SIMPLIFY (f_{x̄_i}, &output_1)
  if (output_1 == 0)
        return (0)
  if (output_1 == 2)
        output_1 = TAUTOLOGY (f_{x̄_i})

/* Test if termination at both branches was a tautology */
if (output_0 == 1 && output_1 == 1)
  return (1)
else
  return (0)
```

Figure 6. A multi-level equivalence algorithm.

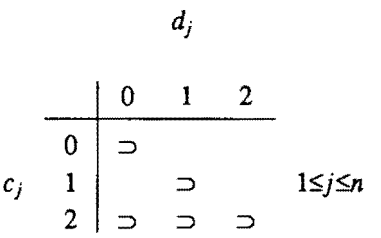


Figure 7. Diagram for the cover of two cube entries.

If a cube **c** contains a cube **d**, cube **d** is redundant and can be removed from the intermediate function.

Thus in the multiple cube case an intermediate function evaluates to 1 if one of these cubes say **c** is completely made out of 2's. This because cube **c** covers all other cubes in the intermediate function. These other cubes are redundant, reducing the problem to the single cube case.

Of course it is possible to apply other simplification rules to every intermediate function, to check if it evaluates to 0 or 1. But these checks are much more complicated than the two tests described above and hence are not performed.

3. Several selection criteria.

As demonstrated in chapter 2, SELECT is the only means of influencing the speed of the algorithm. In this section, several ideas regarding the choice of the cofactorisation variables are discussed. The efficiency of each algorithm is discussed in the next chapter by means of a comparison of the test results of each implementation on a set of test files. Basically we can divide the selection criteria in two different types. The first is based on the number of entries in the columns of the modified matrix, representing the multi-level logic function. The second set is based on the fan out of every intermediate function in the IF set. Just for comparison purposes, a criterion which doesn't fit in any of these sets is introduced. It randomly selects a variable for the cofactorisation operation. It is denoted by SELECT version 1.0.

3.1 Selection based on the number of entries per column.

In the algorithm, cofactoring is performed with respect to the elements in the PI set. As a result only the Pi elements (1.4) in every cube are subjected to this operation. If cofactorisation of a function f represented by a modified matrix F is performed with respect to literal x_j, the column representing this literal in F is removed, as well are the cubes in which the literal x_j appears complemented.

The algorithm of chapter 2 is based on the concept of simplification of intermediate functions and on possible propagation of logic constants after cofactorisation. In order to get results during the equivalence check, the output of a function h at a node of the binary cofactor tree has to evaluate to 0 or 1. Thus the selection of the cofactorisation variables has to be such that it forces the output of h to a logic constant value as fast as possible.

An intermediate function simplifies to a logic constant if it contains a single cube which is an empty cube or a cube which is completely made out of 2's. See paragraph 2.2.1. As a result, we want to remove as many entries unequal 2 as possible during each cofactorisation step. This can be done by cofactoring with respect to the column in the modified matrix containing the largest number of entries unequal 2. But if the selected column, say the one representing x_j, contains much more 1 entries than 0 entries, cofactorisation with respect to x_j and $\overline{x}_j$ results in an asymmetrical binary cofactor tree. The cofactorisation with respect to x_j only removes the column representing x_j and a small number of cubes from the modified matrix, while cofactorisation with respect to $\overline{x}_j$ removes a column and a large number of the cubes. As a result, the node containing h_{x_j} holds a rather large function (many cubes) and the function in the node $h_{\overline{x}_j}$ contains a small function. Thus, the algorithm may terminate rather quickly in the branch which is connected to the node with $h_{\overline{x}_j}$ while it may take a long time to get results in the branch connected to the node with the large function h_{x_j}, because the chance of an intermediate function with a small amount of cubes evaluating to 0 or 1 within the next few cofactorisation steps is larger than with a large number of cubes. With this in mind it is possible to develop several selection criteria. To facilitate future reference each criterion is discussed in a separate paragraph denoted by a select version number in the title.

In this section the terminology select a column is used, of course the variable which is represented by an entry in this column is meant and used for the Shannon expansion.

3.1.1 SELECT version 2.0.

This select algorithm tries to find the column with exactly the same number of 0 and 1 entries in order to keep the binary cofactor tree as symmetrical as possible. If this column doesn't exist, it selects the column with the largest number of entries unequal 2. In other words it tries to reduce the functions in each child

node, which are a result of the Shannon expansion of the parent node, by the same amount. If this column isn't found it tries to reduce at least one child node by the largest amount possible. This in order to get at least one branch of the tree to a fixed value as soon as possible.

3.1.2 SELECT version 2.1.

This version gives priority to the symmetrical binary cofactor tree. It tries to keep the reduction at each child node as equal as possible. In order to achieve this goal the column in which the difference between the number of 0 entries and the number of 1 entries is the smallest is selected.

3.1.3 SELECT version 2.2.

As opposed to the previous version, this one gives priority to forcing at least one branch of the tree to a fixed value as fast as possible. In order to do so it selects the column with the largest number of entries of one type. These entries are of course unequal to two.

3.1.4 SELECT version 3.0.

Because the multi-level equivalence algorithm is based on the propagation of constant values towards the output of a circuit described by equation (2.1), an intermediate function has to assume a constant value as soon as possible. As a result we can evaluate every intermediate function in order to find out for which cofactorisation variable the cofactors simplify to 0 or 1.

For every cube in the modified matrix we can assign scores to the Pi elements (1.4). This score depends on the chance of an intermediate function going to a constant value after cofactorisation with respect to some variable. In the end when the complete function is evaluated, the scores are added for every column. The column with the highest score is selected for the next Shannon expansion. While evaluating the intermediate functions in order to calculate the scores we can distinguish a number of different cases.

First we will discuss all possibilities in case of an intermediate function with only one cube. This single cube can have one or more entries unequal 2. It is clear that if there is a single variable, $x_j \in$ PI, set cofactorisation always results in a constant value, regardless if we determine the cofactor with respect to x_j or $\bar{x}_j$. Thus for every column of the modified matrix with one or more variables of this type we add a high score say α for every variable of this type to the score already assigned to this column.

If there are more variables in a single cube intermediate function, cofactorisation only results in a constant value if it is calculated with respect to the complemented value of this variable. Thus a lower score say β is added for every variable of this type to the scores of the proper columns.

In a multiple cube intermediate function we can distinguish several cases. A multiple cube intermediate only assumes a constant value if one of the cubes is forced to contain only 2's after the Shannon expansion. Thus in case of a single variable in one of the cubes we assign a high score α to the column representing this variable. If there are more variables in a cube, cofactorisation with respect to one of these only reduces the number of columns in the modified matrix, and removes some of the cubes from the intermediate, thus a low score γ is assigned to them.

It is not possible to determine fixed values for α, β and γ. The "best" values are found by using the implemented software on a set of test files, with different values for these score factors.

3.1.5 SELECT version 5.0, 5.1.

This select criterion is identical to SELECT version 3.0. The only difference is found in the method of assigning scores to the variables in the multiple cube intermediate function case.

As described in version 2.2, at least one branch of the tree is likely to be forced to a logic constant within the next few cofactorisation steps, if we cofactor with respect to the variable represented by the column with the largest number of entries unequal 2. We still assign a high score α to the single variable in a cube, because they give us the guarantee of a constant value in the next step. But in order to find the column with the largest amount of entries unequal 2, the number of entries in every column of the intermediate is counted. Then a score ranging from 0 to an upper limit β, proportional with the number of entries, is assigned to every column. Version 5.0 and 5.1 differ from each other in the magnitude of the scores α and β.

3.2 Selection based on the fan out of the intermediate functions.

The basic thought behind the selection criteria based on the fan out of intermediate functions is that an intermediate with a large fan out has influence on many other intermediate functions when it evaluates to a constant value. Thus some selection criteria are developed which take the fan out in consideration when selecting a Shannon expansion variable.

3.2.1 SELECT version 3.1.

This selection criterion is identical to 3.0, accept for the fact that the scores which are assigned to the columns of the intermediate functions are multiplied by the intermediates fan out, before they are added to the total score already assigned to the column.

3.2.2 SELECT version 6.0.

As mentioned before, the article in reference [4] describes a powerful algorithm for solving the tautology question for two-level functions. This algorithm is based on the concept of unate functions. With SELECT version 6.0 we try to find out, if we get efficient test results if we apply the concept of unate functions to the intermediates of a multi-level function. Note that intermediate functions are two-level functions. First some basic concepts regarding unate functions are introduced.

A logic function f is monotone increasing in a variable x_j if changing x_j from 0 to 1 causes the output of f, if it changes at all, to increase from 0 to 1. It is monotone decreasing in x_j if a change in this variable from 0 to 1, causes the output to decrease from 1 to 0. Also under the restriction that the output doesn't have to change. A function that is either monotone increasing or decreasing in x_j is said to be **unate** in x_j. It is called a **unate function**, if it is unate in all elements of the PI set.

Whether a function is unate or not can easily be deduced from its truth table representation, but our main representation is the one using cubes. A two-level function which is described by a matrix is unate in one or more variables, if the columns representing these variables are either void of 0's or void of 1's. The big disadvantage of our representation by matrices is that if a function f represented by matrix **M** is unate, **M** may still have columns with 0 as well as 1 entries. This is shown in figure 8.

Given a two-level function in algebraic representation:

$f = x_1\bar{x}_2 + \bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3$

The truth table:

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

The corresponding matrix representation:

$$F = \begin{bmatrix} 1 & 0 & 2 \\ 2 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \qquad F_R = \begin{bmatrix} 1 & 0 & 2 \\ 2 & 0 & 1 \end{bmatrix}$$

From its truth table can be seen that the function is unate since it is increasing in x_1 and x_3, and decreasing in x_2. If we examine the matrix F, the function has two unate columns but the third column clearly isn't. F_R represents the function of after removal of the redundant row.

Figure 8. Example of a unate function in different representations.

But in the example of figure 8, the third row of matrix **F** is covered by the first one. As a result, it can be removed, which leaves three unate columns in **F** (see F_R). Thus it seems that only if the matrix representation of a unate function is reduced to some minimal form, the columns of the matrix are either void of 0's or void of 1's. This is called the minimal term prime implicant cover.

In the unate recursion algorithm of reference [4] the following proposition is used. A unate cover is a tautology if and only if it contains a row of all 2's. In other words, if a unate function has a row of all 2's, it evaluates to 1. In order to apply this proposition, a cofactorisation variable is selected. The choice of this variable is guided by the heuristic of making f_{x_j} and $f_{\bar{x}_j}$ unate after a minimum number of Shannon expansions.

Since we may not be dealing with a matrix representation in the minimal form as discussed above, the logic functions described by f_{x_j} and $f_{\bar{x}_j}$ may be unate even though their representations aren't. In this reference they try to find the most binate variable. This is a variable represented by a column with 0 as well as 1 entries. By this choice an attempt is made at keeping the total number of cubes which are a part of both f_{x_j} and $f_{\bar{x}_j}$ as small as possible.

This two-level algorithm can be translated to the multi-level tautology problem. According to the proposition in the discussed algorithm, a function evaluates to 1 if it has a row of all 2's. Thus in order to get results in the two-level case, they will try to reach this situation as fast as possible. This is something we also want to happen in the intermediate functions of multi-level case.

The idea is to apply this unate recursion algorithm to the intermediate function with the largest fan out. Then the complete multi-level function is cofactored with respect to this most binate variable. This is repeated until the intermediate with largest fan out evaluates to a constant value. The intermediate with the next largest fan out is selected and so on until h (2.1) evaluates to 0 or 1.

The formula used in the two-level algorithm to find the most binate variable is described in equation (3.1).

$$j = \max_i \left(\alpha \, MIN\,[N_{0_i}, N_{1_i}] + \beta\,[N_{0_i} + N_{1_i}]\right) \tag{3.1}$$

$$\alpha = 1$$

$$\beta = 2$$

$$N_{0_i} = \textit{number of } 0 \textit{ entries in } i^{th} \textit{ column}$$

$$N_{1_i} = \textit{number of } 1 \textit{ entries in } i^{th} \textit{ column}$$

This function tries to find the most binate variable, but if all variables are unate, it tries to find the column with the largest number of entries unequal 2. Since if a column is unate, the MIN expression in (3.1) is always 0 and a score equal to the number of entries, times β is assigned to it. If one of the columns is binate, a score equal to the total number of entries times β, plus the minimum of the number of entries of both types is assigned, giving priority to the most binate variable in many cases. This function (3.1) is also used in the multi-level algorithm to select the Shannon expansion variable.

The performance of all these criteria on a large set of test files is compared in chapter 6.

4. Semi-flattening of multi-level logic functions.

Chapter 3 starts with stating that SELECT is the only means of influencing the speed of the algorithm. While implementing the software for the multi-level tautology algorithm, the idea to partially flatten the multi level functions was brought forward. This brings a multi-level function "closer" to the two-level functions. If a multi-level function is flattened, the length of the propagation paths for constant values is reduced, which means that if an intermediate function evaluates to a constant value, it may take less following cofactorisation steps before the effect is noted at the output of equation (2.1). Opposed to this assumption is the fact that if we partially flatten a multi-level function by substitution of intermediates in the intermediates which they are linked to, the resulting intermediate is much more complicated. Thus we have to test this idea, to see if the advantage of the short propagation paths is superior to the disadvantage of the more complex intermediate functions.

Thus, to test the effect of semi-flattening, some criteria are developed, which decide which intermediate is substituted and which isn't. The effect of these ideas in terms of speed of the algorithm is tested in the next chapter. But first a solution to the flatten algorithm is discussed.

4.1 A solution to the flatten problem.

Basically flattening is a straightforward operation. It involves repeated application of the intersection and complementation operation. We restrict the flatten algorithm to substitution of intermediates in which all columns representing Io_1 to Io_m (1.4) equal 2. The reasons for this restriction are that it is just a wild guess if flattening really improves the speed of the algorithm. Thus it is useless to put a lot of effort into developing a versatile and efficient flatten algorithm at this stage.

The second reason is a more practical one. Say we substitute the intermediate f_j, in which for example Io_i is unequal 2. This variable Io_i represents a complete intermediate function which is linked to f_j. In order to keep the flattened version of the complete function equal to the original one, we have to substitute this linked intermediate in f_j, before we substitute f_j in the intermediate it is linked to. An other solution would be to simply substitute f_j the way it is and relink all intermediates which were previously linked to f_j to the intermediate in which f_j is substituted.

But as mentioned above, the idea of flattening was introduced rather late. As a result, the used data structure is not fit for efficiently discovering which functions are linked to for example f_j. The only information directly available for every intermediate, is the intermediate it is linked to. Thus if we want to implement the option of randomly substituting intermediates, the complete Io part (1.4) of the modified matrix representing an intermediate has to be searched in order to find the functions linked to it. This is a rather time consuming operation, since these arrays can become very large, if there are a lot of other intermediate functions, and hence not performed.

Note that even with this restriction it is still possible to flatten a function to a sum of product form by repeated application of the algorithm. Every time the algorithm is used, all allowable intermediates are substituted.

We restrict our attention to substitution of intermediates of the type with Io_1 to Io_m equal to 2. The substitution problem is pictured in figure 9. It involves the cube of intermediate function f_i with the variable x_j and the complete intermediate function f_j. If this particular cube c of f_i in figure 9 is rewritten in

the algebraic representation, we get equation (4.1).

$$\begin{aligned} c &= x_2 \cdots x_l \cdots f_j \quad \text{if } x_j = 1 \\ c &= x_2 \cdots x_l \cdots \overline{f_j} \quad \text{if } x_j = 0 \end{aligned} \tag{4.1}$$

Thus, substitution is the intersection of cube **c** with the intermediate f_j or with the complement of f_j if x_j is 0. The complement and intersection of cubes are precisely defined in chapter 1. But in this case a cube is intersected with a matrix. Before the intersection, x_j is reset to 2, since after substitution f_j doesn't exist any more and Io_j is of no further importance.

$$Pi_1 \cdots Pi_l \cdots Pi_n Io_1 \cdots Io_j \cdots Io_m$$

$f_l =$ 2 1 $\cdots$ 1 2 1 0 $\cdots$ 2 $\cdots$ 2
0 2 $\cdots$ 0 2 2 2 $\cdots$ 2 $\cdots$ 2
$link = \cdots$
$index = \cdots$
.
.
.

$f_i =$ 2 1 $\cdots$ 2 2 2 2 $\cdots$ x_j $\cdots$ 2
0 0 $\cdots$ 2 2 2 2 $\cdots$ 2 $\cdots$ 2
$link = \cdots$
$index = \cdots$
.
.
.

$f_j =$ 0 1 $\cdots$ 2 2 2 2 $\cdots$ 2 $\cdots$ 2
1 2 $\cdots$ 2 2 2 2 $\cdots$ 2 $\cdots$ 2
$link = i$
$index = n + j$
.
.

Figure 9. Representation of the elements involved in substitution.

First the complement of a matrix representation is discussed. A matrix represents a sum of product logic function. If such a function is complemented every product term modifies to the sum of the complemented literals in this term. All these complemented product terms are intersected to find the complement of the original function.

As a result, the **complement of a matrix F** representing a two-level logic function is the intersection of the complement of all the cubes in **F**. Note that this involves the intersection of matrices.

The **intersection of two matrices, F and G.** Given matrix **F** with cubes $f_1,...,f_n$ and matrix **G** with cubes $g_1,...,g_m$. The intersection of **F** and **G** is a matrix **H** with cubes $h_{11},..,h_{1m},h_{21},..,h_{2m},..,h_{n1},..,h_{nm}$. With h_{ij} according to (4.2).

$$h_{ij} = f_i \cap g_j \tag{4.2}$$

Now the intersection of a cube with a matrix is defined as well. Since a cube can be regarded as a matrix with a single row. By these definitions, the flatten problem is reduced to the level of matrix representations and cubes which makes it suitable for a software implementation.

A disadvantage of the intersection of a cube with a matrix and especially a complemented matrix is that the resulting matrix tends to contain a lot of redundant and even duplicate cubes. These large intermediate functions slow down further substitution of them selves and others. They also keep the complete function unnecessarily large, which may slow down some of the steps in the tautology algorithm.

For these reasons a simple redundancy check is introduced. It removes all the cubes in an intermediate which are covered by an other cube. This doesn't reduce the intermediate to an absolute minimal form, but removes all duplicate rows and the redundancy of the type listed in (4.3)

$$a + ab = a \tag{4.3}$$

4.2 Two flatten criteria.

As discussed in the introduction of this chapter, we want to find out, if the advantage of the shortened propagation paths, has a greater influence on the speed of the algorithm than the disadvantage of the more complex intermediate functions which are a result of the substitution operation. The first flatten criterion is rather straight forward. It substitutes all intermediate functions with no variables of type Io_j, equal 0 or 1 (1.4).

The substituted intermediates are the ones with the longest propagation paths. Thus, this substitution reduces these paths by one. They are also the ones that only contain variables representing elements of the PI set. In the tautology check, cofactorisation is always performed with respect to these input variables. Thus, if the complete logic function is cofactored with respect to some variable, these intermediate functions have a high probability of being affected by the Shannon expansion.

The intermediate function f'_j which is a result of the substitution of f_s in f_j contains a sub-set of the primary input variables of f_s. Thus, because of the substitution, the probability of f'_j being affected by the Shannon expansion increases and f'_j is more likely to evaluate to a logic constant. The longest propagation path in the semi-flattened function is shorter than the one in the unflattened function. As a result, a branch in the cofactorisation tree may terminate sooner, thus speeding up the complete tautology check.

The second criterion is based on the fan out of the intermediate functions. In the chapter describing several versions of SELECT, a set of criteria also based on the fan out are discussed. The idea is that if an intermediate with a large fan out evaluates to a logic constant, this effect is noted in a lot of other intermediates. These may evaluate to a logic constant as well. If this concept is combined with the idea of the short propagation paths, we arrive at the thought to substitute all intermediates with a fan out below a certain threshold.

If this idea is applied repeatedly until all intermediates with a fan out below this threshold are substituted, the ones with little influence if they evaluate to a logic constant are removed. The remaining ones are the ones with the large fan out, which affect many other intermediates if they evaluate to a logic constant. It is possible that after completion of the flatten operation functions with a fan out below the

threshold still exist, since the restriction regarding substitution with variables Io_j unequal 2 is still valid. The results obtained with these two flatten criteria are discussed in one of the next chapters.

5. Some notes on the implementation.

The multi-level tautology algorithm is implemented in C. This chapter is intended to give a precise description of the data structure used to represent multi-level functions. Also some remarks about the basic routines CREATE_H and COFACTOR are made, since they operate in a slightly different way as may be expected from previous definition of the representation of logic functions and the calculation of the Shannon expansion. These modifications are made to avoid unnecessarily large data structures, because these would slow down the speed of several calculations.

5.1 The data structure.

The data structure for the representation of the multi-level functions consists of four basic elements. These are the structures FUNCT, CUBE and two types of arrays. See figure 10. As stated before, a multi-level function is made out of linked two-level functions. See for example figure 3.

```
typedef struct cube
  {     int           inter_nr;
        int           not_two;
        short         *array;
        struct cube   *prev;
        struct cube   *next;
  } CUBE;


typedef struct funct
  {     int           no_var;
        int           no_inter;
        struct cube   *cube;
        CUBE          **info;
  } FUNCT;
```

Figure 10. Definition of CUBE and FUNCT.

The basic element of a two-level function is a cube, defined according to equation (1.4). Such a cube can be represented by an array of short integers, with the entries 0,1 and 2. Also specified by (1.4). In order to represent a sum of cubes, the type CUBE is introduced. The arrays are stacked to form the well known matrix representation. This is done by linking the the elements of the type CUBE by means of the prev and next fields. The pointer in CUBE with the name array points to an array representing a cube. In order to speed up certain calculations the number of elements in this array unequal to 2 is stored in the not_two field of CUBE. To specify uniquely to which intermediate function an element of the type CUBE belongs, every intermediate function is numbered and this number is stored in the inter_nr field of the structure.

For each intermediate function it needs to be known to which intermediate it is linked and by which index it is represented. For this purpose an element of the type CUBE is added to the last element of every structure representing an intermediate. The array pointer points to a small array, which contains the following entries:

Index	Purpose
0	Stop code to signal end of intermediate function
Odd index	Number of the intermediate it is linked to.
Even index	Index where variable Io can be found.

The not_two field of this tail element is used to store the number of entries in this small array. Note that the fan out of the intermediate can be calculated from this number (5.1).

$$fan\ out = \frac{(not\_two - 1)}{2} \tag{5.1}$$

Thus all entries with an odd index in this array contain the number of an intermediate function. In order to jump from an intermediate to the one it is linked to, the second type of array is used. This array, pointed to by the info field in the type FUNCT, contains pointers to the first elements of every intermediate function. This pointer is stored at the index corresponding to the intermediate number of the element it points to. For example if intermediate function number 4 is linked to intermediate number 2, the tail array of intermediate 4 contains the number 2 at an odd index. At element 2 of the info array a pointer pointing to the first element of intermediate 2 is stored.

Some additional information about the length of the info array and the arrays representing the cubes is stored in the no_inter and no_var fields respectively of an element of type FUNCT. Figure 11 tries to picture the data structure.

5.2 The routine CREATE_H.

The first step in the algorithm is the creation of the *h* function (2.1). See figure 6. This function is made out of the two functions which equivalence is checked. According to previous descriptions, a unique column is assigned to every variable of the type Pi and type Io. If the *f* and *g* functions are linked to create the *h* function some columns filled with 2's need to be added to both functions if we want to maintain this property of unique columns for different variables.

Since the largest amount of the input variables of functions *f* and *g* are the same these columns can be shared. But if for example the *f* function has more input variables than the *g* function, a column filled with 2's has to be inserted in the *g* function for every additional input variable in *f*. These columns are inserted in between the part representing the Pi variables and the part representing the Io variables.

When the data structure is built, completely according to the specification of assignment of unique columns to every different variable, a column of 2's needs to be added to both functions for every column representing an intermediate variable in the other function. This results in a kind of matrix representation for the *h* function, with several different data fields. It is drawn schematically in figure 12.

The biggest disadvantage of this representation is that there are two large blocks completely filled with 2's. Since the only operation performed with respect to the "columns" of this structure is cofactorisation, which is only executed with respect to the primary input variables, a data reduction can be introduced. As a result of this restriction to the cofactorisation variables, there is no need to assign unique columns to the different intermediate variables of the functions *f* and *g* if they are combined to form the *h* function. Thus only some columns filled with 2's have to be added to the function with the smallest amount

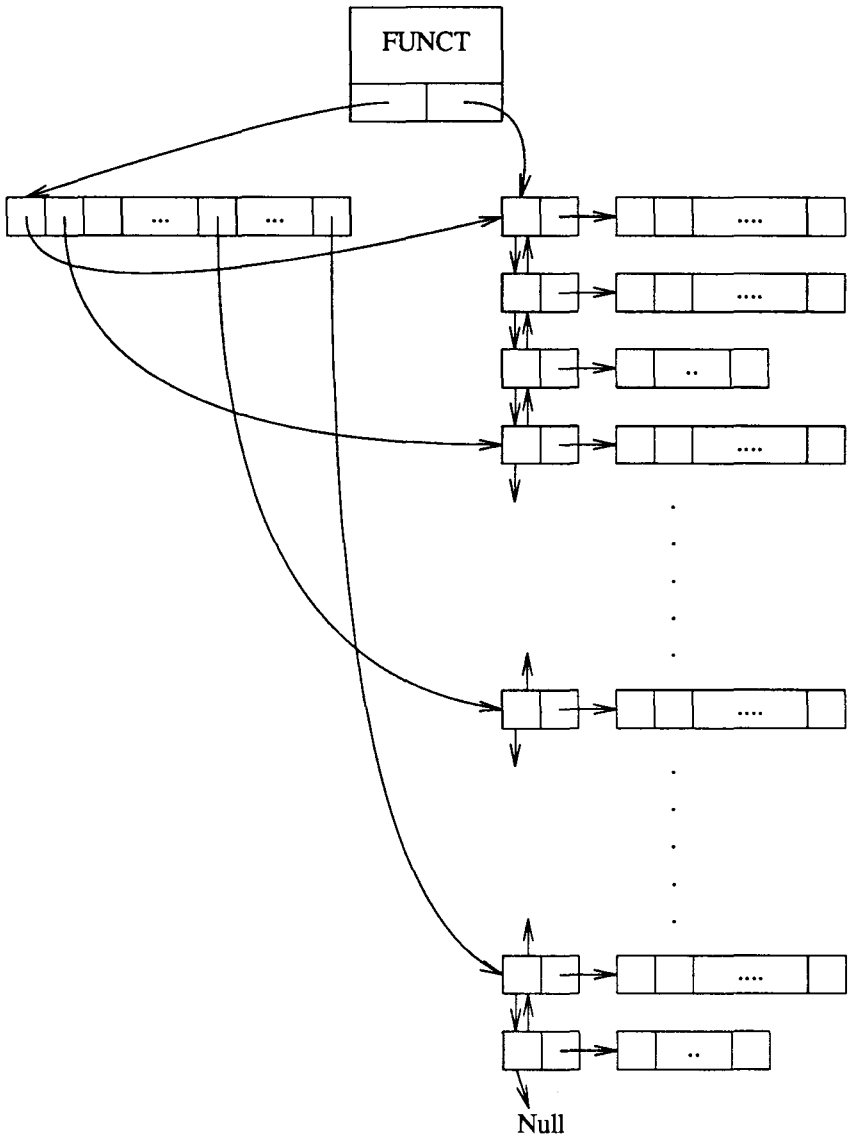


Figure 11. Graphic representation of the data structure.

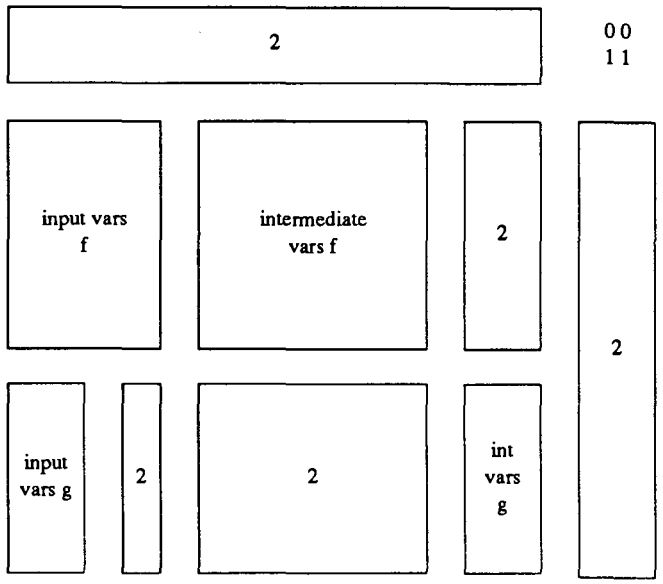


Figure 12. H function without data reduction.

of different intermediate variables. This results in a structure pictured in figure 13.

In short, CREATE_H, creates an intermediate function representing $\overline{fg} + fg$. It copies the arrays in the intermediate functions of f and g to the larger arrays of h, with the correct amount of two's inserted in the proper places. Finally it renumbers all intermediate functions and updates the link information in the small tail arrays to the new intermediate numbers.

5.3 The routine COFACTOR.

This routine has to perform the Shannon expansion of the h function in a node of the cofactorisation tree. As can be seen from figure 6, COFACTOR calculates the function in one of the two child nodes, depending on the fact if the cofactorisation variable is complemented or not. Since both child nodes need to be known, COFACTOR copies the h function to a data structure representing h_x. Suppose the Shannon expansion is performed with respect to the variable represented by the i^{th} column, and the variable is uncomplemented. I.e. represented by a 1 entry in the i^{th} position of the arrays. For every array in the data structure of function h is checked if it needs to be copied to the structure of h_x according to the rules listed in (1.6). If the resulting cube is an empty one the array isn't copied otherwise it is.

In order to keep the data structure small, the arrays are copied to the structure of h_x with exception of the i^{th} entry. This as opposed to the definition in (1.6) which states that the i^{th} entry has to be replaced by a 2. This action is allowed since replacing a complete column by one completely filled with 2's, makes the function independent of the variable represented by this column. It will never be selected as a cofactorisation variable again and may as well be removed from the data structure. Since the routine always removes one column representing a primary input variable, it also needs to update some of the information in the tail arrays of every intermediate function. I.e. the index entries have to be decreased by one. These are the entries with an even index.

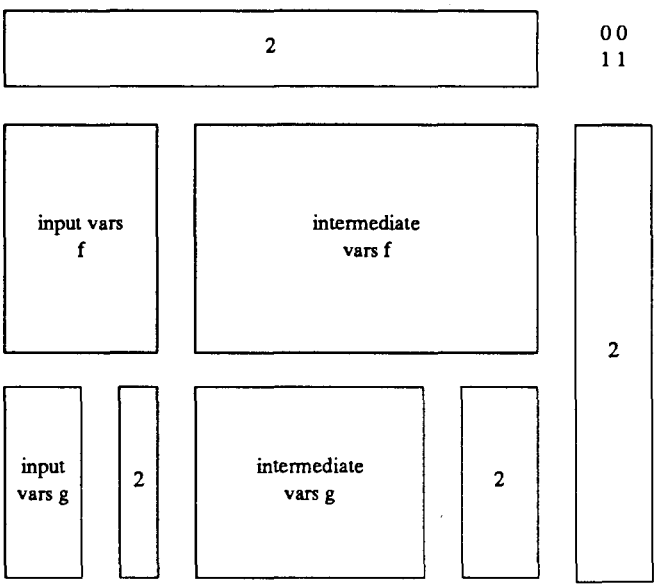


Figure 13. H function with data reduction.

5.4 How to use the program.

The program is used to check the equivalence of two files describing a logic circuit by means of "logic syntax". The program is called as in (5.2)

$$tau\ [-options]\ file\_1\ file\_2 \tag{5.2}$$

to check the equivalence of the circuits described in file_1. and 2. Legal options are:

-o The results of the comparison is always printed on the terminals CRT. But with this option the results are also summarised in the file ML_OUT.res.

-f If this option is used, the functions in file_1 and 2 are partially flattened, before the equivalence check is executed.

6. Results for multi-level equivalence checking.

In order to determine which select criterion is the "best" one, the program is used to test the equivalence of a large set of test functions. These functions are mostly multi-level descriptions of multiple output logic circuits. The algorithm is designed to evaluate single output logic functions. But since every output of a circuit is described by a separate equation, the equivalence of two multiple output circuit descriptions can be tested by applying the algorithm repeatedly to each of these separate equations, until the equivalence question is answered for every function in the file.

For comparison purposes, the files are characterised by a number called depth. This is the largest number of successive opening brackets in the file. For example the depth of equation (6.1) is 3, since every opening bracket is considered as the beginning of a new intermediate function, it is also a size for the length of the longest path in one of the data structures.

$$f = ((a + b\overline{c}) + a((\overline{ac} + da) + ef)) \tag{6.1}$$

Two numbers are introduced to measure the performance of the implementation. The first one is the number of CPU seconds used to answer the tautology question. The program is implemented in C and executed on a HP 9000 computer running UNIX.

The second figure of merit is the number of leaves. As discussed before, cofactorisation of the h function (2.1) results in a binary cofactorisation tree. If the algorithm is an efficient one it doesn't need many Shannon expansions to determine if the functions are equivalent or not. This results in a small tree. An inefficient algorithm results in a large tree. The number of leaves in a tree, is a good quantity to give an impression of the size of it. Hence it is well suited as a figure of merit. In case of multiple output logic functions, the number of leaves, to give an impression of the number of calculations needed , is the sum of the leaves of all the trees, since every output produces its own tree while checking the equivalence.

In the comparison of the criteria, the number of leaves will be used. The CPU time is listed to give an impression of the actual time used for the calculation. Since if the program is used twice to check the equivalence of two files, the returned time varies while the number of leaves is always the same.

If we compare all CPU times needed to check the tautology of the same files with different versions of SELECT, we see that in nearly all cases the check with the shortest CPU time also produces the tree with the smallest amount of leaves. See appendix B for a complete listing of the test results. Thus, it is allowed to use the number of leaves as the main figure of merit.

The names of the files containing descriptions of logic circuits which are used for testing the program, can have one out of tree different extensions:

- inp, denoting a file with a sum of product description.
- 2222.gf, denoting a multi-level description in which the circuit is realised by means of 3 * 3 and inverter or gates.
- dec.gf, denoting a multi-level description in which the circuit is realised by means of 4 *4 and inverter or gates.

The general properties such as the number of inputs, number of outputs and depth of all used files are listed

in table B.1 of appendix B. The comprehensive listing of test results in appendix B is summarised by two representative figures which are listed in tables 1 and 2. The first one, average number of leaves is the average of all numbers of leaves in one table of appendix B. The second one, average time is calculated in exactly the same way. By inspection of the complete test result tables it can be seen that, if a selection criterion A results in a slightly higher average number of leaves with respect to some other criterion B, criterion A is slower than B in most of the cases, even in nearly all of the cases where the number of leaves is very small. Thus these two figures aren't mutilated by extreme large amounts of leaves which occur in some cases.

6.1 Is the implementation correct ?.

Before the performance of several selection criteria is discussed, we need to know if it is really possible to check the equivalence of two multiple output multi-level logic circuit descriptions. As mentioned before there are three types of files describing the same circuit.

The following procedure is used to test the program. Say there are three files A, B and C each describing the same circuit. First the program is used to check the equivalence of files A and B. Then C is checked against A. If these files are equivalent, an error is introduced in one or more output equations of file C. If the algorithm is correct, comparing file C with A as well as B has to result in a message that the distorted output functions aren't the same. This happens in all test cases.

The introduced errors range from deletion and addition of variables to the deletion of complete intermediate functions. If redundant variables are added, in an attempt to create a non-equivalence, the check doesn't fail and still reports that the file are equivalent.

Some of the test files used in this chapter aren't equivalent, this results in a warning from the program that, one or more output equations aren't the same. This difference can always be traced down to a variable which has to be complemented in order to restore the equivalence.

Note, that since the equivalence of two files can be tested by means of this algorithm, it is also possible to check the flatten algorithm. To do so, the program is slightly modified. It reads the same file twice, during the first time the functions in the file are flattened. The second time they are not. Then the equivalence of both versions of the same file is checked. It turns out that the flattened description is always completely equivalent to the original one. Thus flatten functions propperly.

6.2 A comparison of the results.

In this section, the test results are discussed. We start with a comparison of the results for the implementation with SELECT versions 2.0, 2.1 and 2.2. The complete results are listed in table B.3.

If the number of leaves produced by SELECT 2.0 is compared with the amount as a result of SELECT 2.1, version 2.0 is slower in only 9 cases. In 5 cases they are equally fast. As described in chapter 2, SELECT 2.0 tries to find columns with exactly as much 1 as 0 entries. Probably there aren't many columns of this type, thus the column with the largest number of entries is selected in most cases.

In SELECT 2.2 absolute priority is given to the column with the largest number of entries. As a result we can expect 2.2 to be faster than 2.0. According to table B.3 is SELECT 2.2 faster in 76 out of the 87 cases. The average number of leaves produced per calculation is reduced by a factor 140. These average

values are listed in table 1. As can be seen i SELECT version 2.2 is superior to both other versions.

The criteria with number 3 were designed to assign scores to the primary input variables. Version 3.1 is faster than 3.0 in 54 cases. They are equally fast in 10 cases. As can be seen from table 1 and appendix B, is version 3.1 slightly better than 3.0. Versions 5.0 and 5.1 are also based on the concept of assigning scores to the primary input variables. But as listed in table 1, they aren't as good as the SELECT 3. criteria. In average version 5.0 is twice as fast as 5.1. This illustrates the importance of the choice of α and β. From these comparison results we can conclude that:

- The criterion to select the variable represented by the column with the largest number of entries as the cofactorisation variable, is a rather good choice.
- The assignment of scores to variables, depending on the chance of an intermediate evaluating to a logic constant after cofactorisation, is an even better criterion.
- The choice of the score factors α and β has a large influence on the performance of the selection criterion.
- Criteria which take the fan out of an intermediate into account, while selecting the "best" Shannon expansion variable, give better results than their counterparts which don't.
- The criterion based on unate intermediate functions doesn't give the expected optimal performance.

After testing SELECT 3.1 with different score factors, it is chosen as the best criterion. It is implemented with $\alpha = 100$, $\beta = 50$ and γ set to 0.

Table 1
Average test results.

SELECT version number	average #leaves	average time (s)
1.0	33819	877
2.0	101900	2909
2.1	119212	3512
2.2	722	55
3.0	1029	66
3.1	656	51
5.0	8255	284
5.1	18980	588
6.0	33545	883

6.3 Results obtained with semi-flattening.

The two flatten criteria discussed in chapter 4 are tested in an implementation with SELECT 3.1 as the algorithm to choose the "best" Shannon expansion variable. The criterion based on repeated substitution of

intermediates with a fan out below a certain threshold is denoted with version number 2.0. The other one with 1.0. The complete results are listed in appendix C. They are also summarised in table 2.

Flatten 2.0 was tested with several values for this threshold, but tests show that if this threshold is chosen larger than one, the algorithm needs a very long time to solve the tautology problem. This because there are a lot of intermediates with a very small fan out, since every expression within two brackets is considered as a separate intermediate function. Thus the algorithm flattens the function almost completely, which takes a very long time. In order to limit the time needed for the flatten operation only the intermediates with a fan out of one are substituted.

Table 2
Average test results with flattening.

FLAT version number	average #leaves	average time (s)
1.0	569	41
2.0	577	72

According to the results in tables C.1 and B.4 is the selection criterion 3.1 in combination with flatten version 1.0 faster in 72 out of 87 cases than the same test without flattening. If we use flatten version 2.0, the programme is still faster in 62 cases.

Thus although some CPU time is needed to partially flatten a multi-level function, flatten version 1.0 is still faster in nearly all cases. The fact that the results for flatten 2.0 aren't as good can't be ascribed completely to the fact that this flattening operation is more complex. Since the average number of leaves, in table 2, is slightly larger than the average number needed with version 1.0.

For every entry in the tables of appendix B, we can calculate the number of leaves produced per second. This quantity can be assigned the the comparison of file X with file Y. It turns out that every time file X is compared with file Y, this quantity is nearly the same. Most deviations in this quotient can be ascribed to the fluctuation in the measured CPU time. This number of leaves produced per second is independent of the version of the selection criteria even if they are used with flattening 1.0.

Thus if the number of leaves produced per second is a constant for the equivalence check of two files, the time to compare these two files decreases if flatting modifies the functions to functions which equivalence can be tested with fewer leaves in the binary cofactoring tree.

6.4 Two-level versus multi-level.

The algorithm for equivalence checking of two-level logic functions based on unate recursion, described by Brayton in [4] is also implemented. The CPU time needed by this two-level algorithm and the CPU time needed by the multi-level tautology algorithm to compare two-level logic function descriptions is listed in table 3.

Table 3
CPU time two-level versus multi-level algorithm.

file name		two-level	multi-level
file A	file B	time (s)	time (s)
primes8.log	primes8.sim	3.1	19.7
primes9.log	primes9.sim	10.0	54.4
primes10.log	primes10.sim	35.7	187.6

The two-level algorithm is about a factor 5 to 6 faster than the multi-level algorithm if two files both containing a two-level description are compared. One note needs to be made with these results. The routine used to read the functions in the multi-level algorithm is more complicated than the one used to read the functions in the two-level case. As a result our multi-level routine needs more time to read a two-level function. This works in favour of the unate recursion algorithm, but it is a fact that it is faster than the multi-level routine although the real factor in check times is less than 6.

7. Recommendations for future work.

At the moment the algorithm is designed to check the equivalence of two single output multi-level logic functions. It checks the equivalence of multiple output logic functions by considering each output equation as a single output multi-level logic circuit description. These single output functions are compared.

It turns out that in a lot of cases, these single output equations share a set of intermediate functions. For example the equations for functions y_1 and y_2 in figure 2 will both contain the intermediate function f_4. Thus these shared intermediates need to be evaluated every time the tautology of two single output functions is checked. As a result, selection of a cofactorisation variable based on all output functions in the files and Shannon expansion of all these functions simultaneously avoids this repeated evaluation of the shared intermediates and may speed up the complete tautology check considerably.

The algorithm is only capable of checking the equivalence of two completely specified logic functions. The algorithm can be modified such that it accepts a list of input vectors which are mapped to don't cares. If we cofactor a function f with respect to the variable x, we create a new function f_x. The truth table of f_x is independent of x and contains all entries of the table of f in which x = 1. The other function $f_{\bar{x}}$ contains all truth table entries of f with x = 0. The column representing x is of course deleted in the tables for f_x and $f_{\bar{x}}$. Thus, cofactorisation with respect to $\bar{a}\bar{b}c$ creates a function $f_{\bar{a}\bar{b}c}$ with a truth table found by deletion of the columns representing a, b and c from f's truth table. Only the rows of f's table with a = 1, b = 0, and c = 1 are placed in the table of $f_{\bar{a}\bar{b}c}$.

As discussed, the algorithm creates a binary cofactoring tree. Every leaf represents a small boolean function which is the result of a sequence of cofactor operations. The truth table of such a leaf function (e.g leaf function $f_{\bar{a}\bar{b}c}$) is found from the table of the root function as discussed above. The root function is a tautology if all leaf functions are a tautology, thus if the truth table of every leaf function contains only 1's in the column representing the output. Two incompletely specified logic function descriptions are at least equivalent for all specified input vectors. As a result, the incompletely specified logic functions are equivalent if all leaf functions, which are a result of the cofactorisation paths with respect to an input vector with a specified output, are tautologies. Thus we simply exclude the leafs which are a result of the cofactorisation paths with respect to the input vectors with unspecified outputs in order to compare incompletely specified logic functions. Or in short, by not cofactoring the h function with respect to the sequences described by the input vectors in the don't care list, we exclude the entries of the truth table of h which may be 0, from occurring in one of the truth tables of the leaf functions. Also see the example in figure 14.

Since the results obtained with semi-flattening are promising, some further research into better flatten criteria has to be conducted.

The multi-level routine can be combined with the unate recursion algorithm, in order to make it powerful in case two two-level functions are compared. This is also an efficient approach if flattening of the functions results in completely flattened function, although this is a situation which isn't very likely to occur.

Finally it is pointed out that the program can be used in redundancy detection and elimination algorithms, and automatic test pattern generation algorithms.

A node of a circuit is termed redundant if it can be replaced by a logic constant value without altering the output behaviour of the circuit. To determine whether or not a node is redundant, two copies of the circuit description are created. One is left unaltered and one is altered so that the node in question represents a constant logic value. If these two circuit descriptions are equivalent, the node is redundant, otherwise it isn't.

In order to create test patterns for stuck at faults, a copy of the circuits description is made. In the copy the node which is tested, is replaced by the logic constant value representing the stuck at fault. Now the *h* function is created and the tautology check is performed. If the node we are testing isn't redundant, some cofactorisation paths in the binary cofactoring tree will result in a leaf function which isn't a tautology. As a result, for the input vector corresponding with this path the description with the stuck at fault is different from the one without this fault. By expanding the tree until the union of input vectors, corresponding with the paths in the tree cover the complete input space, we get several input vectors which case the original function to be different from the one with the stuck at fault. These can be used to test the circuit for a stuck at fault at this node. If there are for example 6 input variables, such a vector can consist of 1 to 6 elements. If less then 6 elements are specified in such an input vector, the value of the unspecified variables is unimportant.

Given the truth table of a function with don't cares represented by x.

a	*b*	*c*	
0	*0*	*0*	*1*
0	*0*	*1*	*1*
0	*1*	*0*	*1*
0	*1*	*1*	*1*
1	*0*	*0*	*x*
1	*0*	*1*	*0*
1	*1*	*0*	*x*
1	*1*	*1*	*1*

This results in f_0 *if we map the don't cares to 0.*

$$f_0 = \bar{a} + abc$$

This results in f_1 *if we map the don't cares to 1.*

$$f_1 = \bar{a} + a\bar{b}\bar{c} + ab\bar{c} + abc$$

This results in a h function as listed below. The input vectors which map to don't cares are $a\bar{b}\bar{c}$ *and* $ab\bar{c}$

$$h = \bar{a} + a\bar{b}c + abc$$
$$h_a = \bar{b}c + bc$$
$$h_{ab} = c$$
$$h_{abc}=1$$
$$h_{a\bar{b}} = c$$
$$h_{a\bar{b}c} = 1$$
$$h_{\bar{a}} = 1$$

Note that $h_{a\bar{b}\bar{c}}$ *and* $h_{ab\bar{c}}$ *results in a 0, causing the files to be different, but these cofactorisation sequences are excluded in the don't care list.*

Figure 14. Example of tautology checking of a function with don't cares.

8. Conclusions.

Tests show that the algorithm is capable of testing whether two files, each containing the description of a multiple output multi-level logic function, are equivalent or not.

The speed of the algorithm can be influenced considerably by means of the choice of the selection criteria for the Shannon expansion variable. The criteria, based on the number of entries per column of the modified matrix, prove to be efficient. But the ones based on the assignment of a score to each primary input variable, which is related to the probability of the intermediate function evaluating to a logic constant after cofactorisation, are even better. If this is combined with the idea of multiplication of each score by the fan out of the intermediate function under evaluation, it results in the best selection criterion found so far.

The algorithm is reasonably fast while checking the equivalence of two multi-level descriptions, or the equivalence of a two-level with a multi-level description. But it can't compete with a two-level tautology algorithm if the equivalence of two-level circuit descriptions is checked.

The first results obtained in terms of CPU time needed for an equivalence check with semi-flattening of multi-level logic functions, look promising.

Appendix A.

Relation selection order of cofactorisation variables, versus number of nodes in the binary cofactor tree.

In this section an example is given to show the importance of the order in which the function is recursively cofactored with respect to the elements in its primary input set. For convenience the algebraic representation is used.

Given a function f with PI = $\{x_1,x_2,x_2\}$ is cofactored with respect to the variables $x_1x_2x_3$ in (A.1) and with respect to the variables $x_3x_2x_1$ in (A.2).

$$
\begin{aligned}
f &= x_1 + \bar{x}_1x_2x_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1\bar{x}_2 \\
f_{x_1} &= 1 \\
f_{\bar{x}_1} &= x_2x_3 + x_2\bar{x}_3 + \bar{x}_2 \\
f_{\bar{x}_1x_2} &= x_3 + \bar{x}_3 \\
f_{\bar{x}_1x_2x_3} &= 1 \\
f_{\bar{x}_1x_2\bar{x}_3} &= 1 \\
f_{\bar{x}_1\bar{x}_2} &= 1
\end{aligned}
\qquad \text{(A.1)}
$$

This cofactorisation order leads to 6 calculations, i.e. 6 nodes in the binary cofactor tree.

$$
\begin{aligned}
f_{x_3} &= x_1 + \bar{x}_1x_2 + \bar{x}_1\bar{x}_2 \\
f_{x_3x_2} &= x_1 + \bar{x}_1 \\
f_{x_3x_2x_1} &= 1 \\
f_{x_3x_2\bar{x}_1} &= 1 \\
f_{x_3\bar{x}_2} &= x_1 + \bar{x}_1 \\
f_{x_3\bar{x}_2x_1} &= 1 \\
f_{x_3\bar{x}_2\bar{x}_1} &= 1 \\
f_{\bar{x}_3} &= x_1 + \bar{x}_1x_2 + \bar{x}_1\bar{x}_2 \\
f_{\bar{x}_3x_2} &= x_1 + \bar{x}_1 \\
f_{\bar{x}_3x_2x_1} &= 1 \\
f_{\bar{x}_3x_2\bar{x}_1} &= 1 \\
f_{\bar{x}_3\bar{x}_2} &= x_1 + \bar{x}_1 \\
f_{\bar{x}_3\bar{x}_2x_1} &= 1 \\
f_{\bar{x}_3\bar{x}_2\bar{x}_1} &= 1
\end{aligned}
\qquad \text{(A.2)}
$$

For this order of the cofactorisation variables, 14 calculations were needed to obtain the same result.

Appendix B.

Results of equivalence checking with different SELECT routines.

Table B.1.
General properties of the test files.

File name	#inputs	#outputs	depth
5xp1.2222.gf	7	10	6
5xp1.dec.gf			6
5xp1.inp			0
9sym.2222.gf	9	1	10
9sym.dec.gf			9
9sym.inp			0
alu2.2222.gf	10	8	6
alu2.dec.gf			6
alu2.inp			0
alu3.dec.gf	10	8	7
alu3.inp			0
alu3.sim.gf			6
apla.2222.gf	10	12	7
apla.dec.gf			7
apla.inp			0
bw.2222.gf	5	28	5
bw.dec.gf			5
bw.inp			0
co14.2222.gf	14	1	6
co14.dec.gf			7
co14.inp			0
con1.2222.gf	6	2	3
con1.dec.gf			3
con1.inp			0
dc1.2222.gf	4	7	3
dc1.dec.gf			3
dc1.inp			0
dc2.2222.gf	7	7	6
dc2.dec.gf			7
dc2.inp			0
dk17.2222.gf	10	11	6
dk17.dec.gf			6
dk17.inp			0
dk27.2222.gf	9	9	5
dk27.dec.gf			5
dk27.inp			0
duke2.2222.gf	18	29	8
duke2.dec.gf			9
duke2.inp			0
f2.2222.gf	4	4	3
f2.dec.gf			2
f2.inp			0

Continued on next page.

Table B.1. Continued

File name	#inputs	#outputs	depth
in6.2222.gf in6.dec.gf in6.inp	17	23	7 7 0
in7.2222.gf in7.dec.gf in7.inp	21	10	8 6 0
misex1.2222.gf misex1.dec.gf misex1.inp	7	7	5 4 0
misex2.2222.gf misex2.dec.gf misex2.inp	24	11	9 8 0
misg.2222.gf misg.dec.gf misg.inp	15	23	6 6 0
mish.2222.gf mish.dec.gf mish.inp	8	21	5 4 0
radd.2222.gf radd.dec.gf radd.inp	8	5	6 6 0
rd53.2222.gf rd53.dec.gf rd53.inp	5	3	5 5 0
rd73.2222.gf rd73.dec.gf rd73.inp	7	3	8 8 0
rd84.2222.gf rd84.dec.gf rd84.inp	8	4	7 8 0
risc.2222.gf risc.dec.gf risc.inp	7	31	4 3 0
sao2.2222.gf sao2.dec.gf sao2.inp	10	4	8 7 0
sqn.2222.gf sqn.dec.gf sqn.inp	7	3	7 6 0
vg2.2222.gf vg2.dec.gf vg2.inp	25	8	7 8 0
wim.2222.gf wim.dec.gf wim.inp	4	7	4 3 0

Table B.2.
Results with select version 1.

file name		SELECT1.0	
file A	file B	#leaves	time (s)
5xp1.2222.gf	5xp1.dec.gf	746	43.3
5xp1.2222.gf	5xp1.inp	832	28.4
5xp1.dec.gf	5xp1.inp	1034	30.3
9sym.2222.gf	9sym.dec.gf	666	455.2
9sym.2222.gf	9sym.inp	792	258.7
9sym.dec.gf	9sym.inp	870	228.8
alu2.2222.gf	alu2.dec.gf	1806	94.4
alu2.2222.gf	alu2.inp	1806	64.1
alu2.dec.gf	alu2.inp	1088	134.1
alu3.dec.gf	alu3.inp	3052	39.5
alu3.dec.gf	alu3.sim.gf	3052	89.9
alu3.inp	alu3.sim.gf	2070	91.8
apla.2222.gf	apla.dec.gf	1142	95.3
apla.2222.gf	apla.inp	1142	55.2
apla.dec.gf	apla.inp	1085	52.5
bw.2222.gf	bw.dec.gf	738	33.5
bw.2222.gf	bw.inp	958	26.5
bw.dec.gf	bw.inp	1024	25.8
co14.2222.gf	co14.dec.gf	210	39.5
co14.2222.gf	co14.inp	210	22.1
co14.dec.gf	co14.inp	210	14.6
con1.2222.gf	con1.dec.gf	110	3.6
con1.2222.gf	con1.inp	110	2.5
con1.dec.gf	con1.inp	110	2.4
dc1.2222.gf	dc1.dec.gf	124	4.2
dc1.2222.gf	dc1.inp	124	3.0
dc1.dec.gf	dc1.inp	152	3.0
dc2.2222.gf	dc2.dec.gf	457	26.6
dc2.2222.gf	dc2.inp	458	16.1
dc2.dec.gf	dc2.inp	389	14.7
dk17.2222.gf	dk17.dec.gf	533	34.8
dk17.2222.gf	dk17.inp	554	22.6
dk17.dec.gf	dk17.inp	573	22.4
dk27.2222.gf	dk27.dec.gf	260	17.0
dk27.2222.gf	dk27.inp	260	10.0
dk27.dec.gf	dk27.inp	236	9.4
duke2.2222.gf	duke2.dec.gf	80589	4857.0
duke2.2222.gf	duke2.inp	93320	2788.8
duke2.dec.gf	duke2.inp	50598	1246.1
f2.2222.gf	f2.dec.gf	80	1.9
f2.2222.gf	f2.inp	30	1.5
f2.dec.gf	f2.inp	20	1.4
in6.2222.gf	in6.dec.gf	92923	2374.9
in6.2222.gf	in6.inp	91942	1629.8
in6.dec.gf	in6.inp	78860	1396.7
in7.2222.gf	in7.dec.gf	689310	18089.3
in7.2222.gf	in7.inp	669979	13599.9
in7.dec.gf	in7.inp	559007	9190.3

Continued on next page.

Table B.2. Continued

file name		SELECT1.0	
file A	file B	#leaves	time (s)
misex1.2222.gf	misex1.dec.gf	252	9.6
misex1.2222.gf	misex1.inp	274	8.0
misex1.dec.gf	misex1.inp	294	7.2
misex2.2222.gf	misex2.dec.gf	4031	618.7
misex2.2222.gf	misex2.inp	4056	335.4
misex2.dec.gf	misex2.inp	4919	303.7
misg.2222.gf	misg.dec.gf	10842	376.2
misg.2222.gf	misg.inp	10850	286.6
misg.dec.gf	misg.inp	10842	302.2
mish.2222.gf	mish.dec.gf	334	10.9
mish.2222.gf	mish.inp	350	7.6
mish.dec.gf	mish.inp	334	6.9
radd.2222.gf	radd.dec.gf	706	40.7
radd.2222.gf	radd.inp	706	24.4
radd.dec.gf	radd.inp	826	30.1
rd53.2222.gf	rd53.dec.gf	146	7.4
rd53.2222.gf	rd53.inp	148	4.8
rd53.dec.gf	rd53.inp	148	5.5
rd73.2222.gf	rd73.dec.gf	598	68.9
rd73.2222.gf	rd73.inp	610	34.5
rd73.dec.gf	rd73.inp	610	53.1
rd84.2222.gf	rd84.dec.gf	1400	165.8
rd84.2222.gf	rd84.inp	1400	108.6
rd84.dec.gf	rd84.inp	1354	162.4
risc.2222.gf	risc.dec.gf	407	13.9
risc.2222.gf	risc.inp	464	10.6
risc.dec.gf	risc.inp	422	8.8
sao2.2222.gf	sao2.dec.gf	916	107.4
sao2.2222.gf	sao2.inp	1328	66.5
sao2.dec.gf	sao2.inp	1766	85.6
sqn.2222.gf	sqn.dec.gf	354	29.7
sqn.2222.gf	sqn.inp	354	19.9
sqn.dec.gf	sqn.inp	432	22.3
vg2.2222.gf	vg2.dec.gf	140722	6659.9
vg2.2222.gf	vg2.inp	142528	3952.1
vg2.dec.gf	vg2.inp	160403	5021.7
wim.2222.gf	wim.dec.gf	136	4.5
wim.2222.gf	wim.inp	136	3.2
wim.dec.gf	wim.inp	130	2.9

Table B.3
Results with select version 2.

file name		SELECT2.0		SELECT2.1		SELECT2.2	
file A	file B	#leaves	time (s)	#leaves	time (s)	#leaves	time (s)
5xp1.2222.gf	5xp1.dec.gf	498	28.8	606	32.8	408	25.7
5xp1.2222.gf	5xp1.inp	564	22.9	1192	37.8	384	17.0
5xp1.dec.gf	5xp1.inp	612	21.3	1146	32.9	374	13.9
9sym.2222.gf	9sym.dec.gf	582	386.0	834	508.0	496	319.7
9sym.2222.gf	9sym.inp	716	191.7	850	260.5	680	165.9
9sym.dec.gf	9sym.inp	724	181.1	862	215.7	754	195.8
alu2.2222.gf	alu2.dec.gf	1432	99.3	1602	97.1	802	59.5
alu2.2222.gf	alu2.inp	1740	82.5	2672	106.0	542	29.2
alu2.dec.gf	alu2.inp	1348	61.7	2404	84.1	536	26.7
alu3.dec.gf	alu3.inp	1158	37.0	3246	86.7	564	19.5
alu3.dec.gf	alu3.sim.gf	2328	101.2	2894	125.4	670	34.0
alu3.inp	alu3.sim.gf	2054	52.8	4052	115.9	370	18.7
apla.2222.gf	apla.dec.gf	1422	116.9	1470	120.7	954	90.3
apla.2222.gf	apla.inp	1212	61.1	1706	78.0	802	47.2
apla.dec.gf	apla.inp	1278	63.6	1538	70.7	874	48.6
bw.2222.gf	bw.dec.gf	884	39.3	886	39.8	616	31.2
bw.2222.gf	bw.inp	5058	30.5	1242	34.1	798	24.4
bw.dec.gf	bw.inp	5076	29.6	1184	31.6	832	23.5
co14.2222.gf	co14.dec.gf	210	45.5	210	41.2	210	42.7
co14.2222.gf	co14.inp	210	23.8	210	23.7	210	23.6
co14.dec.gf	co14.inp	210	16.7	210	15.0	210	16.4
con1.2222.gf	con1.dec.gf	126	4.5	130	4.2	64	2.3
con1.2222.gf	con1.inp	98	2.9	116	2.8	54	1.4
con1.dec.gf	con1.inp	98	2.7	110	2.7	54	1.4
dc1.2222.gf	dc1.dec.gf	130	5.7	150	5.2	96	3.6
dc1.2222.gf	dc1.inp	134	4.6	148	3.8	100	2.8
dc1.dec.gf	dc1.inp	106	3.2	132	2.9	94	2.3
dc2.2222.gf	dc2.dec.gf	515	33.5	571	33.2	385	24.3
dc2.2222.gf	dc2.inp	524	21.6	756	22.9	296	12.9
dc2.dec.gf	dc2.inp	441	16.9	593	20.2	323	13.4
dk17.2222.gf	dk17.dec.gf	684	44.4	634	42.0	565	37.9
dk17.2222.gf	dk17.inp	678	27.0	792	27.9	508	21.2
dk17.dec.gf	dk17.inp	695	26.3	755	27.8	499	20.2
dk27.2222.gf	dk27.dec.gf	310	21.0	312	18.7	282	17.3
dk27.2222.gf	dk27.inp	280	11.6	312	10.7	262	10.3
dk27.dec.gf	dk27.inp	302	11.8	308	10.4	232	9.2
duke2.2222.gf	duke2.dec.gf	164691	11095.3	166857	11878.0	3383	331.2
duke2.2222.gf	duke2.inp	539966	17533.8	874752	30937.2	2082	113.4
duke2.dec.gf	duke2.inp	364116	15198.8	410292	17418.7	2242	152.4
f2.2222.gf	f2.dec.gf	80	2.0	56	1.9	64	1.8
f2.2222.gf	f2.inp	30	1.5	80	1.6	80	1.6
f2.dec.gf	f2.inp	24	1.3	80	1.5	80	1.5
in6.2222.gf	in6.dec.gf	636397	15525.7	620723	15586.0	1692	131.9
in6.2222.gf	in6.inp	71343	1676.5	57147	1516.6	1930	75.0
in6.dec.gf	in6.inp	69986	1686.0	75268	1890.3	2008	82.6
in7.2222.gf	in7.dec.gf	4212968	95081.7	3980424	89873.7	1256	82.7
in7.2222.gf	in7.inp	1115019	25342.8	1467713	34207.4	801	41.2
in7.dec.gf	in7.inp	234827	4679.6	723063	15314.0	714	31.3

Continued on next page.

Table B.3 Continued

file name		SELECT2.0		SELECT2.1		SELECT2.2	
file A	file B	#leaves	time (s)	#leaves	time (s)	#leaves	time (s)
misex1.2222.gf	misex1.dec.gf	434	15.4	472	16.5	140	6.4
misex1.2222.gf	misex1.inp	580	13.4	762	16.9	150	4.7
misex1.dec.gf	misex1.inp	432	10.5	654	14.2	160	4.7
misex2.2222.gf	misex2.dec.gf	195521	11648.5	198971	12010.3	1637	260.3
misex2.2222.gf	misex2.inp	203836	6969.6	189542	6976.3	1328	143.1
misex2.dec.gf	misex2.inp	166229	6488.1	167237	6983.6	1191	109.3
misg.2222.gf	misg.dec.gf	14052	536.6	11134	580.8	824	48.5
misg.2222.gf	misg.inp	10636	274.7	4894	208.0	942	40.5
misg.dec.gf	misg.inp	10770	281.3	4886	217.6	900	38.8
mish.2222.gf	mish.dec.gf	362	10.1	362	10.9	180	5.7
mish.2222.gf	mish.inp	252	5.9	498	11.7	176	4.2
mish.dec.gf	mish.inp	236	5.5	482	10.9	176	4.1
radd.2222.gf	radd.dec.gf	606	35.2	618	43.5	414	26.6
radd.2222.gf	radd.inp	470	16.9	808	30.2	394	14.8
radd.dec.gf	radd.inp	454	19.2	796	34.5	396	17.7
rd53.2222.gf	rd53.dec.gf	138	8.2	146	9.1	136	7.5
rd53.2222.gf	rd53.inp	138	5.1	144	5.5	140	4.7
rd53.dec.gf	rd53.inp	140	5.6	142	6.1	144	5.6
rd73.2222.gf	rd73.dec.gf	582	64.5	598	74.2	584	65.7
rd73.2222.gf	rd73.inp	598	36.2	608	41.5	584	32.9
rd73.dec.gf	rd73.inp	600	52.7	610	55.5	586	50.9
rd84.2222.gf	rd84.dec.gf	1166	189.8	1210	187.4	1198	166.7
rd84.2222.gf	rd84.inp	1196	104.0	1224	107.0	1190	100.3
rd84.dec.gf	rd84.inp	1224	147.2	1234	142.2	1242	145.1
risc.2222.gf	risc.dec.gf	388	13.3	421	14.3	315	11.4
risc.2222.gf	risc.inp	472	10.9	594	13.1	336	8.6
risc.dec.gf	risc.inp	364	8.3	422	9.1	284	7.0
sao2.2222.gf	sao2.dec.gf	980	101.4	2316	231.0	666	79.1
sao2.2222.gf	sao2.inp	734	47.8	3232	178.1	476	35.8
sao2.dec.gf	sao2.inp	580	44.8	2918	132.0	444	39.5
sqn.2222.gf	sqn.dec.gf	352	29.7	430	34.5	308	27.1
sqn.2222.gf	sqn.inp	386	19.9	526	26.5	248	14.8
sqn.dec.gf	sqn.inp	408	19.2	568	24.9	234	13.4
vg2.2222.gf	vg2.dec.gf	305834	17516.9	355772	20800.8	6501	454.2
vg2.2222.gf	vg2.inp	226768	8410.2	485632	16544.5	3082	138.6
vg2.dec.gf	vg2.inp	281072	10176.4	516878	18740.9	2639	129.0
wim.2222.gf	wim.dec.gf	122	4.2	152	4.9	90	3.6
wim.2222.gf	wim.inp	114	3.0	116	3.0	104	2.8
wim.dec.gf	wim.inp	120	2.9	140	3.1	92	2.3

Table B.4.
Results with select version 3.

file name		SELECT3.0		SELECT3.1	
file A	file B	#leaves	time (s)	#leaves	time (s)
5xp1.2222.gf	5xp1.dec.gf	360	24.0	364	25.0
5xp1.2222.gf	5xp1.inp	396	17.3	426	18.5
5xp1.dec.gf	5xp1.inp	410	14.5	350	13.4
9sym.2222.gf	9sym.dec.gf	568	374.7	586	381.1
9sym.2222.gf	9sym.inp	640	176.7	658	145.6
9sym.dec.gf	9sym.inp	760	192.3	818	209.6
alu2.2222.gf	alu2.dec.gf	784	54.5	724	52.2
alu2.2222.gf	alu2.inp	818	39.9	730	38.0
alu2.dec.gf	alu2.inp	602	28.1	614	29.6
alu3.dec.gf	alu3.inp	678	22.4	402	16.7
alu3.dec.gf	alu3.sim.gf	670	35.0	608	30.7
alu3.inp	alu3.sim.gf	498	19.6	484	18.5
apla.2222.gf	apla.dec.gf	932	87.0	854	83.9
apla.2222.gf	apla.inp	996	52.5	882	46.9
apla.dec.gf	apla.inp	938	50.4	848	48.8
bw.2222.gf	bw.dec.gf	598	30.6	590	34.8
bw.2222.gf	bw.inp	858	24.9	818	28.4
bw.dec.gf	bw.inp	880	23.4	848	28.3
co14.2222.gf	co14.dec.gf	210	41.5	210	44.2
co14.2222.gf	co14.inp	210	22.1	210	24.5
co14.dec.gf	co14.inp	210	15.2	210	16.9
con1.2222.gf	con1.dec.gf	52	1.9	44	1.9
con1.2222.gf	con1.inp	46	1.2	42	1.2
con1.dec.gf	con1.inp	48	1.4	42	1.2
dc1.2222.gf	dc1.dec.gf	94	3.6	94	4.0
dc1.2222.gf	dc1.inp	92	2.7	94	2.9
dc1.dec.gf	dc1.inp	96	2.3	96	2.5
dc2.2222.gf	dc2.dec.gf	367	23.9	445	27.6
dc2.2222.gf	dc2.inp	370	14.4	382	16.1
dc2.dec.gf	dc2.inp	331	13.7	425	18.1
dk17.2222.gf	dk17.dec.gf	519	34.7	498	40.3
dk17.2222.gf	dk17.inp	504	20.4	460	21.4
dk17.dec.gf	dk17.inp	525	20.3	531	23.3
dk27.2222.gf	dk27.dec.gf	318	18.7	312	20.1
dk27.2222.gf	dk27.inp	304	10.9	292	12.4
dk27.dec.gf	dk27.inp	296	10.4	290	11.1
duke2.2222.gf	duke2.dec.gf	2713	275.6	2479	248.1
duke2.2222.gf	duke2.inp	4460	181.0	1952	104.4
duke2.dec.gf	duke2.inp	2898	180.2	2230	145.0
f2.2222.gf	f2.dec.gf	64	1.8	64	1.8
f2.2222.gf	f2.inp	72	1.4	64	1.4
f2.dec.gf	f2.inp	72	1.4	64	1.3
in6.2222.gf	in6.dec.gf	1573	126.0	1467	112.7
in6.2222.gf	in6.inp	1492	67.2	1115	53.9
in6.dec.gf	in6.inp	1501	70.8	1277	61.2
in7.2222.gf	in7.dec.gf	980	69.1	918	59.3
in7.2222.gf	in7.inp	1046	47.6	697	37.7
in7.dec.gf	in7.inp	1011	41.3	503	25.9

Continued on next page.

Table B.4. Continued

file name		SELECT3.0		SELECT3.1	
file A	file B	#leaves	time (s)	#leaves	time (s)
misex1.2222.gf	misex1.dec.gf	134	6.2	140	6.4
misex1.2222.gf	misex1.inp	182	5.2	148	4.5
misex1.dec.gf	misex1.inp	162	4.6	142	4.4
misex2.2222.gf	misex2.dec.gf	1801	281.4	1559	246.1
misex2.2222.gf	misex2.inp	1770	162.1	1606	162.7
misex2.dec.gf	misex2.inp	1705	127.0	1469	117.3
misg.2222.gf	misg.dec.gf	818	45.2	760	40.0
misg.2222.gf	misg.inp	590	28.0	638	27.8
misg.dec.gf	misg.inp	624	28.7	624	27.0
mish.2222.gf	mish.dec.gf	174	5.5	176	5.3
mish.2222.gf	mish.inp	184	4.4	184	4.3
mish.dec.gf	mish.inp	168	4.1	168	3.8
radd.2222.gf	radd.dec.gf	414	25.0	394	24.0
radd.2222.gf	radd.inp	664	22.8	434	16.9
radd.dec.gf	radd.inp	444	20.0	362	17.6
rd53.2222.gf	rd53.dec.gf	140	7.5	146	7.4
rd53.2222.gf	rd53.inp	140	4.6	148	4.8
rd53.dec.gf	rd53.inp	148	5.6	148	5.6
rd73.2222.gf	rd73.dec.gf	582	63.4	606	61.2
rd73.2222.gf	rd73.inp	582	33.3	612	33.3
rd73.dec.gf	rd73.inp	582	51.4	612	53.5
rd84.2222.gf	rd84.dec.gf	1166	161.8	1332	160.2
rd84.2222.gf	rd84.inp	1220	97.4	1300	100.1
rd84.dec.gf	rd84.inp	1208	142.2	1310	146.5
risc.2222.gf	risc.dec.gf	302	11.1	304	10.9
risc.2222.gf	risc.inp	376	9.2	330	8.4
risc.dec.gf	risc.inp	292	7.1	274	6.8
sao2.2222.gf	sao2.dec.gf	648	76.9	584	69.6
sao2.2222.gf	sao2.inp	718	47.2	568	36.4
sao2.dec.gf	sao2.inp	670	47.0	660	43.6
sqn.2222.gf	sqn.dec.gf	274	25.6	268	23.8
sqn.2222.gf	sqn.inp	268	15.9	256	15.0
sqn.dec.gf	sqn.inp	276	15.1	258	14.3
vg2.2222.gf	vg2.dec.gf	6247	445.3	3583	230.8
vg2.2222.gf	vg2.inp	7608	338.0	2710	130.1
vg2.dec.gf	vg2.inp	21134	798.9	2925	143.2
wim.2222.gf	wim.dec.gf	90	3.5	88	3.5
wim.2222.gf	wim.inp	94	2.7	90	2.5
wim.dec.gf	wim.inp	92	2.3	86	2.2

Table B.5.
Results with select version 5.

file name		SELECT5.0		SELECT5.1	
file A	file B	#leaves	time (s)	#leaves	time (s)
5xp1.2222.gf	5xp1.dec.gf	414	26.5	634	33.4
5xp1.2222.gf	5xp1.inp	472	19.9	700	24.5
5xp1.dec.gf	5xp1.inp	686	23.9	948	27.6
9sym.2222.gf	9sym.dec.gf	604	392.1	694	450.6
9sym.2222.gf	9sym.inp	724	222.8	762	214.6
9sym.dec.gf	9sym.inp	828	253.7	864	226.6
alu2.2222.gf	alu2.dec.gf	1002	78.2	1542	94.9
alu2.2222.gf	alu2.inp	1074	56.7	1544	64.6
alu2.dec.gf	alu2.inp	774	40.3	956	40.5
alu3.dec.gf	alu3.inp	2678	85.5	2900	76.4
alu3.dec.gf	alu3.sim.gf	2010	86.9	2716	105.2
alu3.inp	alu3.sim.gf	674	45.4	1522	56.9
apla.2222.gf	apla.dec.gf	1096	126.1	1138	103.2
apla.2222.gf	apla.inp	1098	68.5	1138	60.4
apla.dec.gf	apla.inp	1040	68.2	1026	56.7
bw.2222.gf	bw.dec.gf	654	39.6	682	34.7
bw.2222.gf	bw.inp	888	31.2	930	27.6
bw.dec.gf	bw.inp	946	30.3	996	27.2
co14.2222.gf	co14.dec.gf	210	53.3	210	42.2
co14.2222.gf	co14.inp	210	31.7	210	24.1
co14.dec.gf	co14.inp	210	20.4	210	15.9
con1.2222.gf	con1.dec.gf	72	3.6	110	3.8
con1.2222.gf	con1.inp	72	2.3	110	2.8
con1.dec.gf	con1.inp	86	2.7	110	2.6
dc1.2222.gf	dc1.dec.gf	106	5.1	112	4.2
dc1.2222.gf	dc1.inp	110	4.0	112	3.1
dc1.dec.gf	dc1.inp	122	3.6	132	3.0
dc2.2222.gf	dc2.dec.gf	375	32.3	423	27.3
dc2.2222.gf	dc2.inp	374	19.6	426	16.6
dc2.dec.gf	dc2.inp	437	20.8	381	15.5
dk17.2222.gf	dk17.dec.gf	495	46.1	525	37.3
dk17.2222.gf	dk17.inp	530	27.7	546	22.6
dk17.dec.gf	dk17.inp	543	28.6	557	22.5
dk27.2222.gf	dk27.dec.gf	252	20.9	260	17.0
dk27.2222.gf	dk27.inp	256	12.4	260	10.2
dk27.dec.gf	dk27.inp	228	10.6	236	9.6
duke2.2222.gf	duke2.dec.gf	57843	4008.4	62989	4581.0
duke2.2222.gf	duke2.inp	75634	2513.6	76614	2658.6
duke2.dec.gf	duke2.inp	27974	738.2	34640	963.0
f2.2222.gf	f2.dec.gf	64	1.9	64	1.9
f2.2222.gf	f2.inp	72	1.5	80	1.5
f2.dec.gf	f2.inp	72	1.5	80	1.5
in6.2222.gf	in6.dec.gf	8843	408.7	19811	739.1
in6.2222.gf	in6.inp	7082	205.3	19502	460.0
in6.dec.gf	in6.inp	4575	156.5	41631	966.2
in7.2222.gf	in7.dec.gf	117544	2847.6	378478	11771.0
in7.2222.gf	in7.inp	117476	2201.7	402048	9105.3
in7.dec.gf	in7.inp	114086	1869.1	257572	4706.8

Continued on next page.

Table B.5. Continued

file name		SELECT5.0		SELECT5.1	
file A	file B	#leaves	time (s)	#leaves	time (s)
misex1.2222.gf	misex1.dec.gf	156	7.0	188	8.1
misex1.2222.gf	misex1.inp	194	5.4	230	6.5
misex1.dec.gf	misex1.inp	266	6.9	270	7.2
misex2.2222.gf	misex2.dec.gf	3379	532.0	3717	571.5
misex2.2222.gf	misex2.inp	3580	304.6	3964	357.5
misex2.dec.gf	misex2.inp	4649	319.9	4775	335.1
misg.2222.gf	misg.dec.gf	10208	344.1	10842	383.7
misg.2222.gf	misg.inp	10216	255.0	10842	288.7
misg.dec.gf	misg.inp	10150	252.2	10834	290.6
mish.2222.gf	mish.dec.gf	180	5.8	198	6.3
mish.2222.gf	mish.inp	196	4.7	242	5.6
mish.dec.gf	mish.inp	180	4.4	226	5.3
radd.2222.gf	radd.dec.gf	674	40.0	706	40.9
radd.2222.gf	radd.inp	674	24.0	690	24.5
radd.dec.gf	radd.inp	794	29.9	810	30.7
rd53.2222.gf	rd53.dec.gf	146	7.7	146	7.6
rd53.2222.gf	rd53.inp	148	4.8	148	4.8
rd53.dec.gf	rd53.inp	148	5.7	148	5.7
rd73.2222.gf	rd73.dec.gf	606	63.2	600	65.1
rd73.2222.gf	rd73.inp	612	34.1	610	35.7
rd73.dec.gf	rd73.inp	612	52.8	610	53.9
rd84.2222.gf	rd84.dec.gf	1254	159.4	1400	169.4
rd84.2222.gf	rd84.inp	1240	101.4	1390	106.3
rd84.dec.gf	rd84.inp	1340	147.1	1354	150.0
risc.2222.gf	risc.dec.gf	397	13.9	403	13.9
risc.2222.gf	risc.inp	454	10.8	460	11.0
risc.dec.gf	risc.inp	418	9.5	418	9.5
sao2.2222.gf	sao2.dec.gf	878	105.7	862	105.2
sao2.2222.gf	sao2.inp	988	59.3	1116	64.1
sao2.dec.gf	sao2.inp	894	53.6	1272	72.1
sqn.2222.gf	sqn.dec.gf	310	27.9	324	29.3
sqn.2222.gf	sqn.inp	322	17.5	326	18.1
sqn.dec.gf	sqn.inp	390	18.7	400	19.5
vg2.2222.gf	vg2.dec.gf	35230	1984.5	59634	3087.2
vg2.2222.gf	vg2.inp	37400	1248.5	99552	2985.9
vg2.dec.gf	vg2.inp	34963	1483.5	111011	3787.6
wim.2222.gf	wim.dec.gf	108	4.1	132	4.7
wim.2222.gf	wim.inp	108	2.9	136	3.4
wim.dec.gf	wim.inp	106	2.7	118	3.0

Table B.6.
Results with select version 6.

file name		SELECT6.0	
file A	file B	#leaves	time (s)
5xp1.2222.gf	5xp1.dec.gf	746	38.2
5xp1.2222.gf	5xp1.inp	796	26.2
5xp1.dec.gf	5xp1.inp	962	27.4
9sym.2222.gf	9sym.dec.gf	666	436.9
9sym.2222.gf	9sym.inp	792	248.6
9sym.dec.gf	9sym.inp	834	225.4
alu2.2222.gf	alu2.dec.gf	1810	100.2
alu2.2222.gf	alu2.inp	1810	69.9
alu2.dec.gf	alu2.inp	1088	40.1
alu3.dec.gf	alu3.inp	3052	79.6
alu3.dec.gf	alu3.sim.gf	3052	124.3
alu3.inp	alu3.sim.gf	2052	73.8
apla.2222.gf	apla.dec.gf	1146	103.4
apla.2222.gf	apla.inp	1146	56.6
apla.dec.gf	apla.inp	1058	53.7
bw.2222.gf	bw.dec.gf	738	34.4
bw.2222.gf	bw.inp	1008	28.7
bw.dec.gf	bw.inp	1048	27.6
co14.2222.gf	co14.dec.gf	210	41.3
co14.2222.gf	co14.inp	210	22.8
co14.dec.gf	co14.inp	210	14.8
con1.2222.gf	con1.dec.gf	110	3.6
con1.2222.gf	con1.inp	110	2.5
con1.dec.gf	con1.inp	110	2.5
dc1.2222.gf	dc1.dec.gf	124	4.4
dc1.2222.gf	dc1.inp	124	3.1
dc1.dec.gf	dc1.inp	152	3.2
dc2.2222.gf	dc2.dec.gf	457	27.2
dc2.2222.gf	dc2.inp	458	16.6
dc2.dec.gf	dc2.inp	389	15.3
dk17.2222.gf	dk17.dec.gf	531	34.9
dk17.2222.gf	dk17.inp	564	21.6
dk17.dec.gf	dk17.inp	573	21.2
dk27.2222.gf	dk27.dec.gf	260	16.1
dk27.2222.gf	dk27.inp	260	9.8
dk27.dec.gf	dk27.inp	236	9.1
duke2.2222.gf	duke2.dec.gf	80635	4986.4
duke2.2222.gf	duke2.inp	81828	2871.3
duke2.dec.gf	duke2.inp	38968	1193.5
f2.2222.gf	f2.dec.gf	80	2.2
f2.2222.gf	f2.inp	80	1.5
f2.dec.gf	f2.inp	80	1.5
in6.2222.gf	in6.dec.gf	92923	2505.6
in6.2222.gf	in6.inp	91948	1747.5
in6.dec.gf	in6.inp	78860	1511.9
in7.2222.gf	in7.dec.gf	689310	18840.7
in7.2222.gf	in7.inp	669979	13923.5
in7.dec.gf	in7.inp	558559	9304.1

Continued on next page.

Table B.6. Continued

file name		SELECT6.0	
file A	file B	#leaves	time (s)
misex1.2222.gf	misex1.dec.gf	252	9.4
misex1.2222.gf	misex1.inp	274	7.0
misex1.dec.gf	misex1.inp	292	7.2
misex2.2222.gf	misex2.dec.gf	4031	565.2
misex2.2222.gf	misex2.inp	4056	317.0
misex2.dec.gf	misex2.inp	4919	300.3
misg.2222.gf	misg.dec.gf	10842	363.6
misg.2222.gf	misg.inp	10850	273.1
misg.dec.gf	misg.inp	10842	275.0
mish.2222.gf	mish.dec.gf	334	9.6
mish.2222.gf	mish.inp	350	7.4
mish.dec.gf	mish.inp	336	7.2
radd.2222.gf	radd.dec.gf	706	39.4
radd.2222.gf	radd.inp	706	23.8
radd.dec.gf	radd.inp	826	29.5
rd53.2222.gf	rd53.dec.gf	146	7.5
rd53.2222.gf	rd53.inp	148	5.0
rd53.dec.gf	rd53.inp	148	5.6
rd73.2222.gf	rd73.dec.gf	598	63.9
rd73.2222.gf	rd73.inp	610	36.1
rd73.dec.gf	rd73.inp	610	53.4
rd84.2222.gf	rd84.dec.gf	1400	163.6
rd84.2222.gf	rd84.inp	1400	107.7
rd84.dec.gf	rd84.inp	1362	146.5
risc.2222.gf	risc.dec.gf	407	13.5
risc.2222.gf	risc.inp	464	10.6
risc.dec.gf	risc.inp	422	9.1
sao2.2222.gf	sao2.dec.gf	916	100.9
sao2.2222.gf	sao2.inp	1116	67.2
sao2.dec.gf	sao2.inp	1522	79.7
sqn.2222.gf	sqn.dec.gf	354	29.0
sqn.2222.gf	sqn.inp	354	18.2
sqn.dec.gf	sqn.inp	432	19.4
vg2.2222.gf	vg2.dec.gf	140722	6021.9
vg2.2222.gf	vg2.inp	142528	3856.8
vg2.dec.gf	vg2.inp	160633	4834.5
wim.2222.gf	wim.dec.gf	136	4.6
wim.2222.gf	wim.inp	136	3.3
wim.dec.gf	wim.inp	132	3.0

Appendix C.

Results of equivalence checking with different FLAT routines.

Table C.1.
Results flattened functions with select version 3.1.

file name		FLAT1.0		FLAT2.0	
file A	file B	#leaves	time (s)	#leaves	time (s)
5xp1.2222.gf	5xp1.dec.gf	308	19.5	396	30.1
5xp1.2222.gf	5xp1.inp	346	14.3	448	20.5
5xp1.dec.gf	5xp1.inp	332	12.0	360	19.2
9sym.2222.gf	9sym.dec.gf	442	204.4	466	243.9
9sym.2222.gf	9sym.inp	602	126.0	658	145.2
9sym.dec.gf	9sym.inp	682	149.0	790	199.9
alu2.2222.gf	alu2.dec.gf	552	40.0	740	53.9
alu2.2222.gf	alu2.inp	518	26.7	648	30.9
alu2.dec.gf	alu2.inp	518	22.6	538	28.0
alu3.dec.gf	alu3.inp	346	13.8	302	12.5
alu3.dec.gf	alu3.sim.gf	342	21.4	302	20.6
alu3.inp	alu3.sim.gf	344	15.1	302	15.3
apla.2222.gf	apla.dec.gf	766	66.0	850	144.6
apla.2222.gf	apla.inp	786	39.9	836	45.3
apla.dec.gf	apla.inp	762	37.8	782	104.8
bw.2222.gf	bw.dec.gf	558	29.3	550	43.4
bw.2222.gf	bw.inp	764	24.7	772	30.9
bw.dec.gf	bw.inp	786	21.9	782	26.7
co14.2222.gf	co14.dec.gf	210	29.0	210	31.5
co14.2222.gf	co14.inp	210	16.7	210	17.7
co14.dec.gf	co14.inp	210	12.8	210	14.0
con1.2222.gf	con1.dec.gf	38	1.5	42	3.3
con1.2222.gf	con1.inp	38	1.1	40	2.1
con1.dec.gf	con1.inp	38	1.0	40	2.0
dc1.2222.gf	dc1.dec.gf	94	3.5	98	4.4
dc1.2222.gf	dc1.inp	94	2.8	96	3.1
dc1.dec.gf	dc1.inp	92	2.3	94	2.7
dc2.2222.gf	dc2.dec.gf	363	20.3	417	24.7
dc2.2222.gf	dc2.inp	342	12.7	386	15.0
dc2.dec.gf	dc2.inp	335	13.3	395	15.0
dk17.2222.gf	dk17.dec.gf	426	28.2	430	38.5
dk17.2222.gf	dk17.inp	448	16.6	478	25.5
dk17.dec.gf	dk17.inp	409	17.7	425	19.8
dk27.2222.gf	dk27.dec.gf	252	13.8	248	15.1
dk27.2222.gf	dk27.inp	252	8.6	268	9.9
dk27.dec.gf	dk27.inp	236	8.0	234	8.7
duke2.2222.gf	duke2.dec.gf	1787	168.1	1835	413.9
duke2.2222.gf	duke2.inp	1822	96.5	1856	162.1
duke2.dec.gf	duke2.inp	1850	117.9	2058	313.1
f2.2222.gf	f2.dec.gf	48	1.5	48	1.6
f2.2222.gf	f2.inp	64	1.3	48	1.2
f2.dec.gf	f2.inp	48	1.0	48	1.1

Continued on next page.

Table C.1. Continued

file name		FLAT1.0		FLAT2.0	
file A	file B	#leaves	time (s)	#leaves	time (s)
in6.2222.gf	in6.dec.gf	1273	94.6	1233	304.5
in6.2222.gf	in6.inp	1087	48.7	1076	93.8
in6.dec.gf	in6.inp	1133	52.5	1076	240.0
in7.2222.gf	in7.dec.gf	614	46.8	846	557.5
in7.2222.gf	in7.inp	501	31.2	723	398.1
in7.dec.gf	in7.inp	383	22.3	431	193.3
misex1.2222.gf	misex1.dec.gf	126	6.0	126	9.7
misex1.2222.gf	misex1.inp	142	4.5	140	5.4
misex1.dec.gf	misex1.inp	140	4.2	140	6.7
misex2.2222.gf	misex2.dec.gf	1375	184.7	1435	346.1
misex2.2222.gf	misex2.inp	1396	117.9	1460	222.6
misex2.dec.gf	misex2.inp	1237	102.4	1317	120.6
misg.2222.gf	misg.dec.gf	602	40.4	624	34.1
misg.2222.gf	misg.inp	444	22.0	642	28.5
misg.dec.gf	misg.inp	370	20.5	400	21.1
mish.2222.gf	mish.dec.gf	142	5.5	142	5.4
mish.2222.gf	mish.inp	158	4.6	158	4.2
mish.dec.gf	mish.inp	142	4.1	154	4.1
radd.2222.gf	radd.dec.gf	480	25.4	418	31.4
radd.2222.gf	radd.inp	392	16.5	418	18.4
radd.dec.gf	radd.inp	440	18.5	330	20.0
rd53.2222.gf	rd53.dec.gf	136	6.9	146	8.9
rd53.2222.gf	rd53.inp	142	4.5	148	4.6
rd53.dec.gf	rd53.inp	136	5.5	136	7.2
rd73.2222.gf	rd73.dec.gf	606	61.8	606	58.0
rd73.2222.gf	rd73.inp	612	35.6	612	34.0
rd73.dec.gf	rd73.inp	608	53.9	612	47.2
rd84.2222.gf	rd84.dec.gf	1288	145.9	1302	126.2
rd84.2222.gf	rd84.inp	1226	104.2	1288	96.8
rd84.dec.gf	rd84.inp	1288	135.0	1300	125.2
risc.2222.gf	risc.dec.gf	280	10.8	276	15.2
risc.2222.gf	risc.inp	328	8.3	324	9.7
risc.dec.gf	risc.inp	274	7.7	274	9.5
sao2.2222.gf	sao2.dec.gf	450	52.6	492	59.5
sao2.2222.gf	sao2.inp	486	37.3	534	34.8
sao2.dec.gf	sao2.inp	456	41.0	502	34.7
sqn.2222.gf	sqn.dec.gf	236	22.0	260	81.5
sqn.2222.gf	sqn.inp	244	15.6	264	14.4
sqn.dec.gf	sqn.inp	240	13.0	232	69.1
vg2.2222.gf	vg2.dec.gf	3169	234.6	2043	169.4
vg2.2222.gf	vg2.inp	2738	129.6	1912	71.3
vg2.dec.gf	vg2.inp	2267	121.8	2633	177.1
wim.2222.gf	wim.dec.gf	94	3.4	92	4.1
wim.2222.gf	wim.inp	90	2.5	92	2.8
wim.dec.gf	wim.inp	86	2.3	92	2.7

REFERENCES

1. Brayton, Robert K. Hachtel, Gary D. McMullen, Curtis T. Sangiovanni-Vincentelli, Alberto L. "Logic Minimisation Algorithms for VLSI Synthesis". Kluwer Academic Publishers 1984.

2. Hachtel, Gary D. and Jacoby, Reily M. "Verification Algorithms for VLSI Synthesis". Design Systems for VLSI Circuits. Logic Synthesis and Silicon Compilation. Edited by G. de Micheli, A. Sangiovanni-Vincentelli and P. Antognetti. Nato ASI Series 1987. Page 249 - 300.

3. Hachtel, G. D. and Jacoby, R. M. "Algorithms for multi-level tautology and equivalence". Proceedings of IEEE International symposium on circuits and systems. Kyoto, Japan June 1985.

4. Brayton, R. K. Cohen, J. D. Hachtel, G. D. Trager, B. M. and Yun, D. Y. Y. "Fast recursive boolean function manipulation". Proceedings, International symposium on circuits and systems. Rome, Italy May 1982.