UC Irvine
ICS Technical Reports

Title
Technology mapping for register transfer descriptions

Permalink
https://escholarship.org/uc/item/4886b051

Authors
Rundensteiner, Elke A.
Gajski, Dan
Bic, Lubomir

Publication Date
1989

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ARCHIVES
Z
699
C3
no. 89-42
c. 2

Technology Mapping for Register Transfer Descriptions

Elke A. Rundensteiner, Dan Gajski and Lubomir Bic

Department of Information and Computer Science
University of California, Irvine
December, 1989

Technical Report 89-42

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

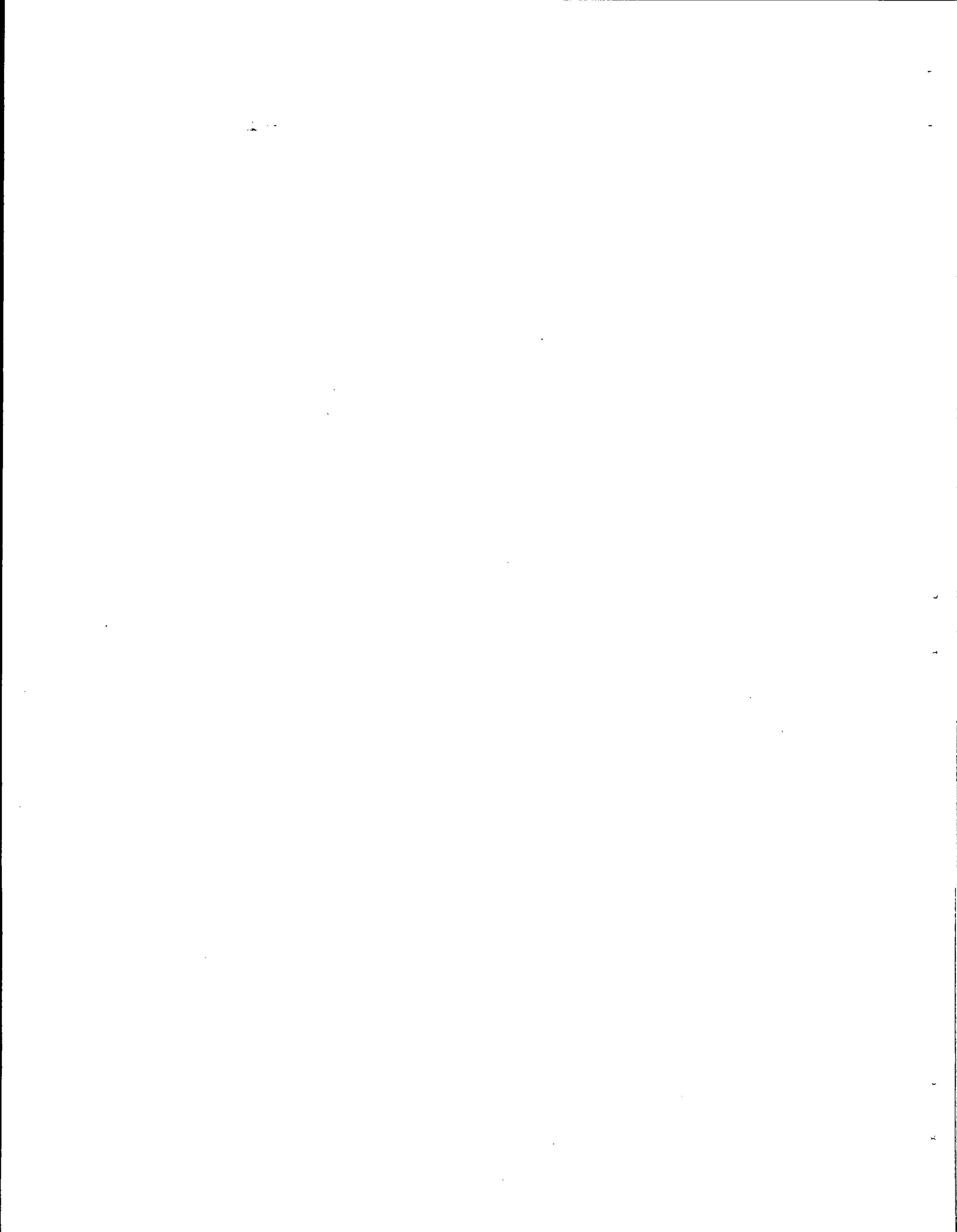
Technology Mapping for Register Transfer Descriptions

Elke A. Rundensteiner, Dan Gajski and Lubomir Bic

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Abstract. The use of VHDL as a hardware description language for automated synthesis has given rise to new problems. A behavioral description of components is given by using standard operators in the language. Therefore, a mismatch between the operators of the language and the functionalities provided by library components arises. In addition, the language does not guarantee uniqueness of descriptions, thus allowing possibly many different ways of describing same design. In this paper we propose a solution to these problems. The Component Synthesis Algorithm (CSA) recognizes a possibly incomplete behavioral description and generates a minimal set of components from a given library. In particular, CSA maps a complex behavioral description of a unit to one hardware component whenever possible. This process, driven by a particular library, emphasizes resource sharing between mutually exclusive operations that are mergable, i.e., that can be performed by the same component.

Key Words: Register-Level Synthesis and Optimization, Component Modeling, Compatibility Property.



Contents

1 Introduction 1

2 CSA Overview 2

3 The Technology Mapping Problem 6

3.1 The Input Description 6

3.2 The Design Representation 6

3.3 The Underlying Hardware Style 9

3.4 The Cost Table 12

3.5 The Cost Function 14

4 Exploiting Fundamental Properties 15

4.1 The Mutual Exclusiveness Property 15

4.2 The Mergability Property 20

4.3 The Compatibility Property 21

5 The CSA Algorithm 24

5.1 The Unit Merging Process 24

5.2 Evaluating the Quality of an Operator Merge 28

5.3 Functionality Recognizer 30

5.4 The CSA Algorithm 31

6 Experiments 34

7 Conclusion 40

A Appendix A **44**

A.1 Introduction 44

A.2 Running CSA 45

A.2.1 The Input Description 45

A.2.2 The CSA Execution 45

A.2.3 Output Files 47

A.3 The CSA Data Structures 49

A.3.1 Compatibility Edge List 49

A.3.2 Compatibility Edges 49

A.3.3 **Others** Table 50

A.3.4 Unit table 50

A.3.5 Listing of the CSA Data Structures 50

A.4 The CSA Algorithm 52

A.4.1 The Pre-processing Phase 52

A.4.2 The Merging Phase 56

A.4.3 The Post-processing Phase 57

A.5 Interrelationships of CSA Modules 59

List of Figures

1 The CSA Block Diagram 4

2 Example Input Description Schema 7

3 Flow Graph of The Example Input Description 8

4 Unit Merging at the Flow Graph Level 9

5 Hardware Implementation of a Choose Value Node 10

6 Hardware Implementation of an Operator Merge 11

7 Table of Costs of Available Units 13

8 Design Description Using Selected Signal Assignment Statements . . . 16

9 Examples of Operator Labels 18

10 Label Evaluation Algorithm 19

11 Use Compatibility Property to Merge Edges 22

12 Use Compatibility Property to Delete Edges 23

13 Reduce-Edges: Algorithm to Maintain the Compatibility Graph 24

14 Algorithm to Merge The Inputs of A Merged Operator Node 26

15 Example Cases of How to Merge The Inputs of A Merged Operator Node 27

16 The CSA Algorithm 32

17 ALU Description Taken from Mano (page 371) 36

18 Cost Functions for the Seven Design Examples 37

19 Number of Components Produced by CSA and Human Designer 38

20 Number of Components Produced by CSA and Human Designer 40

I A VHDL Description Using Conditional Signal Assignment Statements 53

II A VHDL Description Using Selected Signal Assignment Statements . . 55

1 Introduction

High-level synthesis involves mapping the behavioral description of a design to a structure composed of components from a given library that together execute that behavior [3, 7, 9, 12]. However, the use of general hardware description languages such as VHDL [4, 1] as basis for automated synthesis has given rise to new problems. For instance, behavioral VHDL descriptions of components, such as an Arithmetic/Logic Unit, may vary drastically. The problem is to define an algorithm that will generate the same optimal hardware from any description, no matter how imprecisely it is stated.

The reasons for such varied descriptions of one and the same design are manifold. First, the designer may not have at his disposal an exact description of the modeled component, secondly, many slightly variant descriptions of the component may be in circulation, and, lastly, a description language provides the designer with several ways of describing the same functionality. In other words, a hardware description language does generally not guarantee uniqueness of descriptions.

There are two, not necessarily disjoint, solutions to this problem. First, design modeling may be restricted to allow only synthesizable behavioral descriptions of real components. This is an important modeling issue since it has been realized that the quality of a design is directly related to the description style [2, 15, 5]. Second, an automated tool that synthesizes any description into an optimal hardware can be developed [11, 9, 12, 7, 3]. Such a tool must recognize several not necessarily identical descriptions and synthesize them into a minimal set of components. This task is further complicated by the fact that these descriptions are often incomplete, since a design at hand may not require all functionalities that could be provided by a particular component. Hence, the tool has to handle such an incomplete description.

In this paper we propose a solution to these problems by using the second approach. For this purpose, we have developed CSA, a Component Synthesis Algorithm. CSA

determines the number and type of processing units needed to implement a given description while attempting to minimize the underlying hardware implementation costs. Mutually exclusive operations that are mergable according to a chosen library are merged into multi-functional operator nodes. The goal of CSA is to find an optimal grouping of arithmetic operations within each such multi-functional operator node. A multi-functional operator node corresponds to a collection of mutually exclusive operations that are to be executed by one hardware unit, and therefore, a direct mapping to hardware can then take place.

The proposed algorithm is driven by the set of available hardware components, but is nevertheless *technology-independent*. All library-specific information is kept in declarative format separate from the rest of the system. Therefore, it is easily replaceable by another library. This separation allows for *redesign* of a given algorithmic description using new libraries.

The paper is organized as follows. While Section 1 motivates our work, Section 2 introduces the Component Synthesis Algorithm and its environment. In Section 3, we describe the hardware technology mapping problem. Particular properties of the problem that lead to the proposed solution are discussed in Section 4, while the actual algorithm is given in Section 5. Finally, we present the results of running several experiments in Section 6 and conclusions in Section 7.

2 CSA Overview

This section gives an overview of the CSA algorithm. A more detailed description of the problem and the proposed algorithm is given in later sections.

Figure 1 shows the environment of CSA, the Component Synthesis Algorithm. There are two parts to the system, the invariant part is depicted on the left hand side and the library-specific one on the right hand side of the figure. This separation makes

the system technology-independent, as it allows to replace the variant portion of the system by the description of another library. Thus it supports the retargetting of a design to distinct libraries.

The variant portion: This portion contains all information about the chosen component library. It has a fixed format but its content is exchangeable. This library-specific information has to be designed once for each technology. This can be done automatically by parsing an input library description into two tables, the functionality table and the mergability table. The functionality table describes the different functions that can be executed by components of the chosen input library, and it names them uniquely. The mergability table specifies the respective list of functions supported by each component of the input library. It also gives cost estimates (gate-count) for each component.

The invariant portion: This part consists of several subsystems. First, a language input compiler [5] is used as a frontend to parse a behavioral description into an internal representation, a Control Flow/Data Flow Graph (CFDF) [8]. The operation nodes of the data flow graph correspond to generic operations of VHDL, such as, +, -, etc.

The compatibility graph creation algorithm further augments the flow graph by compatibility edges. This is done by inserting compatibility edges between any pair of operator nodes that meets the following two requirements. First, the two operator nodes have to be mutually exclusive with respect to the behavioral description, and second, their functions have to be implementable by the same component. We refer to this characteristic as being *compatible.* In short, the flow graph is transformed into a compatibility graph (CG) where an edge between two operator nodes indicates their compatibility.

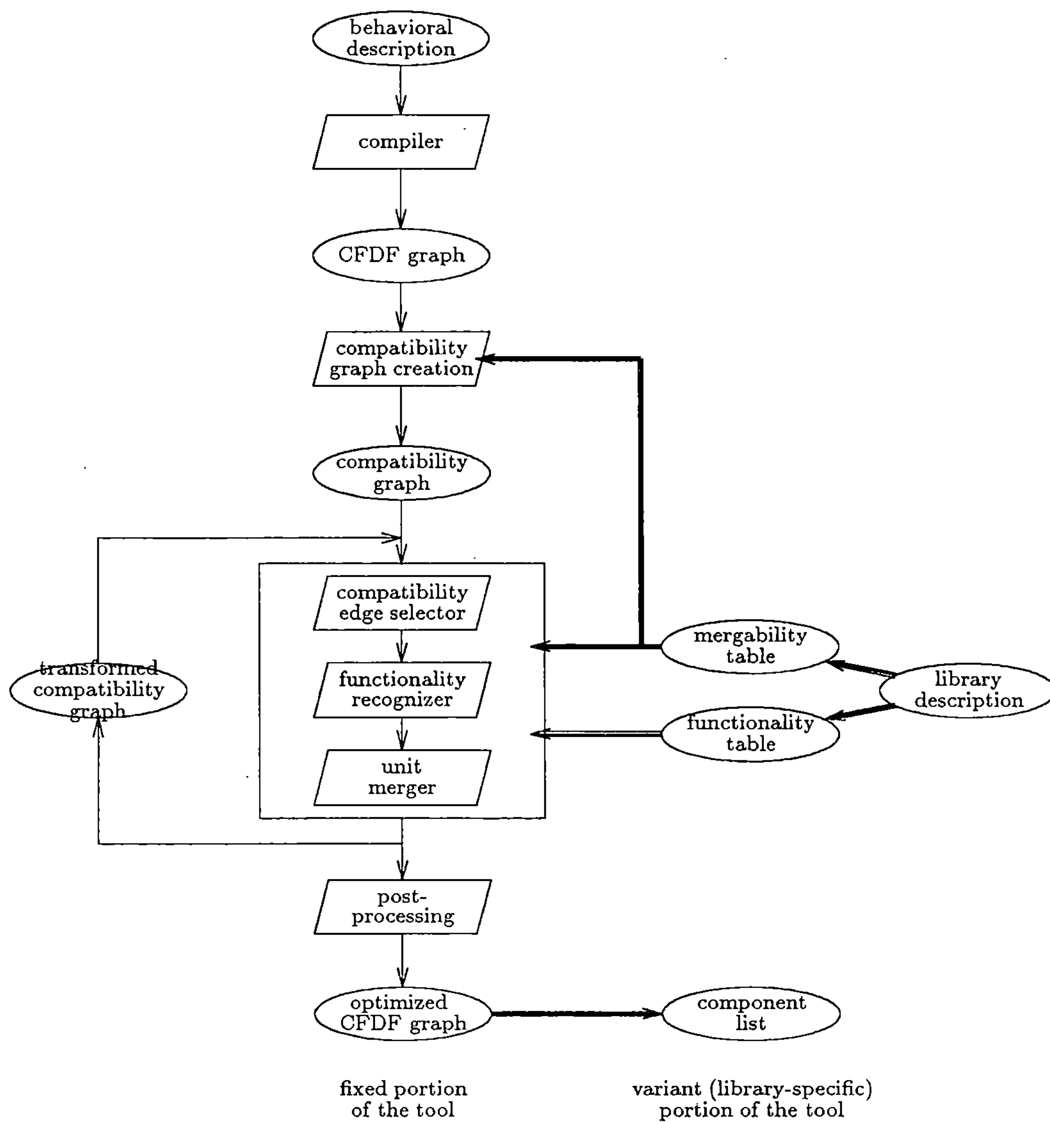


Figure 1: The CSA Block Diagram

CSA utilizes particular characteristics of this graph for guiding its search for an optimal design. In particular, the compatibility edge selector chooses the pair of operator nodes that is to be merged next by evaluating the quality of the compatibility edges. This evaluation is based on cost functions that take into account features of the compatibility graph as well as the operator costs provided by the mergability table. The overall goal of this process is to optimize the groupings of functions within operator nodes.

The fourth subsystem, the functionality recognizer, solves the *mismatch problem* between the behavioral description and the available library functions. Functions supported by a particular library may sometimes correspond to an expression (tree) of the input description instead of just one single operation (node). The functionality recognizer exploits these more complex functionalities by matching patterns describing library functions (as captured in the functionality table) against the flow graph. When a match is found and is favorable, the subgraph structure of the graph is replaced by one new operator node that models the library function. For instance, the flow graph structure for the input description "A + B + 1", which originally is compiled into two connected operation nodes labeled + with three inputs A, B and 1, may be replaced by an operator node labeled ADD-INC and two inputs A and B. Thus, the functionality recognizer optimizes the design by transforming graphs of generic operator nodes into graphs consisting of library-supported operator nodes.

The unit merging algorithm, on the other hand, transforms the compatibility graph by building *multi-functional operator nodes* from existing operator nodes. This process meets the two previously discussed requirements of mutually exclusiveness and mergability since it merges operator nodes have been selected accordingly. The unit merging algorithm assures the semantic equivalence of the input flow graph and the transformed flow graph by inserting choose value nodes that select the correct inputs of

multi-functional operator nodes. It also handles the function select of multi-functional operator nodes by encoding function select conditions and storing them in these nodes.

Lastly, a post-processor translates the compatibility graph back into a flow graph by deleting any remaining compatibility edges. Final output of CSA is a flow graph with operator nodes maximally merged according to the cost criteria captured in the mergability table.

3 The Technology Mapping Problem

3.1 The Input Description

The input description to CSA is a behavioral description of a design. The current prototype of CSA uses VHDL as input hardware description language, however, VHDL can easily be replaced by another language as long as the input compiler is modified accordingly. An example of a description schema that represents a typical input description for CSA is shown in Figure 2.

The description in Figure 2 consists of two nested conditional statements. The two statements, "target1 <= ..." and "target2 <= ..." are to be executed concurrently. On the other hand, the statements labeled by values v_{ij} following a condition-i correspond to different branches of the condition, that is, they are disjoint and therefore only one of them will be executed. The order of operations within each expression enforces partial sequentiality.

3.2 The Design Representation

The input description is translated by a compiler [5] into an internal flow graph representation, a CFDF graph [8]. Figure 3 depicts the internal representation of the description schema shown in Figure 2.

This mapping obeys the following rules:

```
target1 <=
  (condition-1
    (v11: expression1)
    (v12: expression2)
    (v13:
      (condition-2
        (v21: expression3)
        (v22: expression4)
      )
    )
    (v14: expression5)
  )
target2 <=
  (condition-1
    (v11: expression6)
    (v12: expression7)
  )
```

Figure 2: Example Input Description Schema

(1) Expression-i is mapped to a data flow graph composed of operation nodes n_{ij} as internal nodes and variable access nodes r_i as external nodes. These nodes are represented by circles and boxes in the graph of Figure 3, respectively. They are connected by data flow edges.

(2) The CFDF model also provides a choose value construct which is a flow graph equivalent to a multiplexer (represented by a triangle in Figure 3). A choose value node is used to distinguish between inputs to an operator node when there is more than one. Note that choose value nodes allow to include the connection costs at the flow graph level. Each condition statement, condition-i, is mapped to one such choose value node, CV_i. Therefore, there are three choose value nodes in Figure 3. The condition of such a statement forms the guard to the choose value node. The operation node

at the root of the expression tree that represents the condition generates the control signal for the choose value node, thus it is also called a *conditional node.* The branch v_{ij} of a given condition-i is mapped to a separate input port, v_{ij}, of the respective CV_i node.

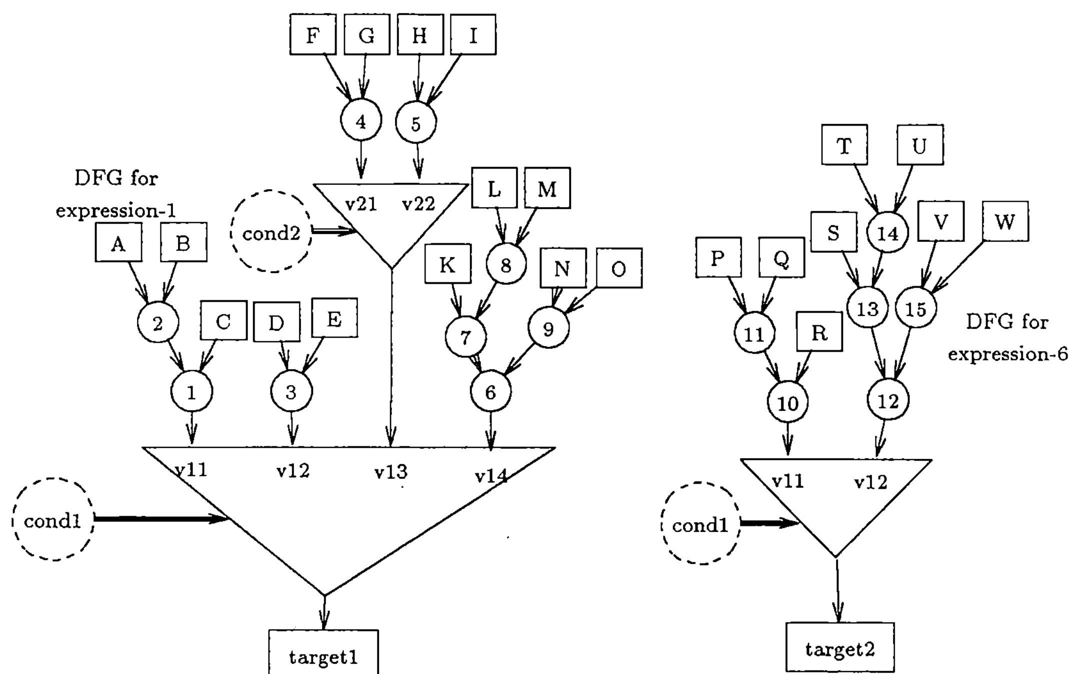


Figure 3: Flow Graph of The Example Input Description

The data flow graph for each expression-i forms a tree with its root anchored in the choose value v_{ij} of the respective choose value node CV_i. These trees are composed to form a forest, with one large tree per assignment statement.

The CFDF model has been extended to include a *multi-functional operator node*, also called a *controllable operator node.* A controllable operator node corresponds to a data flow node construct with a list of operations attached to it. A multi-functional node is depicted on the right side of Figure 4 by a double circle, labeled by an ordered list of functions. As shown in Figure 4 each function is annotated by the condition v_i

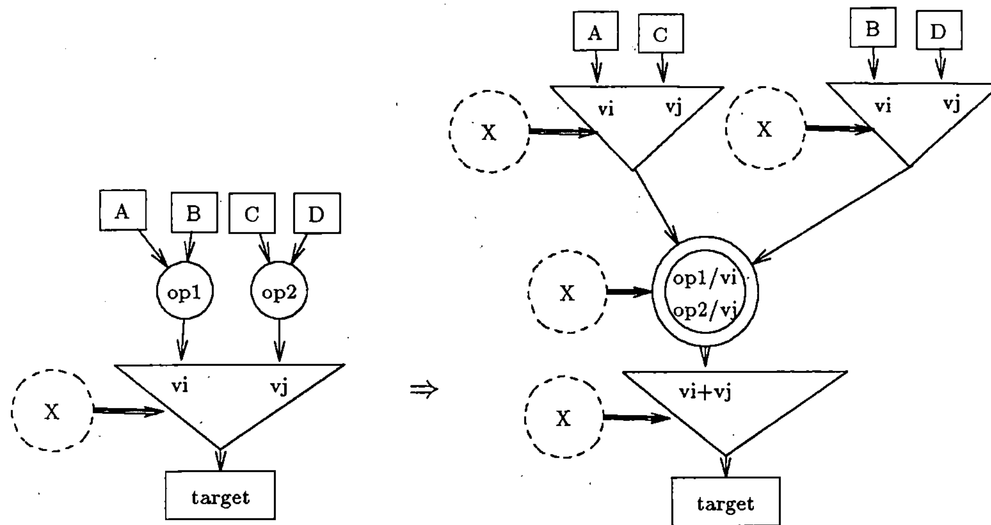


Figure 4: Unit Merging at the Flow Graph Level

under which the respective function is to be executed. For example, in Figure 4 the operations op1 and op2 are executed under the conditions vi and vj, respectively.

The *multi-functional operation node* supports operator merging at the flow graph level. It allows us to perform flow graph transformations by merging operation nodes that are mutually exclusive and thus can be implemented by a single hardware unit into one such node. A example of such a merge for two operation nodes is shown in Figure 4. We place the restriction that only operation nodes that are mutually exclusive can be merged into the same multi-functional operator node.

3.3 The Underlying Hardware Style

The goal of our research is to transform a data flow graph (DFG) as shown in Figure 3 into a hardware implementation of minimal cost under the constraint of operator compatibility. In the following, we describe the chosen hardware style, i.e., the hardware components by which the flow graph constructs are assumed to be implemented.

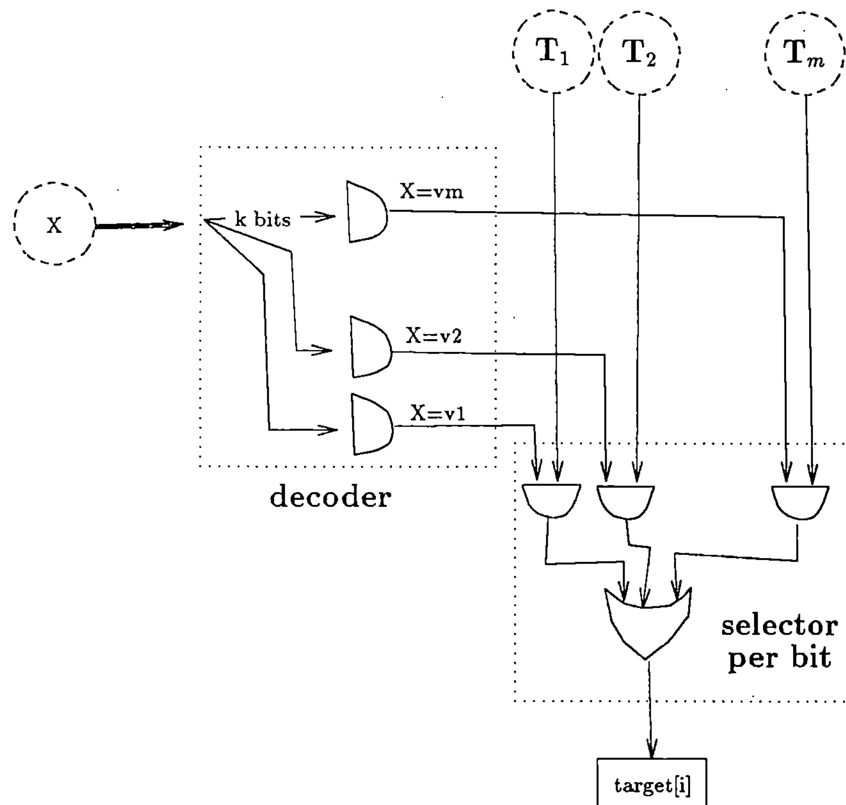


Figure 5: Hardware Implementation of a Choose Value Node

The assumed hardware style for a choose value node CV is depicted in Figure 5. A choose value node CV of size m is implemented by a combination of a special decoder and a special multiplexer, which together form a selector. The decoder decodes its input X into a subset of choose values v_i instead of all possible $2^{|X|}$ values (minterms) as shown in Figure 5. The multiplexer chooses one of the m inputs based on the result of the associated decoder. The implementation of the choose value node of Figure 3 is given in Figure 5.

An operator node is implemented by a functional unit. The cost for a functional unit depends on several factors, such as, the number and type of required function-

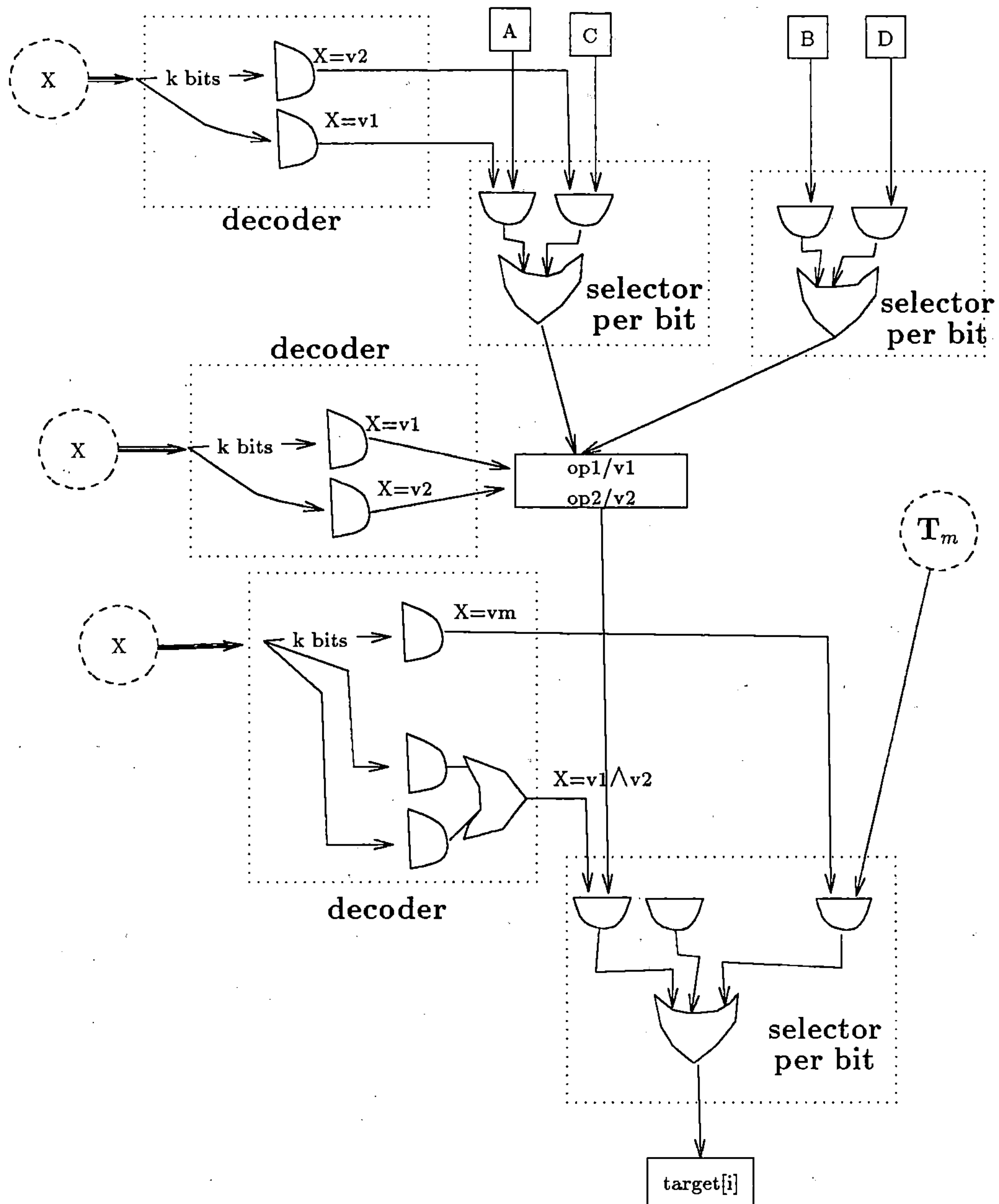


Figure 6: Hardware Implementation of an Operator Merge

alities, bit widths of the input arguments. This is treated in more depth in the next section. The hardware implementation of a multi-functional operator node corresponds to a functional unit and a decoder that selects among its functions. For this, see for instance Figure 6 which shows the hardware implementation of the flow graph depicted in Figure 4.

3.4 The Cost Table

The specification of costs is very important, since it guides the search for mergable operator nodes. Ideally, the hardware costs should reflect the exact layout costs of the data paths. This is infeasible, however, and thus we approximate the costs of an operator node implementation by gate counts. For this, we use a unit table that specifies the costs of individual modules (hardware operators). An example unit table is shown in Figure 7. The cost of an individual arithmetic or boolean operator is easily determined. But since we attempt to optimally group operators within units like an ALU, we also need cost estimates for multi-functional units. An ALU is assumed to be implemented using combinational logic. Hence, the area required by a set of operators is, in general, not equal to the sum of the areas required to implement each operator separately. For instance, an ALU implementing addition and subtraction (in our example unit u3) is only slightly more costly than an ALU implementing only addition (unit u1) and equal to implementing only subtraction (unit u2). Since costs cannot be calculated using simple additive relationships, we provide a unit table that defines costs for typical collections of operators. The table contains a unique name for each grouping of operations, a list of all functions implemented by the given unit (called **functionality**), and the cost for each unit in terms of gate counts per bit (called **op-cost**).

unit name	functionality	op-cost
u1	+	5
u2	-	6
u3	+ and -	6
u4	INC	5
u5	DEC	5
u6	both counting functions	5
u7	one logic function	1
u8	two logic functions	5
u9	all 16 logic functions plus transfer	9
u10	one shifting function	7
u11	all shifting functions { r, l, 0, 1 }	10
u12	$\times$	30
u13	/	30
u14	arithmetic, transfer and counting	10
u15	arithmetic and logic	12
u16	logic and counting	12
u17	arithmetic, logic and counting	13

Figure 7: Table of Costs of Available Units

3.5 The Cost Function

The cost function is fine-grained, i.e., it is at the level of gate counts instead of operator nodes. To approximate the cost of a design, given a flow graph representation, we consider only operation nodes and choose value nodes. As indicated in the previous section, the cost of an operation node, implemented by a functional unit, can be derived from a unit table. If the operator node is multi-functional, then a decoder is needed in addition to select the appropriate function as shown in Figure 6.

In Figure 5, we show the logic-level implementation of a choose value node. Note that choose value nodes allow us to include the connection costs into the cost calculations at the flow graph level. The increased control and wiring complexity that results from resource sharing is thus accounted for. We are now ready to put it all together to formulate the cost function.

Let n be the number of bits and let size(CVi) be the number of choices of a choose-value node CVi. Then the following holds:

1. The cost of an operation node n_i is expressed by **op-cost**(n_i) $\times$ n. **Op-cost**(n_i) is determined by table look-up into the cost unit table described in Section 3.4 in the following manner: **op-cost**(n_i) $= \min_{\text{units u}}$ { **op-cost**(u) | **functionality**(u) $\supseteq$ **functionality**(n_i) }.

2. Let size(n_i) represent the number of conditions under which the operator node n_i will be executed. Then, the cost of a decoder for a multi-functional operation node n_i is size(n_i).

3. By the hardware style shown in Figure 5, the cost of a choose value node CVi with m_i choices (i.e., size(CVi) = m_i) consists of the cost of a decoder, which is m_i, and of a selector which is is $m_i \times n$. Altogether, it results in $m_i \times (1 + n)$ = size(CVi) $\times(1 + n)$.

Thus, the total cost of a data flow graph is approximated by:

$$\text{total-cost} = \Sigma_{operator\ nodes\ n_i}(\ \mathbf{op\text{-}cost}(n_i) \times n + size(n_i))$$
$$+ \Sigma_{choose\ value\ nodes\ CV_i}(size(CV_i) \times (1+n)).$$

One goal of CSA is to minimize this total cost.

4 Exploiting Fundamental Properties

Before presenting the CSA algorithm, we analyze the most important problem characteristics which CSA exploits. They are mutual exclusiveness, mergability, and the compatibility property.

4.1 The Mutual Exclusiveness Property

Definition 1 *Two operator nodes n1 and n2 are* **mutually exclusive** *to each other, if the condition which selects one operation always falsifies the condition selecting the other, and vice versa.*

To determine whether two operator nodes n1 and n2 are **mutually exclusive**, we use a labeling scheme similar to the one presented in [9]. Labels that denote the conditions under which an operator node is executed are associated with each operator node. Once labels have been constructed, the labels of any two nodes can be compared in time proportional to the number of different conditions that appear in the input description to evaluate whether the corresponding nodes are mutually exclusive. First, we discuss how we deal with the situation of default values. Then, we describe the labeling scheme and outline an algorithm that constructs these labels. And finally, we discuss how two of these labels can be compared to evaluate whether the corresponding nodes are mutually exclusive.

Default Choose Values: The choose values of a condition are not always exhaustive. In fact, it is common to designate a default value to the last alternative of a condition. In VHDL, for instance, this is called the **others** value as can be seen in the example design description in Figure 8. The value **others** stands for all values that are

```
architecture dataflow of design-desc is
begin
 with cond1  select
  DATA-OUT1 <=
    DATA-A xor DATA-B     when  "0000",
    DATA-A or DATA-B      when  "0001",
    DATA-A and DATA-B     when  others;

 with cond1  select
  DATA-OUT2 <=
    DATA-A -  DATA-B      when  "0000",
    DATA-B +  DATA-B      when  "1111",
    DATA-A +  DATA-B + "0001"     when  others;
end dataflow;
```

Figure 8: Design Description Using Selected Signal Assignment Statements

not explicitly enumerated in the corresponding condition (in flow graph terminology this corresponds to all values that are not choose values of the corresponding choose value node). The meaning of each **others** value depends on its context, i.e., on the condition in which it occurs. Therefore, each occurrence of the **others** value has a possibly distinct meaning. As this context (a choose value node) may be modified or potentially removed during the unit merging phase, CSA replaces each occurrence of an **others** value by a unique value of the form **others<id>** where id is a unique integer number. For instance, the two **others** values in the example in Figure 8 will be replaced by the strings **others**1 and **others**2, respectively. The first **others** value, i.e., **others1**, corresponds to the 14 values in the range from "0001" to "1111". This can be represented by associating all 14 values with the **others**1 value, or, vice versa, by associating with it all values that it does not represent. For instance, **others**1 = $\neg$ "0000" $\wedge$ $\neg$ "0001". The later approach is preferable to the former one, since the number of possible values may not be finite for certain data types. On the other hand,

the number of values an **others** value does not represent is always finite as it corresponds to the values that are explicitly expressed in the condition. For the following, let $S_{\mathbf{others<i>}}$ denote the set of values that **others**$< i >$ does not represent. Then $v_j \in S_{\mathbf{others<i>}}$ means that the value v_j is not represented by the value **others**$< i >$. In the above example, $S_{\mathbf{others1}}$ = { "0001", "1111" }. CSA collects these **others<id>** values and their corresponding meaning, the set $S_{\mathbf{others<i>}}$, in a separate table. After this relabeling of the default values each value in the flow graph is defined without any ambiguities.

Node labeling scheme: Let there be a unique integer identifier, cond_i, for each operation node that generates the control signal for a choose value node (called a conditional node). Each choose value node that is created for the same condition is guarded by the same conditional node, and thus is annotated by the same identifier cond_i. For instance, in the flow graph in Figure 3 there are two conditional nodes called cond1 and cond2 but three different choose value nodes. Two of these choose value nodes are identified by the same condition cond_1. The label associated with an operation node consists of a list of pairs $(\text{cond1},v_{1j})(\text{cond2},v_{2j})$... where v_{ij} represents a choose value (including a value of the format **others**$< i >$) or the "don't care" value.

Node labeling algorithm: Initially, a template label L is constructed consisting of an ordered list of *all* condition identifiers cond_i of the data flow graph and "don't care" values. This approach simplifies unit merging considerably as all labels are of the same length and conditions are consistently ordered within the label. In depth-first traversal traverse the data flow graph from the targets upwards by traveling in direction opposite to the data flow edges The template label L is attached to every operation node when first visited. Incrementally modify the template label L by replacing the pair (cond_i, "don't care") by the pair (cond_i,v_{ij}) when traversing a choose value node identified by cond_i upwards through the input branch with the choose value v_{ij}. The corresponding template value is set back to "don't care" when descending the flow graph through that choose value node.

The number of different conditions found in the input description corresponds to the length of the labels in terms of integer pairs. The labels of the nodes directly above a choose value node have one more condition value not equal to "don't care" than those directly beneath the choose value node. Furthermore note that if two nodes are executed under the same condition, then they get assigned the same label.

Example 1 *In Figure 9, we list some labels derived by the node labeling algorithm for the operator nodes of Figure 3.*

node number	**label**
node1	*(cond1,v11)(cond2,"don't care")*
node2	*(cond1,v11)(cond2,"don't care")*
node3	*(cond1,v12)(cond2,"don't care")*
node4	*(cond1,v13)(cond2,v21)*
...	...

Figure 9: Examples of Operator Labels

This makes apparent that condition optimization is needed once the operation nodes have been merged.

Label Evaluation: Given the above labeling scheme, label evaluation has become trivial. The following describes how to decide whether two operator nodes n1 and n2 are **mutually exclusive** to each other given their labels. The labels label(n1) and label(n2) are compared by pairwise comparing their elements from left to right.

To summarize, the label evaluation algorithm pair-wise compare the subparts of the labels and if the choose values v_{ij} are truly distinct for a common condition then the operation nodes are *mutually exclusive*, otherwise they are not. Truly distinct means that either both values are regular choose values, or one of them is a default value **others<i>** with $S_{\mathbf{others<i>}}$ including the second value in its set. The latter means that **others<i>** does not represent the second value, thus, they are mutually exclusive. We demonstrate this process by an example.

```
(cond,v1) := leftmost element of label(n1);
(cond,v2) := leftmost element of label(n2);
while (cond,v1) <> NIL (or (cond,v2) <> NIL) do
   if v1 and v2 are distinct and both are not equal to "don't care"
   then begin
      if v1= othersi and v2 is not of the form othersj and v2 ∈ S_othersi
      then stop and return mutually exclusive;
      if v1 is not of the form othersi and v2= othersj and v1 ∈ S_othersj
      then stop and return mutually exclusive;
      if v1 is not of the form othersi and v2 is not of the form othersj
      then stop and return mutually exclusive;
   end then;
   (cond,v1) := next element of label(n1);
   (cond,v2) := next element of label(n2);
end while;
return not mutually exclusive;
```

Figure 10: Label Evaluation Algorithm

Example 2 *Assume the flow graph depicted in Figure 3 and labels from the previous example. Then, for instance, node1 and node2 are not mutually exclusive, since both have the same labels (cond1,v11)(cond2,"don't care"). Node1 and node4 on the other hand are mutually exclusive, since their labels (cond1,v11)(cond2,"don't care") and (cond1,v13)(cond2,v21) have the same condition cond1 but different choose values v11 and v13.*

Proposition 1 *A collection of operator nodes n_i can only be merged into a multi-functional operator node if they all are mutually exclusive by Definition 1.*

Proposition 1 ensures that a multi-functional node models parts of the input description that can be assigned to the same hardware unit, as only one of the modeled functions will be executed during a given time step.

4.2 The Mergability Property

Definition 2 *An operator node n is* **legal** *with respect to a given unit table (e.g. the example unit table described in Section 3.4) if there is a unit u in the unit table with* **functionality***(u)* $\supseteq$ **functionality***(n).*

Definition 3 *Two operator nodes n1 and n2 are* **mergable** *with respect to a given unit table if and only if the merged operator node n := n1 $\cup$ n2 is* **legal** *with respect to the same unit table.*

By Definition 2, this implies that n1 and n2 are **mergable** if and only if there is a unit u in the unit table with **functionality**(u) $\supseteq$ **functionality**(n1) $\cup$ **functionality**(n2).

Definition 4 *A data flow graph DFG consists of three types of nodes, operator nodes, variable-access nodes, and choose value nodes. The edges, called data flow edges, are directed. We augment such a DFG by adding compatibility edges between operator nodes. Every pair of operator nodes of DFG that is mergable by Definition 3 and that*

is mutually exclusive by Definition 1 is connected by such an undirected compatibility edge. The augmented data flow graph is called a **compatibility graph.**

For simplicity, we may show a compatibility graph CG consisting of only compatibility edges and operator nodes with all other edges and nodes removed. A data flow graph DFG and its corresponding compatibility graph CG are presented in Figure 11 a) and b), respectively.

4.3 The Compatibility Property

Proposition 2 *A collection of operator nodes n_i can only be merged into one multi-functional operator node if they are pairwise compatible.*

Proposition 2 states that a compatible set of operator nodes must correspond to a subgraph of the compatibility graph that is completely connected by compatibility edges, i.e., it forms a clique. Proposition 2 establishes a *necessary* condition for merging a collection of operator nodes n_i into one operator node. The question remains whether the compatibility property also constitutes a *sufficient* condition for such merging.

Proposition 3 *The compatibility property is a sufficient condition for an operator merge due to the "completeness" of the application domain.*

This "completeness" may be reasoned as follows: Given three different functions f1, f2 and f3. If in a given technology the three units u1, u2, u3 with **functionality**(u1) = { f1, f2 }, **functionality**(u2) = { f1, f3 }, and **functionality**(u3) = { f2, f3 }, are manufactured then there will always also be a unit u4 which implements all three functions, i.e., **functionality**(u4) $\supset$ { f1, f2, f3 }. This "completeness" assumption is for instance true for the unit table presented in Figure 7, which is typical for most collections of hardware units.

Note that the just stated "completeness" assumption does *not* require "transitivity" of the domain. We don't assume that if in a given technology the two units u1 and u2 exist with **functionality**(u1) = { f1, f2 }, **functionality**(u2) = { f1, f3 },

then there also has to be a unit u3 with **functionality**(u3) = { f2, f3 }. It also says nothing about the costs associated with such a unit.

The previous proposition about the compatibility property of a multi-functional node implies the following.

Proposition 4 *Given an operator node n and a newly created multi-functional node M composed of the original operator nodes* n_1, n_2, ..., n_j *with* $n \neq n_i$ *for all i from 1 to j. Due to the compatibility property, a compatibility edge* $e(n,n_k)$ *(for some* $k \in \{1,...,j\}$*) can only be used for future merges if and only if the edges* $e(n,n_i)$ *exist in CG for all* $i \in \{1,...,j\}$*. Furthermore, since the operator nodes* n_1, n_2, ..., n_j *correspond to one multi-functional operator node M, the edges* $e(n,n_i)$ *for all i can be replaced by one edge, e(n,M).*

In other words, the compatibility property allows us to determine directly from the compatibility graph whether edges can be merged into one edge. We give an example of this next.

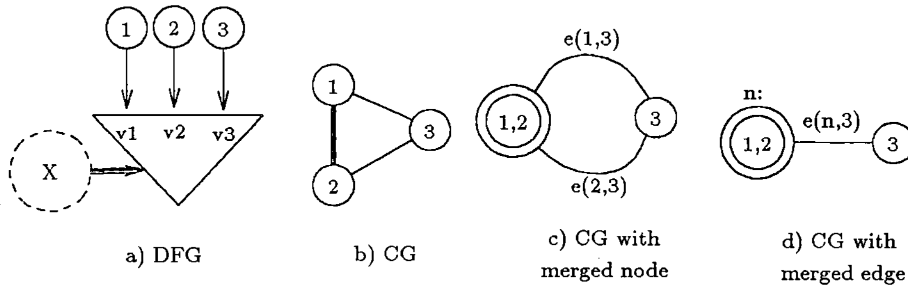


Figure 11: Use Compatibility Property to Merge Edges

Example 3 *Figure 11 a shows a DFG with three nodes n1, n2 and n3. Assume that they are compatible. Then, Figure 11 b shows the corresponding compatibility graph (CG). The three nodes are completely connected by compatibility edges e(n1,n2) and*

e(n1,n3) and e(n2,n3). If we decided to merge node n1 with node n2, then the edges e(n1,n3) and e(n2,n3) both connect the node n3 with the merged node n $:= (n1 \cup n2)$ *(Figure 11 c). By Proposition 4, they can be collapsed into one edge e(n,n3) as is done in Figure 11 d.*

The following proposition on edge reductions is a direct consequence of the previous proposition on the compatibility property. It can again be directly derivable from the compatibility graph.

Proposition 5 *Given an operator node n and a newly created multi-functional node M composed of the original operator nodes* $n_1, n_2, ..., n_j$ *with* $n \neq n_i$ *for all i. If there is one operator node* n_k *(for some* $k \in \{1, ..., j\}$*) for which no compatibility edge e(n,*n_k*) exists then none of the other edges e(n,*n_i*) for* $i \in \{1, ..., j\}$ *can be used for future merges. Thus, they have to be deleted.*

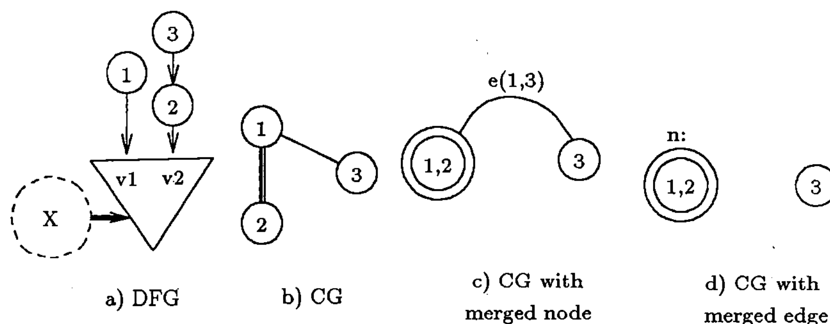


Figure 12: Use Compatibility Property to Delete Edges

Example 4 *Given a DFG and the corresponding compatibility graph (CG) in Figure 12 a and b, respectively. If we decided to merge node n1 with node n2, then e(n1,n3) can no longer be used to form a larger clique since the edge e(n2,n3) is missing (Figure 12 c). Therefore, by proposition 5 the edge e(n1,n3) has to be deleted (Figure 12 d).*

Let e(x,y) be the chosen merge edge connecting the operator nodes x and y. Let M be the resulting merge node.

```
procedure Reduce-Edges( e: node)
repeat
  if there is an edge e(x,n) and e(y,n) in CG
  then merge them into one new edge e(M,n);
  if there is an edge e(x,n) and  not e(y,n) in CG
  then delete the edge e(x,n);
  if there is an edge e(y,n) and  not e(x,n) in CG
  then delete the edge e(y,n);
until all edges adjacent to edge e have either been merged or deleted;
end procedure;
```

Figure 13: Reduce-Edges: Algorithm to Maintain the Compatibility Graph

To summarize, in this section we have presented a scheme by which various constraints on merges of nodes can be directly expressed by the compatibility graph concept. In particular, we have shown in propositions 4 and 5 how the structure of the compatibility graph can be maintained during the process of merging sets of operator nodes by removing and/or merging compatibility edges. This leads to the algorithm shown in Figure 13.

5 The CSA Algorithm

5.1 The Unit Merging Process

Below, we describe the flow graph transformation process for merging operator nodes. This is best explained with the help of an example.

Figure 4 demonstrates the process of merging the two compatible operator nodes n1 and n2 with functionalities op1 and op2. The two nodes n1 and n2 are merged into one multi-functional operator node, depicted in Figure 4 by a double circle. Call this new node n. As discussed in Section 4.1, the node n is annotated by the conditions

v_i under which its respective functions are to be executed. In Figure 4, for instance, the functionalities op1 and op2 are labeled by the conditions v1 and v2, respectively. As these conditions are associated with the functions of an operator node instead of the operator node itself; these conditions are preserved automatically when several functions are collected into one operator node.

Next, the output data flow edges of the new node have to be updated. They now have to connect to the original output destinations of n1 and n2. If n1 and n2 were directly connected to the same choose value node, as is the case in Figure 4, then the size of the choose value node is reduced by combining two of its inputs. In the example at hand, this corresponds to the fact that the two choose values v_i and v_j are merged into one value ($v_i + v_j$). If the choose value node becomes redundant, meaning, only one data input remains, then the node is deleted.

Finally, the input data flow edges to the new node n have to be adjusted to connect to the original inputs of n1 and n2. If the left (right) inputs of n1 and n2 were identical then the new node n is simply connected to that input node. If on the other hand the inputs are distinct, then a choose value node must be inserted to select among the inputs to the newly created multi-functional node n. This choose value node copies information about appropriate conditions from the condition labels of the node n1 and the node n2. The input port of the choose value node that is connected to the original input for node n1 will be guarded by a choose value that corresponds to the conditions associated with all functions of the operator node n1. Similarly, the input port that supplies the input data from the operator node n2 copies its choose value from node n2. These new choose value nodes are in the worst case of size k if k different function were combined in the resulting operation node n. The algorithm that merges the inputs of node n1 and n2 to generate correct inputs for node n is described in more detail in Figure 14. A graphical explanation of the five distinct cases that could occur is given in Figure 15.

Let node n1 and node n2 be the two merged operator nodes and let n be the resulting multi-functional node. Let Ii[n1] and Ii[n2] denote the i-th input nodes to n1 and n2, respectively.

```
repeat
                                                            /* case 1: */
  if Ii[n1] = Ii[n2]  then Ii[n1] is the new input for node n
  else begin
                                                            /* case 2: */
    if Ii[n1] is not a choose value node and Ii[n2] is not a choose value node
    then create a new choose value node CH with I1[CH] = n1 and I2[CH] = n2 and
     the choose values for input ports 1 and 2 are the conditions c1 and c2, respectively;
     the choose value node CH becomes the input node to the new node n.
                                                            /* case 3: */
    else if Ii[n1] is a choose value node and Ii[n2] is not a choose value node
    then begin
       Compare Ii[n2] with the different input nodes Ij[Ii[n1]] of Ii[n1];
       if Ii[n2] is equal to one of them, i.e., for some j: Ii[n2] = Ij[Ii[n1]]
       then add the condition c2 associated with node n2 to the j-th input port of Ii[n1];
                                                            /* case 4: */
       else if Ii[n2] is not equal to one of them,
       then create an additional input port k for Ii[n1] with Ik[Ii[n1]] := Ii[n2],
        and the choose value of that port corresponds to the condition c2.
    end if
    else if Ii[n1] is not a choose value node and Ii[n2] is a choose value node
    then do the same as described in cases 3 and 4 with n1 and n2 reversed.
                                                            /* case 5: */
    else if Ii[n1] and Ii[n2] are both choose value nodes
    then integrate their inputs to create one choose value node by doing the following:
     Merge each input Ik[Ii[n2]] of Ii[n2] with the choose value node Ii[n1] as in cases 3 and 4;
    Transfer the choose value of the k-th input port of Ii[n2] to the integrated choose value node.
   In cases 3, 4 and 5, Ii[n1] becomes the input node to the new node n.
  end if
until all input pairs Ii[n1] and Ii[n2] have been dealt with.
```

Figure 14: Algorithm to Merge The Inputs of A Merged Operator Node

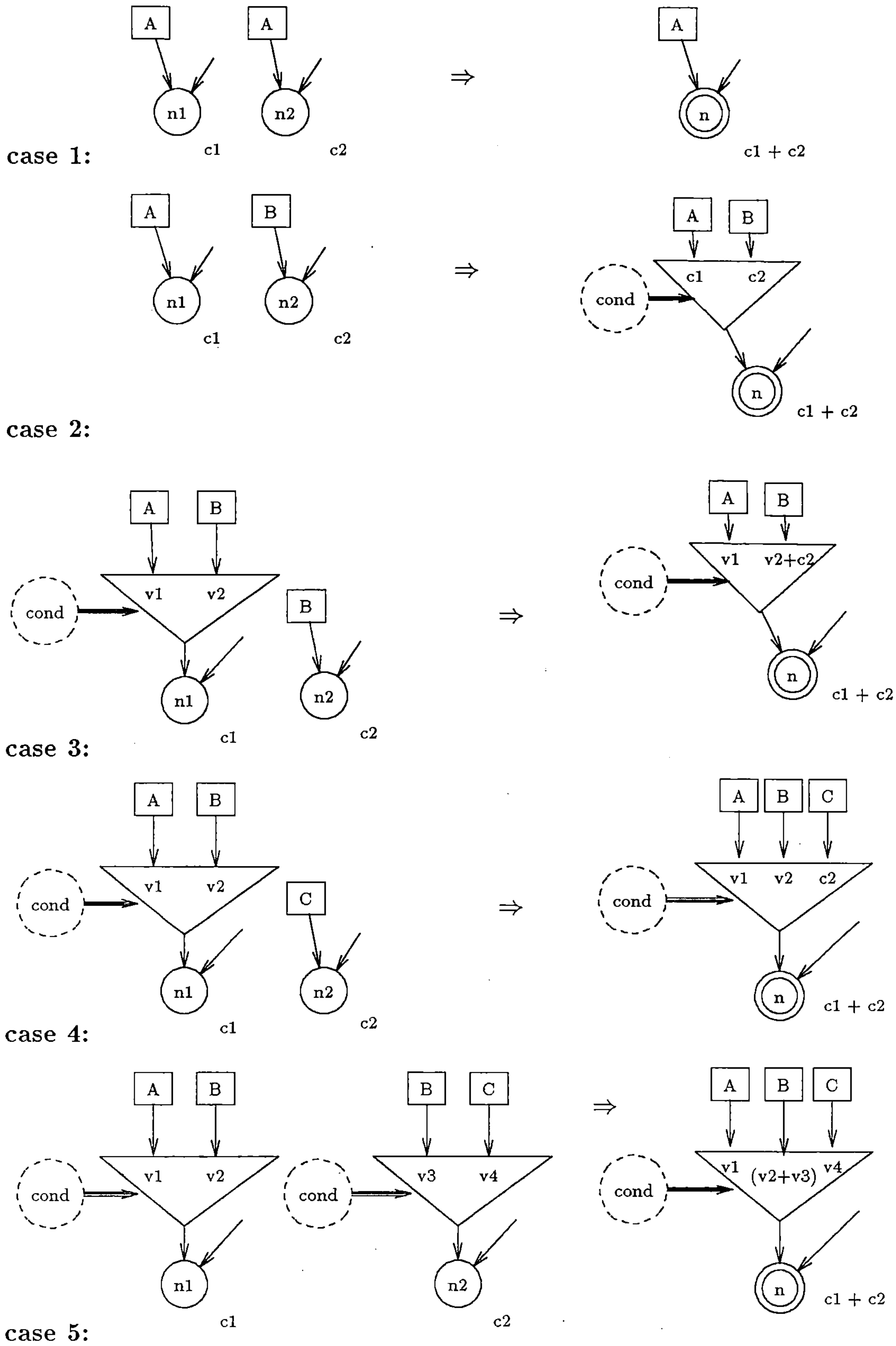


Figure 15: Example Cases of How to Merge The Inputs of A Merged Operator Node

The example depicted in Figure 4 and in Figure 6 shows clearly the trade-off involved in determining "optimal merges" of operator nodes. The example merge reduces, for instance, the hardware costs in two respects: (1) it reduces the number of operator units from two to one and (2) it reduces the size of the choose value node. However, new costs accrue in the form of two choose value nodes that distinguish between the inputs to the multi-functional operator nodes. In hardware, these correspond to two multiplexors. Also, a decoder to select among the two functions increases costs. To further complicate matters, note that these two choose value nodes do not always have to be introduced. They are only needed when the inputs of the merged operator nodes differ. Similarly, the size of the underlying choose value node is not always reduced as shown in our example. This only happens if the merged operator nodes anchor directly in the same choose value node. This discussion makes it apparent that CSA has to trade off between the costs and gains of each such merge.

5.2 Evaluating the Quality of an Operator Merge

Having presented the underlying cost assumptions in Sections 3.3, 3.4 and 3.5, and the unit merging process in Section 5.1, we are now in the position to evaluate the gain and/or cost of operator merging.

Definition 5 *The benefit of an operator merge of a set of nodes e, Benefit(e), is a function of the immediate gain in terms of hardware implementation costs and of the potential for future merges.*
Benefit(e) = p1×Merge-Costs(e) + p2×Mux-Costs(e) + p3×Ancestor-Mergability(e) where p1, p2, and p3 are parameters.

Below, we explain the derivation of these three measures that together constitute the Benefit of a merge. For the following discussion, let n1 and n2 be the two sets of compatible operator nodes, let e:= n1 ∪ n2 denote the resulting merge node, and let n stand for the number of bits of the input arguments.

Measure 1: Merge Costs

The Merge-Cost expresses the gain of reducing the necessary hardware units needed to implement the flow graph. The merge costs for e, denoted by Merge-Costs(e), are calculated as described next. If the two operator nodes n1 and n2 have *identical* functionalities then the operation node resulting from merging them has no additional costs. There is a gain however since one of the two units is no longer needed, thus, w.l.o.g., Merge-Costs(e) = - **op-cost**(n1) $\times$ n. If the two operator nodes n1 and n2 are *mergable* but not necessarily identical, the cost of executing a merge on these two nodes is taken to be

Merge-Costs(e) := + **op-cost**(e)$\times$ n - **op-cost**($n1$) $\times n$ - **op-cost**($n2$) $\times n$

If the resulting multi-functional node has more than one functionality, then a decoder is needed to select among them. The cost for this decoder is calculated by increasing Merge-Costs(e) by |**functionality**(e)|.

Measure 2: Connection or Mux Costs

The second measure, Mux-Cost, takes the connection costs into consideration that result when new choose value nodes have to be introduced. The degree of similarity in input patterns of these two operator nodes determines whether choose value nodes are needed for the two inputs of the new node e as well as the size of these choose value nodes. See Figures 4 for an example of how two choose value nodes have been added. If there are l distinct input values with $l > 1$ for the left input of e then the addition of a choose value node would adjust the costs by

Mux-Costs(e) := $l \times (1 + n)$.

Mux-Costs for the addition of a choose value node for the right input of e are adjusted in a similar manner.

For each pair of outgoing data flow output edges from n1 and n2, that are directly connected to a choose value node, this choose value node can be reduced in size. If there are q different merges of choose values vi and vj into a combined choose value

condition vi $\wedge$ vj (as shown in the example in the Figures 4 then the estimated cost is adjusted by

Mux-Costs(e) := Mux-Costs(e) - $q \times n$.

Measure 3: Ancestor Mergability

The third measure, called Ancestor-Mergability, evaluates the potential of direct ancestors of the merged nodes for being merged. For each pair of operator nodes that are directly connected by data flow output edges as inputs to n1 and n2, respectively, and that are compatible and thus could be merged in the future, increment Ancestor-Mergability(e). This cost evaluation accounts for the fact that a merge directly atop the current merge is likely to reduce the connection costs by making choose value nodes redundant.

5.3 Functionality Recognizer

Next, we describe shortly the functionality recognizer portion of CSA that addresses the functionality mismatch problem created due to the use of VHDL for synthesis. Recall, that the VHDL description is expressed by VHDL primitives, such as, the +, - and $\wedge$ functions and, there is not always a one-to-one mapping from these primitives to the functions supported by the components of the chosen library. For instance, the expression "A+B+1" would result in two operator nodes in the flow graph, but can be directly implemented by an ALU or ADDER/SUBTRACTOR component. Thus, the expression tree for A+B+1 can be optimized and replaced by the ternary operation node + (A,B,1).

The current implementation of CSA recognizes patterns consisting of two operators. This could easily be extended to general pattern matching if needed. This functionality recognition and possibly functionality merging is orthogonal to the operator merging discussed in Section 5.1 as is done with nodes that are not mutually exclusive. We incorporate this into the CSA algorithm by simultaneously considering both types of merges. This is important since it is not always clear a priori whether a functionality

merge will lead to a better design. When calculating the cost of a compatibility edge, we pattern match the subgraph structure of which the two nodes are roots with the function patterns supported by the library. If a match is found, the costs of replacing these patterns by one function node and then merging it with the second node are calculated similarly as described in Section 5.2. If this cost is better than the cost of merging the simple operation node, then the edge is marked according. However, the functionality merge is not immediately carried out. Only when a marked compatibility edge is picked as the next merge edge by the CSA algorithm, then the functionality reduction is executed before the operator merging takes place.

If this pattern matching option is not used, then the design will still synthesize. The algorithm will however not make use of all functionalities provided by the library and thus may not result in an optimal design.

5.4 The CSA Algorithm

We can now formulate our problem in a graph theoretic notation. The problem is to find a "clique cover" of the compatibility graph CG of minimal cost, where the term clique cover means a collection of sets of operator nodes that contains *all* nodes of the graph CG exactly once and each set corresponds to a clique [11]. Theoretically, an optimal solution to this problem can be found by the following steps: (1) find all cliques of CG, (2) create all possible clique coverings out of these sets, (3) compute the cost of each such covering, and (4) select the best one. Unfortunately, this exhaustive algorithm is exponential in time, and thus not practical. Therefore, we approach this problem by applying heuristics in the form of cost estimates described in Section 5.2 to incrementally build a clique cover. CSA iteratively chooses the pair of operator nodes to be merged next based on a heuristic that globally evaluates the "contribution" of this merge to the overall solution. The complete specification of the CSA algorithm is given in Figure 16.

Input: a data flow graph consisting of uni-functional operator nodes.
Output: a data flow graph with (uni- and) multi-functional operator nodes.
Algorithm:

1. Disambiguate default values of conditions as described in Section 4.1.
2. Traverse the data flow graph to assign labels to each operator node (Section 4.1).
3. Transform the input data flow graph into a compatibility graph by inserting compatibility edges between any pair of compatible operator nodes, i.e., nodes that are mergable by the given unit table (Definition 3) and mutually exclusive by the labeling evaluation scheme (Proposition 1).
4. Calculate the benefit of each compatibility edge by the cost evaluation scheme presented in Section 5.2.
5. Choose the compatibility edge e with the largest benefit as evaluated by the heuristic described in Definition 5.
6. If no compatibility edge has been selected in step 5 then STOP.
7. Readjust status of compatibility edges to compensate for the choice of e(n1,n2):
   (a) Delete edges if they have no more potential of being made permanent in the future due to the compatibility property (Section 4.3).
   (b) For all operator nodes, n, that have compatibility edges with n1 and n2, merge the edges e(n1,n) and e(n2,n) into one edge e(n1+n2,n) (Section 4.3).
8. Perform data flow graph transformations to compensate for the choice of e:
   (a) Merge the two operator nodes n1 and n2 that are connected by e into one multi-functional operator node (name it n1).
   (b) Update the output data flow edges of the new node n1 to connect to the original outputs of n1 and n2. If n1 and n2 were directly connected to the same choose value node then reduce the size of this choose value node by combining these two inputs into one (Section 5.1).
   (c) Update the input data flow edges to the new node n1 to connect to the original inputs of n1 and n2. If the left (right) inputs of n1 and n2 were not identical then insert a new choose value node or extend the existing choose value node for the respective input port (Section 5.1).
   (d) Delete the old operator node n2.
9. Delete the edge e. Readjust benefits of compatibility edges. Only edges adjacent to the the newly created node n1 are affected by this (Section 4.3).
10. Goto 5.

Figure 16: The CSA Algorithm

A first inspection of the algorithm specified in Figure 16 makes it apparent that the execution time of the algorithm has been reduced from exponential (for the maximal clique partitioning) to a polynomial time. We can show the following:

Theorem 1 *The worst case run time of the CSA algorithm is $O(n^3)$ where n is the number of operator and choose value nodes in the input graph.*

Below, we sketch the argument for this run time. CSA consists of two phases, the preprocessing phase (steps 1 to 4) which is executed once, and the execution phase (steps 5 to 10). The later corresponds to a loop.

Let us first analyze the preprocessing phase. The default processing can be done in time O(n) since it has to be done at most once for each choose value node. The labeling algorithm, step 2, can be done in time O(n). The generation of compatibility edges, however, is of complexity $O(n^2)$. This is so since in the worst case, namely, when all operator nodes are compatible with all others, up to n^2 such compatibility edges may be produced. Consequently, the performance of evaluating the cost function for all edges is $O(n^2)$ as well, assuming that the number of function patterns is constant.

Next, we examine the execution phase of the algorithm. There are up to n^2 compatibility edges, thus one may assume that in the worst case n^2 iterations of selecting another edge could occur. This is not the case. In fact, the number of iterations is limited to n. The reason for this is the following. During each iteration, two operation nodes are merged into one node. Since there are only n nodes, no merging can take place after n rounds and the algorithm would stop (with step 6).

For the steps within the loop (steps 5 to 10) we observe the following:

Step 5: if we assume that the edges are kept in an unordered list, then the whole list has to be traversed to find the edge with the maximal benefit. Thus, the execution time is $O(n^2)$.

Step 6: O(1).

Step 7: There are at most n compatibility edges associated with each node. You may compare each edge of node n2 with all edges of node1, and therefore, altogether the time is $O(n^2)$.

Step 8: The transformations of step 8 are local within the flow graph, and thus, require an execution time of at most O(n).

Step 9: There are at most n edges associated with each node, and these are the only ones that are involved in cost reevaluations. The insertion back into the list is O(1) since the list is not kept in order. Thus, the time for step 9 is O(n).

Step 10: This is O(1).

Putting it together. As shown above, the steps of the loop body have a time complexity of $O(n^2)$ and the loop is carried out at most n times. Thus, the execution phase of CSA is bounded by $O(n^3)$. Since the preprocessing phase complexity is $O(n^2)$, the overall worst case time bound of CSA is $O(n^3)$.

Let us note here that the analysis assumes a rather primitive implementation. We believe that the use of more sophisticated data structures, such as a heap, instead of an unordered list for the compatibility edges, would reduce the execution time considerably.

6 Experiments

The following section discusses some of the experiments we have performed to validate the CSA algorithm. CSA has been implemented in the C programming language and is currently running on a SUN3 workstation under the UNIX operating system. It consists of approximately 9000 lines of C code not including the VHDL input compiler.

We have assumed two libraries for the test cases: first, the Generic Component library [4] and second, the components provided by Texas Instruments taken from the TTL Data Book [13].

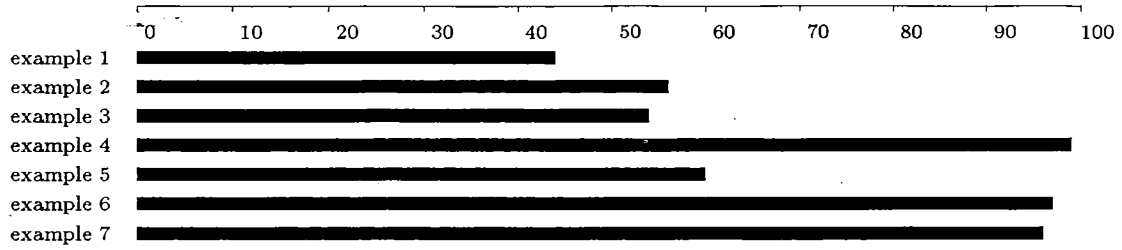
We have chosen seven typical VHDL descriptions of hardware components as input to the algorithm. One of these examples, depicted in Figure 17 describes an ALU component proposed by Mano in [6] (page 371). Two others are variations of this ALU component; the first is a alternative description again as proposed by Mano in [6] (page 381) and the other is an incomplete specification of the former. Another example description is a functional reduced version of the TI 74181 ALU taken from the TTL Databook [13]. We use the VHDL data flow style to model these components [4].

The graph in Figure 18 shows our findings for all example descriptions. There are several observations we want to make. First, the CSA algorithm produces better designs when using the TTL library than the Generic Component Library. There are only two cases (examples 1 and 3) when the use of these two libraries results in identical design. These two descriptions don't contain any expressions that correspond to functionalities supported by TTL and not by the Generic Component Library, and therefore, the choice of the underlying library did not matter.

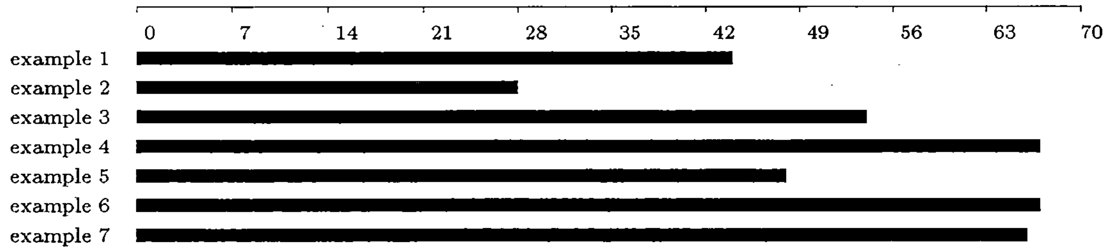
In Figure 18, we also show the results of human designers using the two libraries. In the case of the TI components, the human designer produced the same result as CSA for five out of seven examples. For the last two examples, examples 6 and 7, the designer improved the design for the following reason. The designer recognized the fact that descriptions 6 and 7 are equivalent to description 5. In other words, the designer replaced the functions "A - INV(B) - 1" by "A - B" and the function "A - INV(B)" by "A - B - 1", both of which are supported directly by the TI 74181. Therefore, the designer was able to reduce the description to one component, the TI

```
entity alu is
 port (
     F: in BIT-VECTOR(3 downto 0);
     DATA-A, DATA-B: in BIT-VECTOR(3 downto 0);
     ALU-OUT: out BIT-VECTOR(3 downto 0)
     );
end alu;
architecture dataflow of alu is
begin
 with F select
 ALU-OUT <=
         DATA-A                      when "0000", – Transfer A
         DATA-A + "0001"             when "0001", – Increment A
         DATA-A + DATA-B             when "0010", – Addition
         DATA-A + DATA-B + "0001"    when "0011", – Addition - carry
         DATA-A - DATA-B             when "0101", – Subtraction
         DATA-A - DATA-B - "0001"    when "0111", – Subtraction
         DATA-A - "0001"             when "0111", – Decrement A
         DATA-A                      when "1000", – Transfer A
         DATA-A and DATA-B           when "1001",
         DATA-A or DATA-B            when "1010",
         DATA-A xor DATA-B           when "1011",
         inv(DATA-A)                 when "1100"; – Complement A
end dataflow;
```

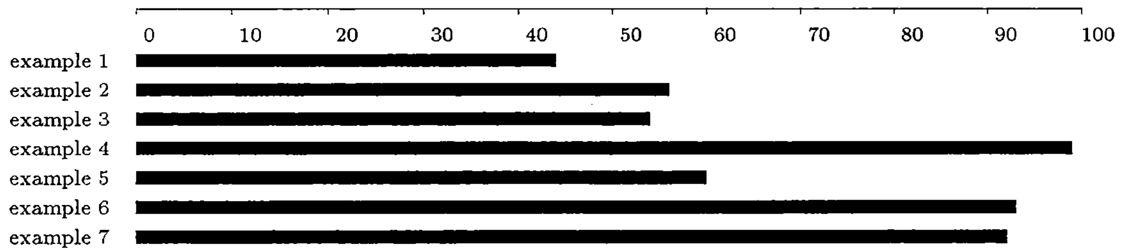
Figure 17: ALU Description Taken from Mano (page 371)



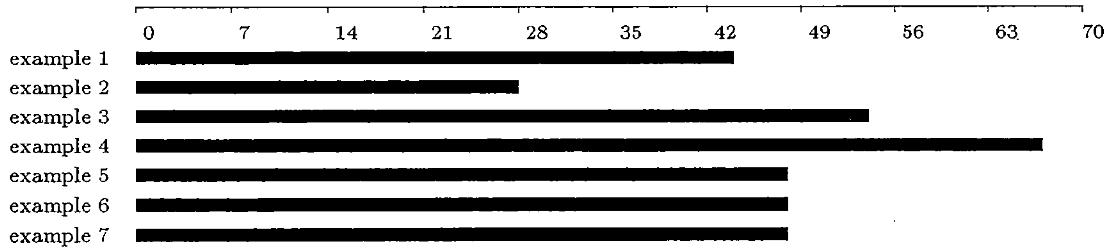
a) CSA using the Generic Component Library



b) CSA using TI Components



c) Human Designer using the Generic Component Library



d) Human Designer using TI Components

Figure 18: Cost Functions for the Seven Design Examples

74181 ALU. This shows that the design modeling has a direct influence on the quality of the design. This result suggests that further research should address the problem of either recognizing semantically equivalent functionalities or restricting the design modeling languages to avoid such distinct descriptions of one function. Note also that the results for example 6 and example 7 are the same, even though the former is more complex than the latter. The reason for this is that CSA cleverly uses 74181's ADD-and-INCREMENT function to implement parts of the "A + NEGATION(B) + 1" function.

The results of the human designer using the Generic Component library versus CSA using the Generic Component library are also compared in Figure 18. Again, the human designer produced the same result as CSA for five out of seven examples. For the last two examples, the designer was able to do marginally better. In this case, however, the recognition of the fact that "A - INV(B)" is semantically equivalent to "A - B - 1" did not help the designer, since this function is not supported by the ALU provided by the Generic Component library. The savings arise from the fact that the human designer implements the expression "A + 1" by the function "inc(A)". This allows him to reduce the size of one of the choose value nodes by 1, since the constant 1 is no longer needed as input as it is implicit in the function description "inc".

Example	CSA using TTL	Designer using TTL	CSA using GENUS	Designer using GENUS
example 1	1	1	1	1
example 2	1	1	2	2
example 3	1	1	1	1
example 4	1	1	3	3
example 5	1	1	3	3
example 6	3	1	5	4
example 7	3	1	5	4

Figure 19: Number of Components Produced by CSA and Human Designer

In Figure 19, we also show the results of using both libraries; this time however we list the number of units that are being generated for the seven behavioral descriptions. The table shows that CSA using TTL components is not only superior in terms of the cost of the resulting design, but also in terms of the number of required components. With the exception of examples 6 and 7, the algorithm generates the minimal number of components and is thus as good as a human designer. For these five cases it reduces the design to one component, which is very desirable since it means that the behavioral description has been recognized. In example 7, this is not feasible since the underlying TI components do not directly support the "A - INV(B) - 1" function.

Figure 19 also shows that the human designer using the Generic Component library reduces the number of components for descriptions 6 and 7 from four to three. This is however done at the cost of increasing the overall cost of the design, since in the Generic Component library it is cheaper to use an ADDER and an INVERTER than to use one unit that executes both functions. The latter, a complete arithmetic/logic unit, implements several other functions and is thus more expensive.

We ran a second set of tests using design specifications that describe a combination of components instead of one component only. In this scenario we assumed that the components execute concurrently. Furthermore, they receive input data from and write output data to a set of register files. One of the input descriptions (referred to as example 8 in Figure 20) corresponds to the design description shown in Figure 8. The results of this experiment are shown in Figure 20. We find that CSA is indeed able to reduce the design descriptions to the initial number of components. In example 10, CSA reorganizes functionalities among the different components and thus the optimized design produced by CSA and by the human designer are distinct. However, the designs are equal in quality; i.e., they consist of the same number of components and the costs are identical.

Example	CSA using TTL	Designer using TTL
example 8	2	2
example 9	4	4
example 10	3	3

Figure 20: Number of Components Produced by CSA and Human Designer

7 Conclusion

In this paper, we give a solution to the problems created when using VHDL as a description language for behavioral modeling of components. In particular, we present an algorithm that recognizes a possibly incomplete behavioral description and generates a minimal set of components from a given library.

Our experiments show that in most cases the CSA algorithm produces a design consisting of the minimal number of components. Thus, the performance of CSA is comparable to that of a human designer. Our experiments have also shown the importance of a functionality recognizer. An important question of high-level synthesis is to deal with this *functionality mismatch* between operations of the description language and operations supported by the library. We have suggested the development of a pattern matching scheme as a general solution to this problem. We believe that this constitutes a valuable contribution to the automated synthesis field, however, several possible improvements remain that we plan to pursue. In particular, we intend to develop data structures that directly support and thus possibly lead to a speed-up of the CSA algorithm.

Acknowledgements We thank Joe Lis for the use of the VHDL compiler, that he has developed as part of the VSS high-level synthesis system [5], as well as for his work

on the design representation.

References

[1] Armstrong, J., *Chip Level Modeling with VHDL*, Prentice-Hall, 1989.

[2] Camposano, R. and R. M. Tabet, Design Representation for the Synthesis of Behavioral VHDL Models, Research Report, IBM General Technology Division, Burlington, VT, RC 14282, Dec. 1988.

[3] Devadas, S. and Newton, A. R., Algorithms for Hardware Allocation in Data Path Synthesis, *IEEE Transactions on Computer Aided Design*, Vol. 8, NO. 2., pp. 768 – 781, July 1989.

[4] Dutt, N., GENUS: A Generic Component Library for High Level Synthesis, Technical Report 89-09, University of California, Irvine.

[5] Lis, J. S., and D. D. Gajski, Structured Modeling for VHDL Synthesis, DAC'89.

[6] Mano, M. M., Computer Engineering Hardware Design, Prentice Hall, Englewood Cliffs, N.J., 1988.

[7] McFarland, M. C., Parker, A. C., and Camposano, R., Tutorial on High Level Synthesis, *25th Design Automation Conference*, July 1988.

[8] Orailoglu, A. and D. D. Gajski, Flow graph representation, *Proc. of 23rd Design Automation Conf.*, Las Vegas, Jun. 1986, 503 – 509.

[9] Park, N. and A. C. Parker, Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications, *IEEE Trans. on Computer-Aided Design,* Vol. 7, No. 3, March 89, 356–370.

[10] Saunders, L., The IBM VHDL Design System, *24th DAC*, pp. 484 - 490.

[11] Tseng, C. and Siewiorek, D. P., Automated Synthesis of Data Paths in Digital Systems, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, CAD-5, 3 (July, 1986), 379 - 395.

[12] Thomas, D. E., Automatic Data Path Synthesis, *Design Methodologies*, (S. Goto, editor), Chapter 13, Elsevier Publishers, 1986.

[13] The TTL Data Book for Design Engineers, Texas Instruments Incorporated, Second Edition, 1967.

[14] VHDL Language Reference Manual, Addison Wesley, 1988.

[15] Walker, R. A., and Thomas, D. E., A Model of Design Representation and Synthesis, *DAC'85*, 1985, 453–459.

A Appendix A

Users Manual of the

Component Synthesis Algorithm

(This appendix was written by Elke A. Rundensteiner.)

A.1 Introduction

This users manual describes the software of a prototype version of the Component Synthesis Algorithm (CSA) developed at the University of California, Irvine. The Component Synthesis Algorithm (CSA) recognizes a possibly incomplete behavioral description of a design and generates a minimal set of components from a given library. CSA has been designed to address problems that have resulted from the use of VHDL [4] as hardware description language for automated synthesis. For instance, a behavioral description of components is given by using standard operators in the language, and therefore, a mismatch between the operators of VHDL and the functionalities provided by the library components arises. Furthermore, VHDL does not guarantee uniqueness of descriptions. CSA solves these problems. In particular, it maps a complex behavioral description of a unit to one hardware component whenever possible. CSA is implemented in the C programming language and is currently running on SUN 3 workstations under the UNIX operation system.

A.2 Running CSA

A.2.1 The Input Description

The input file consists of a textual VHDL behavioral description in the dataflow style. The format of the VHDL language subset accepted by CSA corresponds to the subset accepted by the VHDL compiler developed for the VSS system [1], since the VHDL compiler serves as frontend to CSA. Example input files can be found in Figures I and II and online in the < CSA > input_data subdirectory (currently, rundenst/res/vlsi/merge/CSA). The input file has the following naming convention:

< design-name >.vhdl

In addition to the VHDL description, the example input description in Figure I contains the line "–VSS: design_style FUNCTIONAL" to indicate the desired design style to the VHDL compiler.

CSA can also be embedded into another synthesis system as flow graph optimization procedure. In this case, the input to CSA is a data flow graph in the format specified in the UCI CADLAB Flow Graph Data Structures document [2]. This initial data flow graph consists of single-functional operator nodes and choose value nodes.

A.2.2 The CSA Execution

To execute CSA as a stand-alone system, enter the following command:

% csa < design-name >

You can also include CSA in form of a function call into your favorite synthesis system, as long as the internal data flow graph representation is in the flow graph format described in [2]. In this case, the executable file "csa.o" has to be loaded with your C code. The file "csa_main.h" contains the function definition "csa_algo()" and

therefore must be included into the file in which you want to call the CSA function. Execute CSA with the following function call:

< ptr_to_node_net_list > = csa_algo (< ptr_to_node_net_list >);

The input argument to the csa_algo function call is a pointer of the data type "struct df_node_net_list" (see [2] for the definition of this data type). This argument points to the beginning of the list of all data flow nodes and nets that comprise the input data flow graph. Output is again a pointer to a "df_node_net_list" type data structure. This pointer points to the modified list of data flow nodes, the optimized design. This return argument is needed in the case that the first element of the input list has been deleted.

The output of CSA is a data flow graph expressed by UCI CADLAB Data Flow Graph Data Structures [2]. It now consists of multi-functional operator nodes instead of only single-functional operator nodes. It also contains newly created choose value nodes, which have been inserted to distinguish between the inputs of a multi-functional operator node. In addition, CSA introduces DECODER nodes to control the function select of multi-functional operator nodes. These DECODER nodes contain a truth table that describes under which conditions a function is to be selected for execution. Also, CONCATENATION nodes are added to express the combination of conditions that together guard choose value nodes (if the later are based on more than one condition). Besides these additions to the flow graph, there are two types of reductions: first, uni-functional operator nodes are deleted as their functionalities are merged into multi-functional operator nodes, and secondly, original choose value nodes are removed by CSA whenever they become redundant. For a choose value not to be redundant means that it has only one data input.

A.2.3 Output Files

Upon successful completion, CSA will produce a data file with trace information with the following naming conventions:

< design-name >.csa

This file < design-name >.csa documents the different execution cycles of CSA by listing for each cycle: (1) the remaining compatibility edges (potential operator merges), (2) the compatibility edge selected to be merged next (the next operator merge), and (3) the modified data flow graph.

Furthermore to observe the changes made to the data flow graph by CSA the following two flow graph diagram outputs files are generated:

< design-name >_before_csa.dgm

< design-name >_after_csa.dgm

Both can be used as input files to the Flow Graph Graphical Display Utility (**dp**) described in [1], which displays the flow graph in a graphical format.

The following is an example output of what will be printed to the screen when CSA is run successfully.

```
Print Graph Compiler debug information (y/n)?: n
design name       : example
VHDL input file   : example.vhdl
symbol table file : example.st
node table file   : example.nodes
Initialization complete...

====================================

Beginning Graph Compilation phase...

====================================

completed parsing entity_header
```

```
completed parsing entity_declarative_part
Using FUNCTIONAL design model for current block/process
completed parsing architecture header
completed parsing architecture_declarative_part
completed parsing architecture_statement_part
Input Compiler: data structure created successfully
creating final diagram file: example_final
-- generate_cf_diagram_data: creating file example_final
Printing flow graph node and net tables....
Compilation of design example completed...

====================================

Beginning CSA ...

====================================

-- generate_diagram_file_from_df_list: creating file example_before_csa
**** Initial total cost = 138

 DFG Merging ...

 No more gain.
 Do you want to continue merging?
               0 == no      ==>
               1 == yes     ==> 1

 No more gain.
 Do you want to continue merging?
               0 == no     ==> 0
               1 == yes    ==>

**** CSA: condition node optimization completed.
**** Final total cost = 88
**** CSA output stored in example.csa
-- generate_diagram_file_from_df_list: creating file example_after_csa
```

As seen in the above run, the user may be prompted to determine whether merging should be further pursued. This will happen if there are merging steps left that are correct but will no longer bring any gain.

A.3 The CSA Data Structures

This section describes the data structures needed to support CSA in addition to the UCI CADLAB data flow data structures [2]. A listing of the data structure definitions is included at the end of this section.

A.3.1 Compatibility Edge List

CSA stores all pairs of potentially mergable operator nodes into a table, called compatibility-edge-list. Each entry in this table, called compatibility edge, consists of the following information:

- a pointer into the data flow graph to the first operator node n1
- a pointer into the data flow graph to the second operator node n2
- different entries on the quality of this operator merge to be used by the heuristic;
- pointers to the previous and next entry in the table that holds another compatibility edge (to build a linked list of these edges)

Initially, this table will consist of all possible pairs of mergable nodes. During each iteration of CSA, the size of this list decreases since pairs of operator nodes are taken off the list either (1) since they have been merged or (2) since they are no longer candidates for future merges. When the compatibility list is empty then the algorithm stops.

A.3.2 Compatibility Edges

The data structure for the operator nodes is expressed by UCI CADLAB data flow data structures [2], however, it is also extended by tool-specific information, in this case, CSA-specific. CSA associates with each operator node a list of df_compatibility edges that establish the link from the data flow graph to the compatibility edge list.

A.3.3 Others Table

There is no need for a new data structure to implement the storage of all **others** values, instead we can use the "struct cond_val_info" data structure which is part of the UCI CADLAB data flow data structures [2].

A.3.4 Unit table

CSA needs a table that indicates the collections of operations that can be executed by each hardware unit as well as their cost, called the unit table. It is an array of structures where each structure contains the following entries:

- unique name for this type of unit
- list of one or more functionalities (Op-Type)
- cost of unit per bit (Op-Cost)

A.3.5 Listing of the CSA Data Structures

```
/*************************************************************************/
/*          Data Structures for compatibility list                       */
/* This represents a doubly-linked list of compatibility edges           */
/*************************************************************************/

typedef struct compatibility_edge_list
{
  struct df_node_net              *op_node1,  /* left node  */
                                  *op_node2;  /* right node */
  int                             cost1,
                                  cost2,
                                  potential1,
                                  potential2,
                                  special;
  float                           total;
  struct compatibility_edge_list  *prv,
                                  *nxt;

} COMPATIBILITY_EDGE_LIST, *COMPATIBILITY_EDGE_LIST_PTR;
```

```
/****************************************************************************/
/*                    df_compatibility edges                                */
/*  These data structures represent CSA tool-specific  additions to the     */
/*  operation node df_node_info of the UCI CADLAB data flow data structures */
/****************************************************************************/

enum Position_Type { left, right };

typedef struct fta_tool_info
{
   struct df_node_net    *node;              /* point back to node */
   int                num_op_info_nodes;  /* # of df_op_info nodes */
   int                num_df_comp_edges;  /* # of df_compatibility edges */
   struct df_compatibility_edge *edge_head;   /* pointers to comp. list */
   int            trav_flag;

}  FTA_TOOL_INFO;

/* position in list of compatibility edges of the compatibility table. */

typedef struct df_compatibility_edge
{
 struct df_compatibility_edge        *nxt,   /* linked list of references */
                                     *prv;
 struct compatibility_edge_list      *edge;  /* point to actual table
                                                of these edges */
 struct fta_tool_info                *back;  /* point back to operation node */
 enum Position_Type                  position;

} DF_COMPATIBILITY_EDGE;

/***************************************************************************/
/*   subset of the  UC Irvine CADLAB Data Flow Graph Data Structures      */
/***************************************************************************/
/* this sub structure of DF_OP_INFO maintains a linked list of lists of   */
/* condition-value pairs  for conditional execution of operators.         */
/***************************************************************************/

typedef struct cond_val_info
{
   struct cond_val_pair *cond_pair1;
   struct df_op_info *op_info;
   struct cond_val_info *prv,*nxt;

} COND_VAL_INFO, *COND_VAL_INFO_PTR;

typedef struct cond_val_pair
{
   char *condition;
   char *value;
   struct cond_val_pair *prv,*nxt;
```

```
} COND_VAL_PAIR, *COND_VAL_PAIR_PTR;

/****************************************************************************/
/*        unit table data structures                                      */
/*  This data structure describes a (read-only) unit table that           */
/*  contains the units, their functionalities and their cost.             */
/****************************************************************************/

#define unit_table_size 30

typedef struct function_list
{
  struct function_list       *nxt;     /* linked list of functions */
  Op_Type                    op_type;
  char                       *op_name;

} FUNCTION_LIST, *FUNCTION_LIST_PTR;

typedef struct unit_entry
{
  char                    *unit_name;
  struct function_list    *nxt;
  int                     op_cost;

} UNIT_ENTRY, *UNIT_ENTRY_PTR;

struct unit_entry  unit_table[unit_table_size];
```

A.4 The CSA Algorithm

Below, we describe the main features of CSA. For a more detailed description of the algorithm see [3]. CSA operates in three phases, the pre-processing phase, the merging phase, and the post-processing phase. The interrelationships of the different modules are shown in Section E.

A.4.1 The Pre-processing Phase

First, equivalent data flow graphs that serve as condition inputs to different choose value nodes are merged. The reason for why this is done is best explained with an example. See for instance the example design description in Figure I, where the condition "(S = B"0")" appears in both conditional signal assignment statements.

```
library jlis;
use jlis.defs.all;
entity alu is
  port (
        S: in BIT;
        S2: in BIT;
        DATA-A, DATA-B: in BIT-VECTOR(3 downto 0);
        DATA-OUT1: out BIT-VECTOR(3 downto 0);
        DATA-OUT2: out BIT-VECTOR(3 downto 0)
        );
end alu;

-VSS: design_style FUNCTIONAL

architecture dataflow of alu is
begin
  DATA-OUT1 <=
     DATA-A + DATA-B      when (S = B"0") else
     DATA-A or DATA-B     when (S = S2) else
     DATA-A and DATA-B;

  DATA-OUT2 <=
     DATA-A or DATA-B     when (S = B"0") else
     DATA-A - DATA-B;
end dataflow;
```

Figure I: A VHDL Description Using Conditional Signal Assignment Statements

The VHDL compiler will generate two different data flow expression trees for these two occurrences of the same condition. If this flow graph representation were to be used directly, then the choose value nodes that represent these two conditions would be guarded by different operator nodes. Thus they would be considered to be distinct conditions. This is so because a condition is identified by the unique node number of the operator node that guards the choose value node, also called a condition node. To solve this problem, CSA replaces these two equivalent expression trees by *one* expression tree that feeds both choose value nodes. This allows operator merging to proceed across different signal assignment statements. The recognition of equivalent conditions done by CSA, however, is minimal as it serves demonstration purposes only. The development of a general-purpose pattern recognition tool would be useful extension of this work.

After this, CSA finds the number and type of distinct conditions of the data flow graph and builds one concatenation node, CONDITIONS. This node holds a unique condition label for each condition and it also establishes an order among these conditions.

Next, the default value processing takes place. As seen in Figure II, in VHDL the value **others** stands for the set of all values that have not been enumerated explicitly in a selected signal assignment statement. Therefore, each occurrence of the **others** value represents a distinct meaning that can only be gotten from the context of the statement in which it appears. In the example in Figure II, the first **others** value corresponds to the 14 values in the range from "0001" to "1111". This can be represented by associating all 14 values with this **others** value, or, vice versa, by associating with it all values that it does not represent. For instance, **others** $= \neg$ "0000" $\wedge \neg$ "0001". The later approach is preferable to the former one, since the number of possible values may not be finite. On the other hand, the number of values an **others** value does not

represent is always finite as it corresponds to the values that are explicitly expressed in the selected signal assignment. The meaning of each **others** value may only be gotten from the context in which it occurs. As this context (a choose value node) may be modified or potentially removed during the CSA execution, CSA replace each occurrence of an **others** value by a unique value of the form **others<id>** where id is a unique integer number. CSA collects these **others<id>** values and their corresponding meaning in a separate table.

```
architecture dataflow of design-desc is
begin
 with cond  select
 DATA-OUT1 <=
   DATA-A +  DATA-B      when  "0000",
   DATA-A or DATA-B      when  "0001",
   DATA-A and DATA-B     when  others;

 with cond  select
 DATA-OUT2 <=
   DATA-A -  DATA-B      when  "0000",
   DATA-A +  DATA-B      when  "1111",
   DATA-A +  DATA-B      when  others;
end dataflow;
```

Figure II: A VHDL Description Using Selected Signal Assignment Statements

Next, the operator nodes are labeled by assigning them a label that describes the condition under which they are to be executed. This condition label consists of a list of condition identification and associated value pairs. The condition identifications are taken from the CONDITIONS node, while the values correspond to port choose values of choose value nodes or to the special value "don't care".

Then, compatibility edges are created between each pair of operator nodes that is compatible; i.e., they are mergable by the given unit table and they are mutually exclusive by the labeling evaluation scheme.

At last, the cost of each compatibility edge is calculated by the cost evaluation scheme presented in [3].

A.4.2 The Merging Phase

This section gives a short overview of the main algorithm, for a more detailed description see [3].

1. If there are no compatibility edges left then START post-processing.
2. Choose the compatibility edge e with the largest benefit (smallest cost). The edge e represents a merge of the operator nodes n1 and n2.
3. Readjust status of compatibility edges to compensate for the choice of e(n1,n2):
   (a) Delete edges if they have no more potential to be made permanent in the future due to the compatibility property. These edges will be adjacent to the operator nodes n1 and n2.
   (b) For all operator nodes, n, that have compatibility edges with both n1 and n2, merge the edges e(n1,n) and e(n2,n) into one edge e(n1+n2,n).
4. Perform flow graph transformations on the data flow graph to compensate for the choice of e (for details see [3]):
   (a) Merge the two operator nodes n1 and n2 that are connected by e into one multi-functional operator node (name it n1).

(b) Update the output data flow edges of the new node n1 since it now has to connect to the original outputs of n1 as well as n2. If n1 and n2 were directly connected to the same choose value node then reduce the size of this choose value node by combining these two inputs into one.

(c) Update the input data flow edges to the new node n1 since it now has to connect to the original inputs of n1 as well as n2. If the left (right) inputs of n1 and n2 were identical (data access nodes) then nothing has to happen here. Otherwise, a new choose value node has to be inserted for the respective input port.

(d) Finally delete old node n2.

5. Delete the edge e. Readjust benefits of compatibility edges. Note that only compatibility edges adjacent to the the newly created operator node (n1+n2) are affected by this.

6. goto 1.

A.4.3 The Post-processing Phase

Note that in order to simplify the execution phase of the algorithm, a condition label used by CSA always consists of a complete listing of all conditions found in the data flow graph, in particular, in the sequence as listed by the CONDITIONS node. Since simplifies the creation of choose value nodes and the merging of operator nodes into multi-functional operator nodes. Therefore, a post-processing phase is needed takes care of condition optimization and other such clean-up actions.

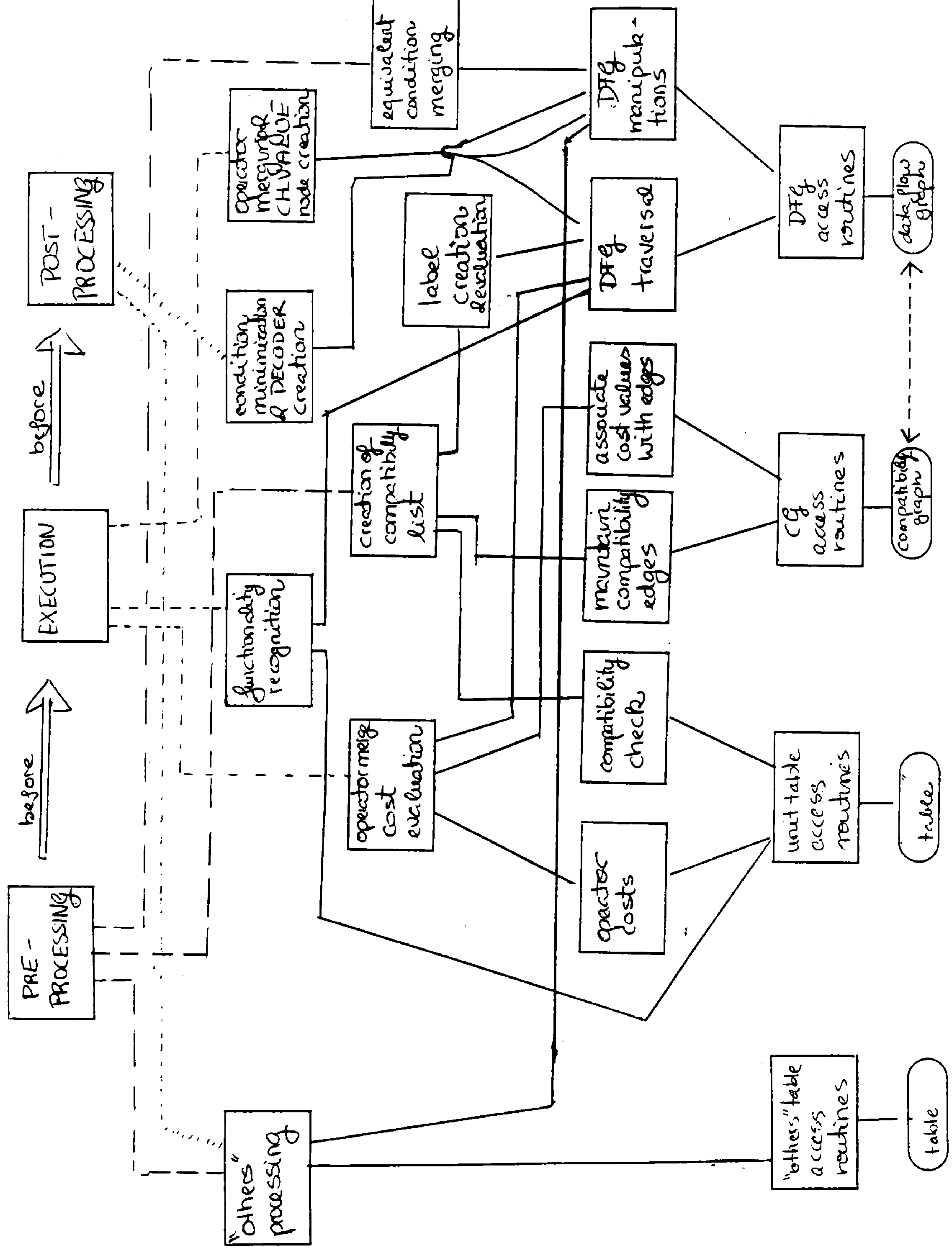
In particular, CSA optimizes the number of conditions for each choose value node in the following way: If the node is an original choose value node that existed in the input data flow graph do nothing. If it is a choose value node that has been

generated by CSA then the number of conditions needed to distinguish between its input ports is optimized. To simplify the execution phase of the algorithm, such a choose value node is guarded by all conditions found in the data flow graph, namely, by the CONDITIONS node. If for one of these conditions the values on all ports are "don't care" and/or *one* other value then that condition is redundant. In this case, CSA generates a data flow node of type concatenation that collects all conditions that are not redundant and it uses this node as guard input to the choose value node. CSA also reduces the values of the input ports correspondingly.

Next, CSA dos the following for each OPERATION node. If the **functionality** of the node is one (even if it has been assigned several times the same function) then the associated condition label is deleted. If the **functionality** of the node is greater than one then a DECODER node is generated. This DECODER node serves as control input to the operation node, as it expresses the function select of the synthesized component in form of a truth table. The DECODER node is connected by a control input port to the corresponding operation node. After this, the number of conditions needed to distinguish between its input ports is optimized as described in the previous paragraph. The operation nodes that serve as conditions in this optimized function select (the columns of the truth table) are connected by separate data input ports to the DECODER node.

The last step of CSA is to replace the occurrence of each value of the form **others<id>** by the string **others**. These values **others<id>** appear in the truth table description of a DECODER node and it could correspond to condition values associated with input ports of choose value nodes.

A.5 Interrelationships of CSA Modules



References

[1] Lis, J. S., The VSS Users Manual, Information and Computer Science Department, University of California at Irvine, July 1989.

[2] Lis, J. S., and Dutt, N. D., The Flow Graph Data Structure Specification, CAD-LAB Internal Document # 4, Information and Computer Science Department, University of California at Irvine, September 1989.

[3] Rundensteiner, E. A., Gajski, D., and Bic, L., Technology Mapping For Register Transfer Descriptions, Technical Report 89-42, Information and Computer Science Department, University of California at Irvine, December 1989.

[4] VHDL Language Reference Manual, Addison Wesley, 1988.