UC Irvine
ICS Technical Reports

Title
Component synthesis from functional descriptions

Permalink
https://escholarship.org/uc/item/7sr8h86h

Authors
Rundensteiner, Elke A.
Gajski, Daniel D.
Bic, Lubomir

Publication Date
1991

Peer reviewed

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

Z
699
C3
no. 90-24
Rev.

Component Synthesis From Functional Descriptions

Elke A. Rundensteiner, Daniel D. Gajski and Lubomir Bic

Department of Information and Computer Science
University of California, Irvine
August, 1990
(Revised version: August, 1991)

Technical Report 90-24

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

COMPONENT SYNTHESIS FROM FUNCTIONAL DESCRIPTIONS

Elke A. Rundensteiner, Daniel D. Gajski and Lubomir Bic
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

August, 1991

Abstract

In the literature, it is generally overlooked that designers use functional models more frequently than behavioral or gate-level models. In functional modeling, the functionality of one or more components, like arithmetic/logic units, memories, and counters, are described as separate concurrent blocks. We present an algorithm, called Component Synthesis Algorithm (**CSA**), for synthesis from these functional descriptions. Our algorithm automatically synthesizes components needed to implement a functional description while minimizing hardware costs and performance. Since a functional description uses standard operators in the hardware description language, a mismatch between the operators of the language and the functionalities provided by library components arises. **CSA** solves this functionality mismatch problem by pattern matching of the description against a library of function patterns. In addition, **CSA** clusters functions to maximally match components from a given library. Experimental results show that automated functional synthesis produces designs that are comparable to those produced by human designers.

Key Words: Functional Synthesis, Component Modeling, Functionality Recognition and Reduction, Register-Transfer Level Technology Mapping, Compatibility Graph.

Contents

1 INTRODUCTION 1

2 RELATED RESEARCH 4

2.1 Behavioral Level Synthesis 4

2.2 Register-Transfer Level Technology Mapping 5

2.3 Logic Level Synthesis 6

3 PROBLEM DESCRIPTION 7

3.1 The Input Description 7

3.2 Formulation of the Component Synthesis Problem 7

4 OUR APPROACH TOWARDS COMPONENT SYNTHESIS 11

5 FUNCTIONALITY RECOGNITION 14

6 COMPONENT MAPPING 16

6.1 Reformulation of the Component Mapping Problem 16

6.2 Compatibility Graph Reduction 17

6.3 Operator Merging 19

6.4 A Heuristic Function for Compatibility Edge Selection 21

6.5 The Component Mapping Algorithm 22

6.5.1 The Heuristic Component Mapping Algorithm 22

6.5.2 The Branch-and-Bound Component Mapping Algorithm 23

6.6 Evaluation of CSA 26

7 EXPERIMENTAL RESULTS 26

7.1 Experimental Setup 26

7.2 Test Series 1 27

7.3 Test Series 2 29

8 CONCLUSIONS AND FUTURE WORK **32**

A APPENDIX: THE COST FUNCTION **37**

A.1 The Overall Cost Function 37

A.2 The Area Cost Function 37

A.3 The Delay Cost Function 38

B APPENDIX: THE BOUND FUNCTION **39**

B.1 The Overall Bounding Function 39

B.2 The Area Bounding Function 40

B.3 The Delay Bounding Function 41

B.4 The Correctness Of The Bounding Function 42

B.4.1 The Correctness of the Area Bounding Function 43

B.4.2 The Correctness of the Delay Bounding Function 43

B.4.3 The Correctness of the Total Bounding Function 44

List of Figures

1	Motivation.	2
2	An Input Description Example.	8
3	A Flow Graph Example.	9
4	The **CSA** Block Diagram.	12
5	**CSA** on the Adder/Subtractor Design Example.	13
6	A Function Recognition Example.	14
7	The Functionality Recognition Algorithm.	15
8	A Compatibility Graph Formation Example.	17
9	A Compatibility Graph Reduction Example.	18
10	An Example of the Operator Merging Process.	20
11	The Heuristic Component Mapping Algorithm.	22
12	The **B&B** Component Mapping Algorithm.	24
13	**CSA's** Search Space for the Adder/Subtractor Design.	25
14	A Bounding Function Example.	40
15	Bounding Function for the Area/Delay Trade-off.	42

List of Tables

1 A Comparison Matrix of Design Quality and Algorithm Performances. 28

2 A Comparison Matrix for Design Quality Using Three Different Libraries. 30

1 INTRODUCTION

High-level or behavioral synthesis involves mapping an algorithmic specification into a set of micro-architecture components that implement the desired behavior while satisfying a set of constraints [16]. Examples of such components include arithmetic/logic units, counters, comparators, registers, memories, and interconnect units, such as multiplexors and buses. Designers however rarely use behavioral descriptions. Instead, they most often use *functional* descriptions. A functional specification describes one or a group of components as separate concurrent blocks (using concurrent statements in the case of VHDL). A typical example of such a functional description, namely, of an arithmetic/logic unit called ALU1, is given in the middle of Figure 1. A main reason for using functional descriptions is that it captures the structural and physical views most familiar to designers.

Figure 1 shows the relationship between high-level [22, 13, 16, 26, 27], functional, and logic level synthesis [3, 2, 9]. High-level synthesis algorithms are concerned with mapping a *behavioral* description of the desired system to a (generally generic) register-transfer level structure that performs that behavior. Functional synthesis or component synthesis on the other hand synthesizes a *functional* description of one or possibly several register-transfer level components to component(s) from a given library that perform the same functionality. A *behavioral specification* is temporal, i.e., it is a procedural description of an algorithm or a set of sequential actions to be executed over time. On the other hand, a *functional specification* is spatial, i.e., it is a description of the functionality of one (or possibly a group of) micro-architecture components. These components modeled be a functional description could be generic components, technology-specific components, or even newly designed architectural portions of a design. Lastly, logic-level synthesis corresponds to automated design and logic optimization at the gate level. Functional synthesis thus fills the gap between behavioral and logic level synthesis.

As can be seen in Figure 1, functional synthesis tools can work in synergism with high-level synthesis tools by mapping (functional descriptions of) register-transfer level designs produced by the high-level synthesis onto actual hardware. On the other hand, the functional input description is often also directly entered by modelers who are more familiar with the register-transfer level of design than with the algorithmic level. The designers think in terms of major components that compose the overall design; and then give a functional description of each of these subsystems. These subsystems (components or groups of components) can be synthesized separately by component and logic synthesis tools. Furthermore, designers may also utilize the functional description style to describe the components of existing designs. Reasons for producing such descriptions of an existing design are manifold: first, the design is simulatable at the functional level if a language, such as VHDL, is used, and second, the specification can serve as design documentation. In addition, it is a perfect basis for redesign into a new technology by synthesizing the description using components from a new technology.

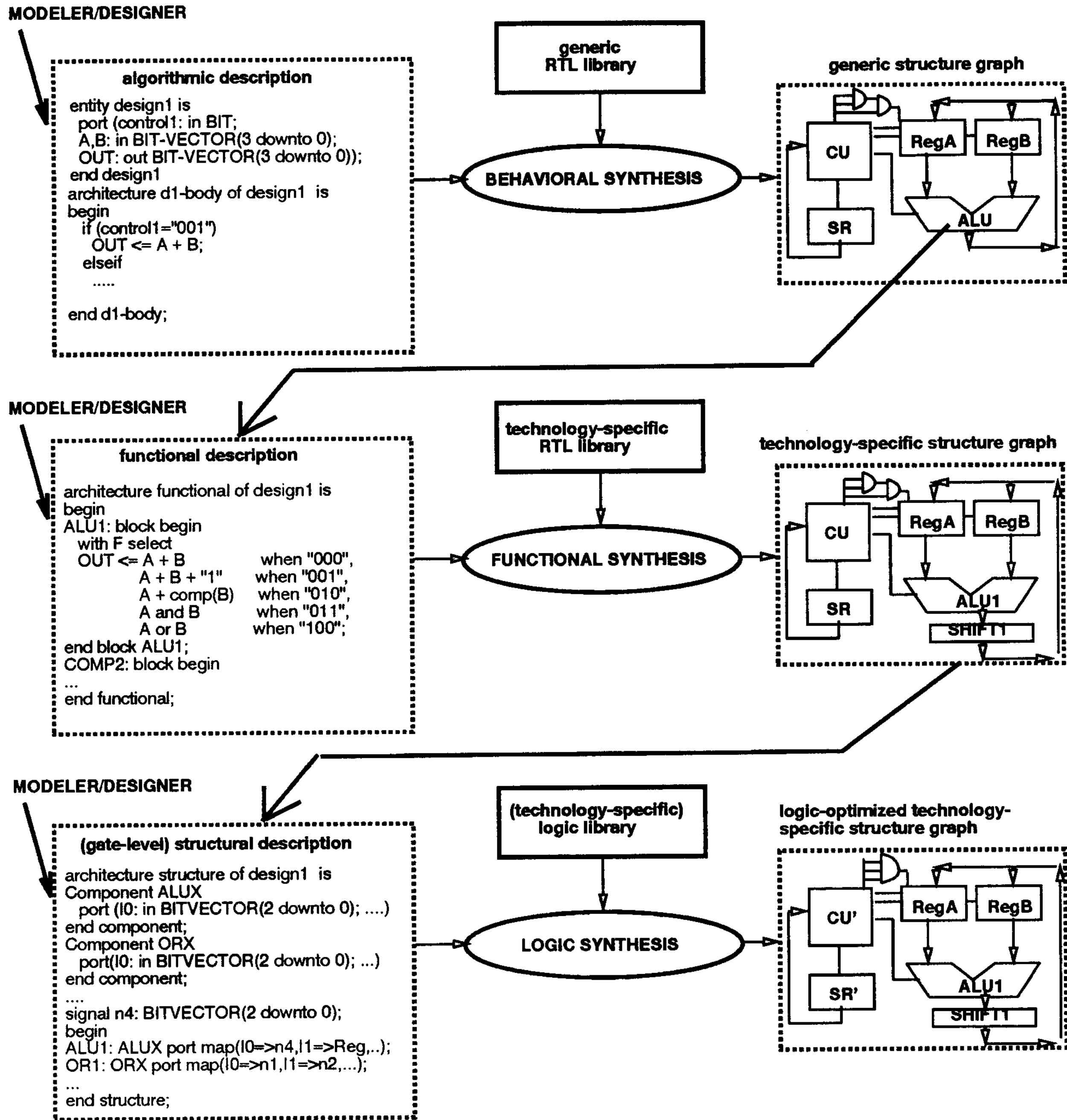


Figure 1: Motivation.

Functional synthesis thus not only solves the *technology mapping* problem on the register-transfer level, but it also addresses the important problem of efficient *redesign* and *technology adaptation*. In order to stay competitive, companies must be flexible enough to quickly redesign their products by replacing obsolete components used in their design with components from a new technology. Performing this redesign manually is a tedious and labor intensive task. It is also extremely costly. We propose to automate this process of retargeting to a new technology by automatically synthesizing a given component description to (a set of) new components from a given component library.

The use of hardware description languages such as VHDL for specifying functional descriptions has given rise to new problems. Functional descriptions of components vary drastically. The modeler may not have an exact description of the component he or she wants to model. Furthermore, the language may provide several different ways to describe the same functionality. Also, of course, in a functional description a single component is described by several, often nested conditional, statements. These statements need to be grouped in order to map them to the same component. To complicate matters even further, the functional descriptions of components are expressed by standard operators in the description language that may not match the functions supported by the target library components.

Since functional descriptions are used for concurrent finite-state designs, scheduling needed for high-level synthesis [26, 22] is not applicable to functional synthesis. Therefore, sharing of units for operators in different time steps (states) is not needed. However, functional descriptions contain nested conditional statements; and operators used in different branches of these conditional statements can share units. This unit sharing across conditional statements is exactly what is exploited by functional synthesis. High-level synthesis often hides conditional expressions in micro-code, and thus does not address the sharing of units due to conditional expressions. An exception to this is [30] which presents an algorithm for conditional resource sharing that generates efficient micro-code control sequences for nested conditional branches as well as for straight line codes.

Usually, high-level synthesis is divided into three stages: allocation, scheduling, and binding. These three tasks, though being tightly interdependent, are generally solved independently because of the complexity of each task. Since scheduling is not relevant to functional synthesis, we can perform the tasks of allocation and binding simultaneously. Therefore, interconnection costs can be considered simultaneously with the hardware allocation costs.

Another difference between behavioral and functional synthesis is that register merging as done in high-level synthesis [26, 22] is not relevant to functional synthesis. We assume that the design representation has been optimized to remove redundant intermediate variable references that had been introduced by a hardware description language. Registers are then needed for *all* remaining variable references because the design is concurrent. This reduces the component synthesis problem to an optimization problem on the combined operator and interconnection costs.

Most binding techniques that minimize the connectivity costs of a design fail to incorporate the added cost of control into their optimization procedures [22]. The Component Synthesis Algorithm (**CSA**) presented in this paper generates control logic, which corresponds to the function select code of a multi-functional unit. For this reason, **CSA** can include appropriate control costs into the cost function.

Related research is presented in Section 2. The functional synthesis problem is formalized and our solution is outlined in Sections 3 and 4, respectively. One algorithm of **CSA**, called functionality recognition algorithm is described in Sections 5, while the second algorithm, called component mapping algorithm, is described in Section 6. Experimental results and conclusions are discussed in Sections 7 and 8, respectively. Lastly, the cost and bound function used by the component mapping algorithm are presented in Appendix A and B. Our earlier work on this problem has been published in [24].

2 RELATED RESEARCH

Below, we compare our work on functional synthesis with research in behavioral-level synthesis, register-transfer level technology mapping, and logic-level synthesis.

2.1 Behavioral Level Synthesis

As discussed in the previous section, the tasks of allocation and binding are related to the problem of functional synthesis. Therefore, we compare the techniques proposed by the high-level synthesis community to address these two problems with our approach.

Tseng [26] was among the first to formulate allocation and binding as clique partitioning problems. EMERALD uses a heuristic approach towards the problem by assigning profit measures for merging nodes into the same cluster for each pair of nodes. Then, a greedy method is applied where all nodes that fall into the category with the highest profit are clustered first, all the ones in the next highest profit group next, and so on. EMERALD handles only straight-line blocks of assignment statements. Also, operators are grouped into sets of functions that do not necessarily correspond to the functionalities supported by existing micro-architecture components. In our work, we overcome this problem by prepruning the solution space based on the mergeability information derived from the given component library. The profit function used by EMERALD for operator merging is somewhat unrealistic. For instance, the merging of an addition with a multiplication operator is considered to be equivalent to the merging of an addition with a subtraction operator as long as both have the same number of common sources and sinks. CSA uses a more accurate profit function that estimates the cost of the design in terms of silicon layout. It considers for instance the bit width of units.

HAL (Hardware Allocator) [22] applies the clique partitioning method to the register and interconnection binding problem. It assumes that scheduling, allocation of functional units as well as unit binding are completed. Similar as in EMERALD, all potential merges with a weight (profit) above a certain threshold are executed. Rather than doing this in a greedy fashion it is done exhaustively within each profit group. The overall control strategy is greedy, however, and thus cannot guarantee an optimal solution. An additional drawback of this work is that control costs are not included into the cost function.

Splicer [19] uses an heuristic approach towards connectivity binding. Given a fixed resource allocation and a schedule, Splicer minimizes the number of connections between functional units and registers by using a branch-and-bound search with the number of multiplexers as criterion. It uses a heuristic function in place of a proper bounding function to more effectively prune the search space. This however removes the guarantee of finding an optimal solution. Similarly as Splicer, the CSA algorithm is based on the branch-and-bound methodology. CSA, however, combines this methodology with the clique partitioning approach for operator merging. Hence, CSA succeeds in pruning the search space substantially without losing the guarantee of an optimal solution.

Lastly, most of the cost functions used by high-level synthesis systems incorporate area measures, while our algorithm handles both area and delay optimization.

2.2 Register-Transfer Level Technology Mapping

A technology mapping step of translating a netlist of generic register-transfer level (RTL) components into a technology-specific RTL structure is sometimes performed between high-level and logic-level synthesis.

Leive was one of the first to address this problem [11, 12]. SYNNER takes a localized approach to the problem by selecting a component from a given library based on some local criterion for one data path node at a time ([12], page 479). Leive and Thomas ([12], pg. 480) write: "This selection process takes a narrow view of optimization in that internode dependencies are not considered". In other words, the constraints of area or delay are only utilized for selecting among the candidate modules for a single RT-node. No absolute design goals, such as, the minimal total area, can be handled by this local optimization strategy. Concurrently with technology mapping, SYNNER performs some logic optimization by reducing certain cascaded logic operations into one logic operation, e.g., it replaces two 2-bit ANDS by one 3-bit AND ([12], page 31). This early work also attempts to utilize its localized approach for merging units that are never active in the same states, a task which nowadays is handled by high-level synthesis.

Dutt and Kipps [5] describe an approach of mapping generic RTL components into technology-specific RTL library cells using the rule-based system DTAS [10]. Their work addresses technology

mapping of a known generic component with a fixed set of functions into a technology-specific component. On the other hand, functional synthesis introduced in this paper synthesizes a possibly complex *functional description*, which could be a description of either generic components, technology-specific components, or even newly designed architectural portions of a design, into component(s) from a technology-specific library.

2.3 Logic Level Synthesis

Work presented in the literature on logic synthesis generally addresses two tasks: logic optimization and technology mapping. In logic optimization, a technology independent logic specification is usually processed using algebraic and/or Boolean methods, such as, two-level minimization [3] and algebraic decomposition [2]. These techniques remove redundant logic and make use of common terms. In logic technology mapping, the optimized Boolean equations are transformed into an interconnection of technology-specific logic elements from a given library of gates [9, 2, 14]. Logic technology mapping itself consists of three tasks [14]: first partitioning into an interconnection of single-output sub-networks, then the decomposition of each sub-network into an interconnection of two-input functions, e.g., AND and OR, and lastly the covering of each sub-network by an interconnection of library cells.

Functional synthesis is also concerned with technology mapping, though, at a higher abstraction level of design. Functional synthesis matches a graph representing a (possibly technology-independent) functional design against a library of technology-specific patterns. One difference between technology mapping at the functional level and at the logic level is that the number of different patterns in functional synthesis is much smaller than in logic synthesis. A Boolean function can be described by many different combinations of logic operators. On the other hand, the functions of components currently available in micro-architecture libraries are rather simple and thus can be described by one statement. Consequently, technology mapping in functional synthesis will result in less potentially overlapping matches of technology-specific patterns on the generic design representation. Therefore, **CSA** uses a simple pattern matching and reduction algorithm rather than the sophisticated dynamic programming algorithm proposed by [9] for the covering subtask of logic synthesis.

Mailhot and De Micheli [14] extend the work by Keutzer [9] by using Boolean matching techniques based on Shannon decomposition. The proposed matching process checks the tautology between a given Boolean function (the network) and the set of functions represents a library element for any permutation of its variables. **CSA** instead relies on a simple matching procedure due to the simplicity of the functions of register-transfer components.

3 PROBLEM DESCRIPTION

In this section, we introduce the functional or component synthesis problem. Functional synthesis synthesizes register-transfer components needed to implement a given *functional* description while minimizing the underlying hardware costs and the design delay. Functional synthesis is composed of two subprograms: *functionality recognition* and *component merging.* First, the functionality recognition step attempts to utilize the complex functions supported by register-transfer components from the library for implementing a given functional description. The input description is however expressed by language operators that do not correspond directly to these functions. The functionality recognition problem thus is similar in flavor to technology mapping, since it matches functions provided by real components with operators described by a hardware description language. Thereafter, the component mapping step allocates components from a given library to implement the functional specification while minimizing hardware costs and performance. During this phase the component mapper maps mutually exclusive operators of the input description to the same component, whenever possible.

3.1 The Input Description

The input for functional synthesis is a *functional description* of a design written in a hardware description language[1]. An example of a typical input description is shown in Figure 2. The description consists of three concurrent conditional statements. The first and second statement form a nested conditional statement, since the variable tmp1 is produced by the first and consumed by the second. The statements labeled by values v_i following a select condition correspond to different branches of the condition, that is, they are disjoint and only one of them will be executed. The input description is translated by a compiler into an internal flow graph representation, an ECDFG graph [23] [2]. Figure 3 depicts the ECDFG representation of the example description given in Figure 2.

3.2 Formulation of the Component Synthesis Problem

The functional synthesis problem can be stated using the following graph theoretic formulation. Let $O = \{op_1, op_2, ..., op_n\}$ be a set of operators, and let $U = \{u_1, u_2, ..., u_m\}$, also called the **unit table,** be a set of functional unit types. Each unit $u_i \in U$ is capable of performing a subset of the functions in O, denoted by **functionality**$(u_i) \subseteq O$. Let the function **op-cost**$(u_i,\ bw(u_i))$ represent a cost estimate for each unit $u_i \in U$ in terms of the required silicon layout area with $bw(u_i)$ the bit width

[1]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.

[2]The ECDFG design representation [23] is an extension of the commonly known Control/Data Flow Graph model [18] with concepts, such as, timing constraints, structural bindings, asynchronous events, etc.

```
with r5 select
tmp1 <= r6 or r7              when v6,
        r7 and r8             when v7;

with r1 select
r16 <= guarded
    (r1 + r2) + r4            when v1,
    r2 - r4                   when v2,
    tmp1                      when v3;

with (r9=r10) select
r16 <= guarded
    r11 + r12                 when v1,
    (r10 + r12) - (r13 + r14) when v4;
```

Figure 2: An Input Description Example.

of u_i. Let the function **delay**(u_i, $bw(u_i)$) represent a delay estimate for each unit $u_i \in U$ in terms of the propagation delay with $bw(u_i)$ the bit width of u_i.

Let $G = (V, A)$ be a directed flow graph with $V = \{v_1, v_2, ..., v_k\}$ the set of nodes and $A = \{a_1, a_2, ..., a_l\}$ the set of edges. An edge $a_i = < v_j, v_k >$ represents the data dependency between the nodes v_j and v_k. V is composed of three disjoint sets $V = N \cup R \cup D$. $N = \{n_1, n_2, ..., n_k\}$ represents the set of operators, $R = \{r_1, r_2, ..., r_l\}$ the set of storage elements, and $D = \{d_1, d_2, ..., d_m\}$ the set of decision nodes that model data selection. Figure 3 represents such a graph G with operator nodes N, storage elements R, and decision nodes D depicted by circles, rectangles, and triangles, respectively. The edges A are depicted by solid arrows. The function **operation**: $N \rightarrow O$ defines the operation modeled by each operator node. For instance, the operator node n_4 executes the logic operation *or*, denoted by **operation**(n_4)=*or*.

The connection points of an edge $a_i = < v_j, v_k >$ with the nodes v_j and v_k are called the output and input ports, respectively. The ports of a node v_i are denoted by PORTj(v_i) with j = 1, ..., #(ports of v_i). A decision node $d_i \in D$ has three or more input ports with the first input port being of type control and all others of type data. Mutually exclusive decision values (constants) are associated with the data input ports of a decision node. A comparison of these decision values with the value at the control input port determines which of the data input values is selected by the decision node.

Example 1 *Figure 3, for instance, represents such a graph G, that is a representation of the functional design description given in Figure 2. In Figure 3 the decision node d_1 has one control input port PORT1(d_1) guarded by r_1 and three data input ports PORT2(d_1), PORT3(d_1), and PORT4(d_1) with the decision values being v1, v2, and v3, respectively. Hence, if r_1=v1 then the value connected to PORT2(d_1), namely, $(r_1 + r_2) + r_3$, will be selected by d_1, if r_1=v2 then the value connected to PORT3(d_1), namely, $r_2 - r_4$, will be selected by d_1, and so on.*

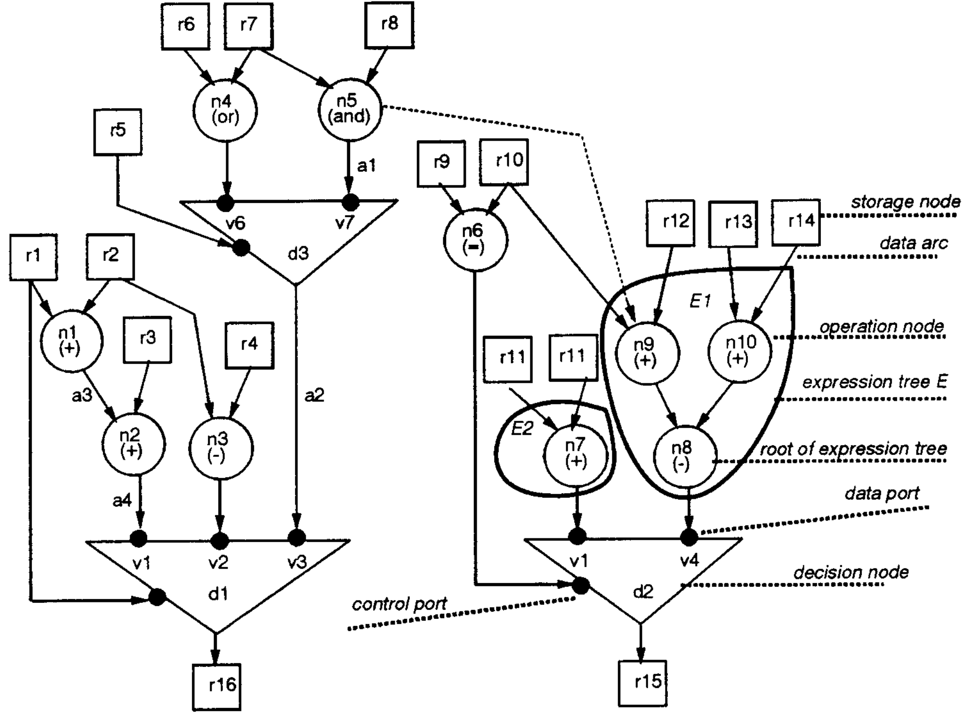


Figure 3: A Flow Graph Example.

A **path** in G is defined to be an ordered list of nodes and edges of G of the form, $(v_1, a_1, v2, a_2, v_3, ..., v_i, a_i, v_{i+1}, ..., v_k)$ where v_1 is the start node and v_k the end node of the path, and each edge a_i is connecting its predecessor in the list, v_i, with its successor in the list, v_{i+1}, that is, $a_i = < v_i, v_{i+1} >$. If v_i and d_j are an operator and decision node in G, respectively, and P_{ij} are the directed paths of data flow edges from v_i to d_j in G, then we denote the set of the data input ports of d_j at which the directed paths P_{ij} connect with d_j by PORT-SET-$v_i(d_j)$.

Then we define the *mutually exclusiveness* of operator nodes as follows: if the condition which selects one operator always falsifies the condition selecting the other, then the two operator nodes are mutually exclusive. More specifically, in the graph theoretic terminology given above mutually exclusiveness is defined as follows.

Definition 1 *Two nodes v_j and v_k are* **mutually exclusive**[3] *in G if there exists a node $d_i \in D$ in G, and all directed paths of data flow edges on which v_j lies lead to data input ports of d_i (denoted by PORT-SET-$v_j(d_i)$) and all directed paths of data flow edges on which v_k lies lead to data input ports of d_i (denoted by PORT-SET-$v_k(d_i)$), and PORT-SET-$v_j(d_i) \cap$ PORT-SET-$v_k(d_i) = \emptyset$.*

Example 2 *In Figure 3, the two operator nodes n_1 and n_5 are mutually exclusive because there exists one directed path of data flow edges from n_1 and n_5 to d_1, respectively, and the path from n_1 to d_1, $< n_1, a_3, n_2, a_4, d_1 >$, ends at PORT1($d_1$), and the path from n_5 to d_1, $< n_5, a_1, d_3, a_2, d_1 >$, ends at PORT3($d_1$). If due to some graph optimization, an additional data flow edge would have been inserted between node n_5 and n_9 ($< n_5, n_9 >$ is depicted as a dashed arrow in Figure 3), then n_5 would have a fanout of two. In this case, the two nodes n_1 and n_5 are no longer mutually exclusive because not all paths from n_5 go through the decision node d_1.*

An **expression tree,** $G_i = (V_i, A_i)$, is defined to be a connected subgraph of G consisting only of operator nodes, i.e., $V_i \subseteq N$. One node is designated as the root, and all paths in G_i are directed from the leaves towards the root. $V_i \subseteq$ N, and $A_i \subseteq$ A. Further, G_i is a complete subgraph of G in the sense that for all pairs of nodes n_j, $n_k \in V_i$, if there is an edge $a_l = < n_j, n_k >$ in A then the same edge a_l also exists in A_i. The function **op**: G $\rightarrow$ $(O \cup \emptyset)$ is a mapping from an expression tree G_i to the operation described by G_i. G_i may trivially correspond to a single operator node n_i, and then, by default, **op**(n_i) is defined to be the primitive operation represented by n_i, namely, **op**(n_i) = **operation**(n_i).

Example 3 *On the right hand side of Figure 3 we mark the expression tree E_1 as consisting of the three operator nodes n_8, n_9, and n_{10}. Let us denote the operation described by the expression tree E_1 by op_{E_1}. Then the function* **op**: *$G \rightarrow (O \cup \emptyset)$ is a mapping from the expression tree E_1 to the operation described by E_1, denoted by* **op**(E_1) *$= op_{E_1}$. This implies that there is a hardware unit in the given library that can directly implement the operation expressed by E_1 as one function, called the op_{E_1} function. The second marked expression tree E_2 corresponds to a single operator node n_7. Therefore, by default,* **op**(E_2) *is defined to be the primitive operation represented by n_7, namely,* **op**(E_2) = **operation**(n_7) = "+".

P is defined to be the collection of all partitions P_i of the graph G into subgraphs G_i. M is defined to be the collection of all *mappings* M_i: P $\Rightarrow 2^U$ with U the unit table. A mapping M_i from a partition P_i of G to a set of functional units from U is defined to be a *legal mapping* iff and only if the following constraints are fulfilled:

1. For all $G_i \in P_i$, **op**(G_i) $\in$ **functionality**($M_i(G_i)$).

[3]**CSA** actually utilizes a more general notion of mutually exclusiveness based on conditional expressions rather than on the existence of decision nodes; see discussion in Section 6.1 and in report [25].

2. Each pair of root nodes n_i of G_i and n_j of G_j with $M_i(G_i) = M_i(G_j)$ must be **mutually exclusive** in G.

The first constraint requires that all operator nodes (or expression trees) that are mapped to the same unit are compatible in functionality. The second requirement states that all operator nodes mapped to the same function unit by the mapping M_i have to be mutually exclusive. Both thus are necessary requirements for ensuring the correctness of the resulting design.

Lastly, the *component synthesis problem* is to find a tuple (P_i,M_i) where $P_i \in$ P is a partition of G into subgraphs G_i and $M_i \in$ M is a legal mapping from P_i to a set of functional units from U which minimizes the cost of the resulting design. The cost of a design is measured by a weighted sum of the area and the performance of the design. A precise definition of the cost function is given in Appendix A.

4 OUR APPROACH TOWARDS COMPONENT SYNTHESIS

In this section, we outline our overall approach in solving the functional synthesis problem defined in the previous section, while more detailed algorithms will be presented in later sections. A top-level block diagram of **CSA** is given in Figure 4. The library-specific information used by **CSA** (see left hand side of Figure 4) is kept in two tables, the functionality table and the unit (mergeability) table. The unit table contains a unique name for each unit, a list of all functions implemented by the given unit u, called **functionality**(u), and a function that measures unit cost, called **op-cost**(), and a function that measures the propagation delay of the unit, called **delay**(). The functionality table enumerates the non-primitive functions that can be executed by components of the input library and gives a corresponding pattern in terms of generic operators. These two tables have to be designed once for each technology. It can be done automatically by parsing an input library description into two tables, or it can be done manually.

CSA consists of two modules, called **functionality recognizer** and **component mapper**. In addition, it uses a graph compiler as pre-processor and a netlist generator as post-processor. The system can be summarized as follows:

(1) A language compiler [13] parses the input description, a functional description, into an internal design representation, an Extended Control/Data Flow Graph (ECDFG) [23]. An example input description and the corresponding ECDFG are given in Figure 5.a and 5.b, respectively.

(2) The **functionality recognizer** addresses the mismatch problem between the operators of the language and the functions supported by library components. It merges expression subtrees into single nodes whenever components are capable of executing those expressions as one function. An example of a partition of the ECDFG into expression trees is given in Figure 5.b, while Figure

5.c shows the resulting reduction of the graph. In Figure 5.c, the reduced operator nodes AI and SI (abbreviations for the terms Add-and-Increment and Subtract-and-Increment) correspond to the expression trees "A + B + 1" and "A - B + 1", respectively.

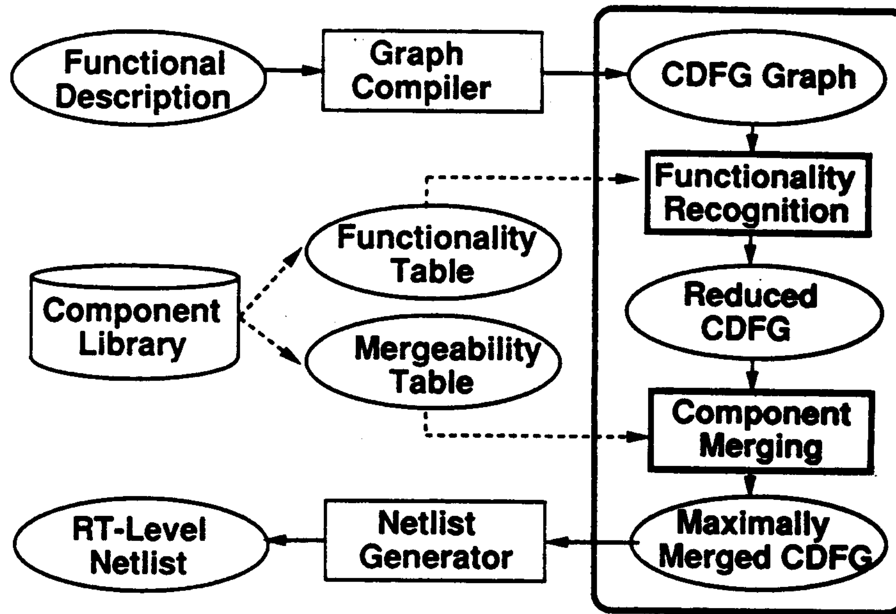


Figure 4: The **CSA** Block Diagram.

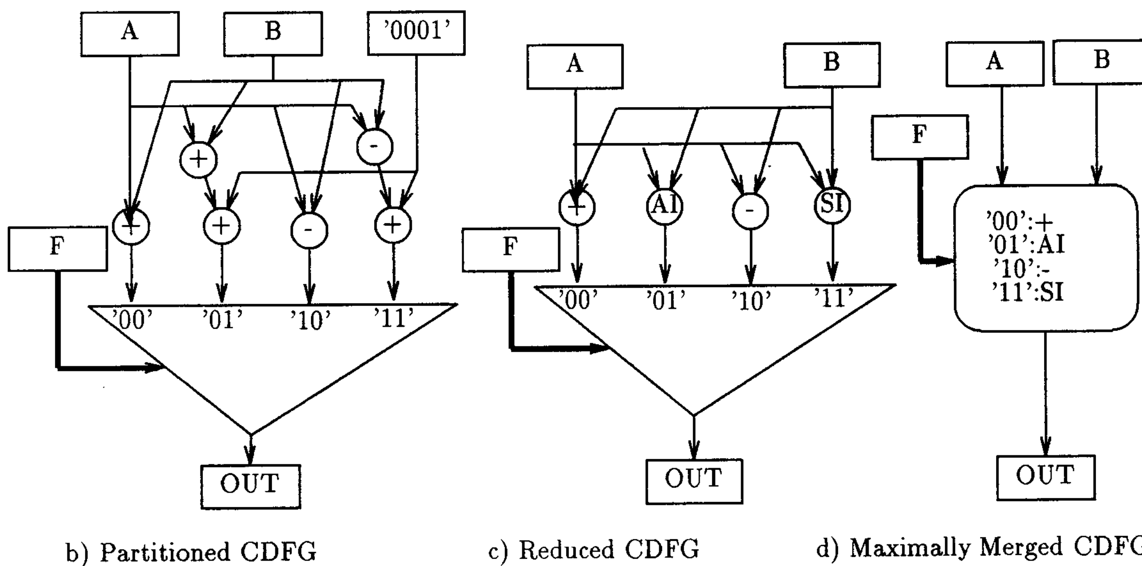
(3) The **component mapper** addresses the problem that functional descriptions use multiple statements to describe the functionality of a single component. It thus merges mutually exclusive operators of the reduced graph to minimize hardware costs and design delay. Figure 5.d shows how the four mutually exclusive operator nodes of Figure 5.c are merged into one multi-functional operator node.

(4) Each merged operator node is mapped to a micro-architecture component from the given library. The netlist generator then creates a netlist of the resulting structure. In Figure 5.d, for instance, the final design is produced by mapping the multi-functional operator node to an Adder/Subtractor unit.

The functionality recognition and the component mapping algorithms are described in the remainder of this paper; while the graph compiler and the netlist generator are discussed elsewhere [13].

```
entity design1 is
  port (F: in BIT-VECTOR(1 downto 0);
        A, B: in BIT-VECTOR(3 downto 0);
        OUT: out BIT-VECTOR(3 downto 0));
end design1;
architecture design1-body of design1 is begin
 with F select
  OUT <= A + B             when "00",
         A + B + "0001"    when "01",
         A - B             when "10",
         A - B + "0001"    when "11";
end design1-body;
```

(a) VHDL description



b) Partitioned CDFG c) Reduced CDFG d) Maximally Merged CDFG

Figure 5: **CSA** on the Adder/Subtractor Design Example.

5 FUNCTIONALITY RECOGNITION

A functional description uses standard language operators that do not always correspond directly to component functions. The functionality recognition algorithm solves this **functionality mismatch** by transforming a design representation of generic operator nodes into a representation consisting of library-supported operator nodes.

For example, in Figure 6 the expression "A+B+1" is originally compiled into two operator nodes *n1* and *n2*. This expression can however be implemented as a single function of an ALU. The graph structure can be modified accordingly by merging *n1* and *n2* into one node *n3*, performing Addition and Incrementation.

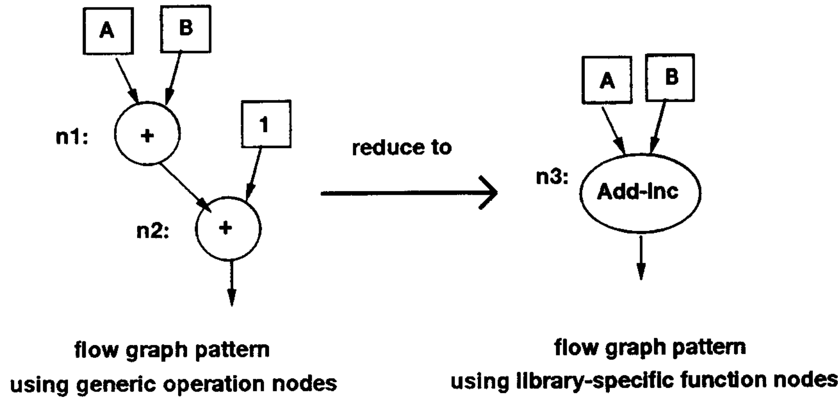


Figure 6: A Function Recognition Example.

The library-specific information used by the algorithm is kept in the **functionality table** *F*. *F* stores the graph patterns of expression trees that are implemented as single functions by library components (Figure 6). Associated with each pattern P is a pair of cost reduction measures, (A_P,D_P), with A_P and D_P the measures for the expected improvement in hardware area and in delay costs when implementing the pattern P by a single component function, respectively. Both measures are a function of the number and type of operators in the pattern and are determined a priori. For example, the pattern *P1="A+1"* might have the cost reduction measure pair (A_{P1}, D_{P1}) = (1,1), while the pattern *P2* = *"A+B+1"* might have the cost reduction measure pair (A_{P2}, D_{P2}) = (2,2). Finally, the total cost reduction measure of a pattern P is determined as follows:

$$\textit{cost-reduction}(P) = \alpha \times A_P + \beta \times D_P$$

with the constants α and β, with α, $\beta \in [0{:}1]$ and β = 1.0 - α, input parameters entered to **CSA** by the designer. The constants α and β correspond to the relative importance of area versus delay optimization. In particular, if the area parameter α = 1.0 (i.e., β=0.0) then the function measures exclusively the area cost of the design, and hence **CSA** will optimize exclusively for area. If the delay parameter β = 1.0 (i.e., α=0.0) then **CSA** will optimize exclusively for performance. Finally, for all other values of α and β, **CSA** will optimize both for area and delay with the relative degree of importance being α and β, respectively.

Input: a *flow graph G with generic operator nodes, a functionality table F*
Output: a *flow graph G' with library-specific operator nodes*
Algorithm:
while *there is an unmarked operator node n in the flow graph do:*
    Match the library function patterns P_i of F
    against the expression trees G_i in G of which node n is the root.
    if *one or more match is found,*
    then begin
        Set P to the pattern with the largest cost reduction according to F.
        Replace the subgraph G_j of G that corresponds to P
        by one operator node m_i with **op**(m_i) *taken from F.*
        Mark m_i.
    end
    else begin
        Mark n.
    end
end while

Figure 7: The Functionality Recognition Algorithm.

The functionality recognizer uses the pattern matching and reduction algorithm given in Figure 7. This algorithm matches the function patterns captured in the table F against the graph G. It traverses the graph G in a bottom-up manner, such that each operator node $n \in G$ is visited once. For each operator node $n \in G$, the function patterns $P_i \in F$ are matched against the expression tree $G_i \in G$ rooted at n. If more than one match is found, then the pattern P_i with the largest cost reduction is selected. For instance, the pattern "A+B+1" will be chosen over the pattern "A+1". Once a pattern P_i has been selected, the subgraph structure G_i of G that corresponds to P_i is reduced to one operator node $n3$ with $\mathbf{op}(n3) = \mathbf{op}(G_i)$ as shown in Figure 6.

A reduced node will not participate in any further pattern matching, as the pattern descriptions kept in F are described exclusively with primitive operator nodes. Hence, pattern matching is

completed in one **pass through** the graph. This assures that the functionality recognizer finds a partition P of G **into subgraphs** G_i, and thus the first subproblem described in Section 3.2 is solved. Each element of P corresponds to one node in the completely reduced graph G'. If the node is a library-specific function, then it corresponds to a subgraph G_i of the original G. If it is a generic operator nodes, then it corresponds to a trivial subgraph G_i of size one. For instance, the graph in Figure 5.b is partitioned into four patterns, two library-specific functions and two generic ones. Thus, the reduced flow graph in Figure 5.c consists of only four operator nodes. Since pattern matching is completed in one pass through the graph, the algorithm's complexity is O((Size of G) $\times$ (Size of the Pattern Set)).

6 COMPONENT MAPPING

The component mapping algorithm solves the problem of merging mutually exclusive operators by reformulating it as a clique partitioning problem.

6.1 Reformulation of the Component Mapping Problem

Two operator nodes *n1* and *n2* are defined to be **mergeable** with respect to a given unit table U if and only if there is a unit $u \in U$ with **functionality**$(u) \supseteq$ **op**$(n1) \cup$ **op**$(n2)$. They are defined to be **mutually exclusive** to each other if the condition which selects one operator always falsifies the condition selecting the other, and vice versa. For a graph-theoretic definition of the term mutually exclusive see Section 3. Details of the condition encoding scheme, the treatment of default values (namely, the **otherwise** clause in a conditional statement), and the recognition of identical conditions (an example of equivalent conditions is given below) can be found in [25]. Two operator nodes *n1* and *n2* are said to be **compatible** with respect to U, if they are both **mutually exclusive** and **mergeable.** **CSA** creates a **compatibility graph** $CG = (N,E)$ from a flow graph $G = (V,A)$ with N the set of operator nodes from V, and E a set of undirected edges, called **compatibility edges.** There is an edge $e_i = < n_j, n_k >$ in CG for each pair of **compatible** nodes n_j, $n_k \in N$.

Next, we demonstrates how a flow graph is transformed into a compatibility graph using a simple design example.

Example 4 *A data flow graph and its corresponding compatibility graph are presented in Figure 8.a and 8.b, respectively. They share the same set of operator nodes. First an edge is inserted between each mutually exclusive operator node pair (Section 3). For instance,* n_2 *is mutually exclusive with* n_3 *and* n_4 *because of decision node* d_1*. Additional mutually exclusive operator nodes can be found by comparing conditional branch expressions. For instance, the expression trees* E_1 *and* E_2 *correspond to equivalent (in this case, identical) conditional expressions, namely, "S1=S2", and therefore the corresponding decision nodes* d_1 *and* d_2 *thus capture the same condition. Therefore, the Plus operation*

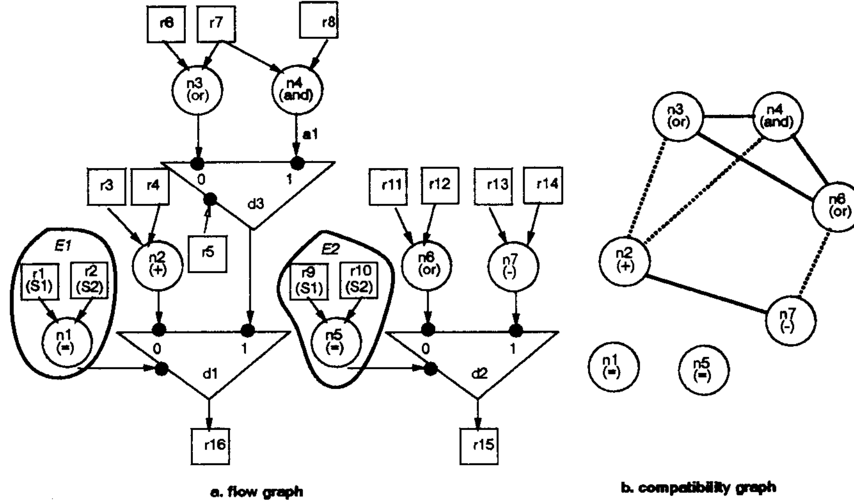


Figure 8: A Compatibility Graph Formation Example.

(n_2) attached to d_1 is found to be mutually exclusive to the Minus operation (n_7) attached to d_2. There are seven mutually exclusive pairs, i.e., seven arcs in the compatibility graph. After this, we need to assure that the selected node pairs are also mergeable. For this example, we assume a component library that contains purely arithmetic units and purely logic units. For this library, all dashed lines which connect logic with arithmetic operations represent non-mergeable pairs, and they have to be removed from the graph. The final compatibility graph contains the four solid arcs.

6.2 Compatibility Graph Reduction

Note that a collection of operator nodes can be mapped to the same functional unit if and only if they are pairwise **compatible**. This follows from the two constraints discussed in Section 3.2. In the terminology defined at the beginning of this section, this means that a subgraph of CG completely connected by compatibility edges, also called as clique, can be mapped to one unit. We express this in our design representation by merging a group of operator nodes, that will be mapped to the same functional unit, into a multi-functional operator node.

During the process of incrementally creating clique covers on the compatibility graph CG, **CSA** adjusts the graph as described below in order to correctly maintain its structure. This reduction of the compatibility graph corresponds to an additional pruning of the search space.

Proposition 1 *Let CG be a compatibility graph, n an operator node and m a newly created multi-functional node composed of the original operator nodes n_1, n_2, ..., n_j with $n \neq n_i$ for all i from 1*

to j. A compatibility edge e_k = <n,n$_k$ > (for some $k \in \{1,...,j\}$) can only be used for future merges if and only if n is compatible with all nodes n_i, i.e., the edges e_i = <n,n$_i$> exist in CG for all $i \in \{1,...,j\}$. If this is the case, the following two **edge reductions** *follow:*

- **equivalent edge property:** *If all edges e_i = <n,n$_i$> exist in CG for $i \in \{1,...,j\}$, then they are called* **equivalent** *to one another. Thus, the CG can be reduced by replacing all of them by one edge, e = <n,m>.*
- **illegal edge property:** *If there is at least 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, the CG can be reduced by deleting all of them.*

In other words, during the clique formation process **CSA** deletes compatibility edges that need no longer be considered for one of two reasons: either their usage would violate the clique property (**illegal edge**), or they have become equivalent to other edges in *CG* and thus will be covered when considering those edges (**equivalent edge**).

An example of how proposition 1 is used to reduce a compatibility graph is given in Figure 9.

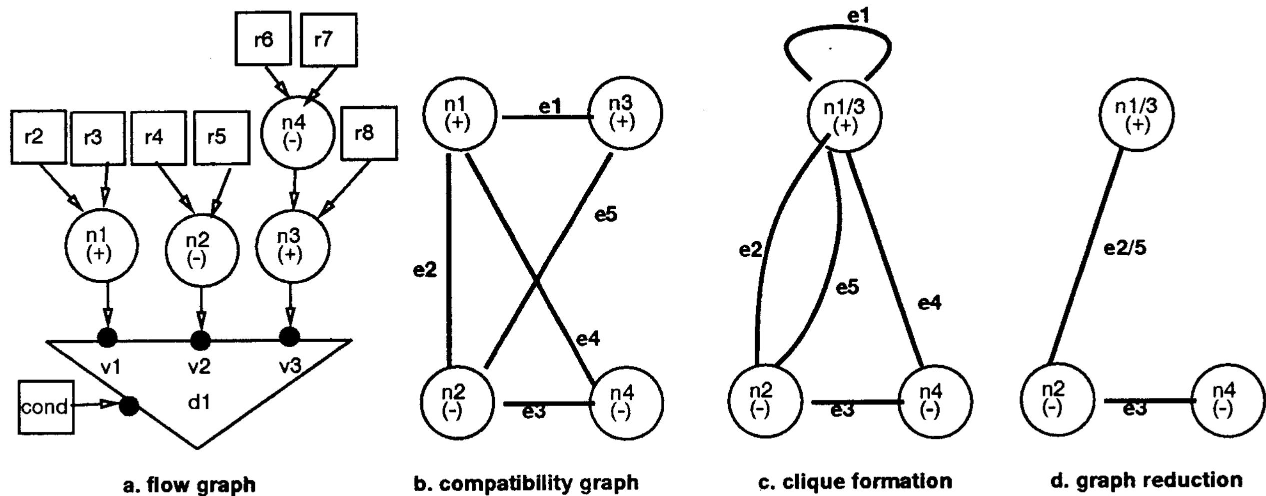


Figure 9: A Compatibility Graph Reduction Example.

Example 5 *Figure 9.a shows a flow graph G with four operator nodes n1, n2, n3 and n4. Note that nodes n3 and n4 are not* **mutually exclusive**, *because they lie on a common path in G. All other*

pairs of nodes are **mutually exclusive**. *Assume that all operator nodes in G are mergeable by a given unit table. Then the compatibility graph CG shown in Figure 9.b can be derived from G. If n1 and n3 are merged into one multi-functional node n (Figure 9.c), then edges $e2$ and $e5$ are equivalent and can be collapsed into one edge $e2/e5$. The edge $e4$, however, violates the illegal edge property and must be removed. The final graph is shown in Figure 9.d.*

In the previous example, the merge of two nodes reduced the compatibility graph from five to two compatibility edges.

6.3 Operator Merging

When generating a partial solution, **CSA** not only adjusts the compatibility graph but also the underlying flow graph. This allows for the simultaneous consideration of (1) the connections costs due to the sharing of units (decision nodes), and (2) the control logic costs for the selection of the correct function of a multi-functional unit (decoder nodes). We outline below how the sharing of units is expressed in the flow graph.

CSA explores the two options of whether or not to map two nodes n_1 and n_2 to the same hardware unit (expressed by the compatibility edge $e = < n1, n2 >$). Correspondingly, **CSA** transforms a flow graph G and its matching compatibility graph CG in one of two ways:

- **case 1:** Map n_1 and n_2 to the same unit.
  - $CG \Rightarrow CG'$ by adjusting the compatibility edges according to the rules described in Section 6.2, and
  - $G \Rightarrow G'$ by directly reflecting the sharing of units in the flow graph as will be described below.
- **case 2:** Do not map n_1 and n_2 to same unit.
  - $CG \Rightarrow CG'$ by simply deleting edge e from CG, and
  - $G \Rightarrow G'$ by simply setting $G' = G$.

Once all decisions for merging/not merging operator nodes have been made, i.e., $CG = \emptyset$, then the solution pair (G, CG) is called a *complete* solution. Otherwise, it is called a *partial* solution.

Case 1, i.e., the mapping of two operator nodes to the same hardware unit, is expressed in the flow graph by merging the two nodes into one multi-functional operator node. The *semantic equivalence* of the original and the transformed flow graph is assured by inserting decision nodes that select the correct inputs for the multi-functional operator node. We handle the *function select*

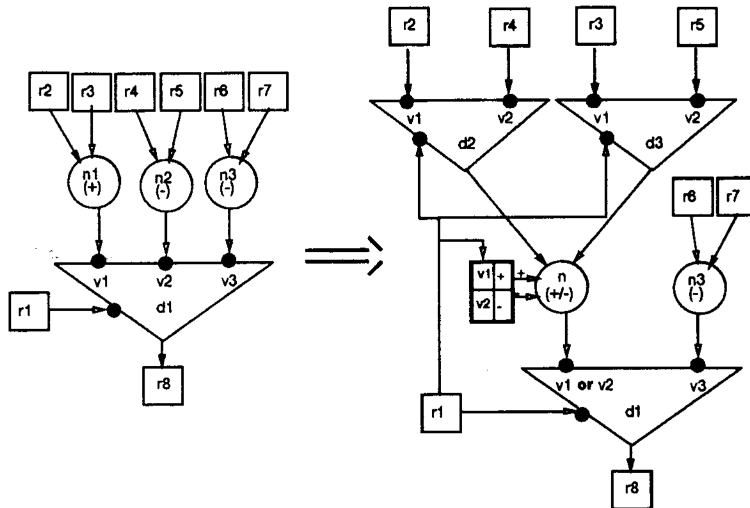


Figure 10: An Example of the Operator Merging Process.

of multi-functional operator nodes by encoding function select conditions and storing them in the associated decoder nodes (depicted by a dark box in Figure 10).

The process of operator merging is best explained with an example. Figure 10 shows how the two compatible operator nodes n_1 and n_2 with functionalities "+" and "−" are merged into one multi-functional operator node n. The node n is controlled by a decoder node which implements the function select logic, i.e., it represents the conditions v_i under which n's respective functions "+" and "−" are executed. For instance, the functions "+" and "−" are controlled by the conditions $r_1 = v_1$ and $r_1 = v_2$, respectively. The new node n is connected to the original output destinations of both n_1 and n_2. If n_1 and n_2 were directly connected to the same decision node, as is the case in Figure 10, then the size of the decision node is reduced or it may even be completely removed. In the example at hand, this corresponds to the fact that the two decision values v_1 and v_2 are merged into one value $(v_1 + v_2)$. Similarly, the input edges to the newly created multi-functional node n are adjusted to connect to the original inputs of n_1 and n_2. Decision nodes may have to be introduced to select among the inputs to n, as can be seen in Figure 10. For a detailed discussion on this see [25].

This example points out the trade-off involved in determining 'optimal merges' of operator nodes. This example merge reduces, for instance, the hardware costs in two respects: it reduces (1) the number of operator units from two to one and (2) the size of the decision node. However, new costs accrue in the form of (1) more complex components (a multi- rather than single-function unit), (2) increased interconnection costs (two additional input decision nodes), and (3) increased control cost

(the decoder to select among the two functions of n). **CSA** trades off between the costs and gains of each such merge based on the heuristic function introduced in the next section.

6.4 A Heuristic Function for Compatibility Edge Selection

While the cost and bound functions are described in Appendix A and B, respectively, the heuristic function to evaluate the gain of mapping two operator nodes to the same hardware unit is given next. This function, which is an extension of the heuristic function developed in [25], associates with each compatibility edge an estimate of the benefit it is likely to bring to the solution (see the discussion in the previous section). The component mapping algorithm described in the next section uses this heuristic for the selection of the compatibility edge that is to be used next to form a clique.

The benefit of mapping two operator nodes to one unit, denoted by *benefit()*, is a measure of the change in area costs and in design performance due to the operator merge. *benefit()*: N $\times$ N $\Rightarrow$ Real, with N the set of operator nodes in the graph G, is a function from a pair of operator nodes to a cost value defined by:

$$\textit{benefit}(n_1, n_2) = \alpha \times \textit{area_benefit}(n_1, n_2) + \beta \times \textit{delay_benefit}(n_1, n_2)$$

with $n_1, n_2 \in$ N, and α and β the relative area and delay optimization parameters defined in a previous section.

The area benefit of mapping two operator nodes to one unit, denoted by *area_benefit()*, is the sum of the three measures described below. If the two operator nodes n_1 and n_2 are merged into one node n then the benefit in terms of hardware operator costs of executing this merge, $n = < n_1, n_2 >$, is *area-operator-costs*(n) := *bound_node_area*(n) - *bound_node_area*(n_1) - *bound_node_area*(n_2) (with *bound_node_area()* the minimal cost of implementing an operator node as defined in Appendix B). If the resulting multi-functional node has more than one functionality, then the cost for a decoder to select among them is added to the measure. The second measure, *area-connection-cost()*, corresponds to the connection costs that result from the sharing of units. This takes into account the number and size of decision nodes that have to be inserted to disambiguate between different inputs to multi-functional operator nodes [25]. The third measure, called *ancestor-mergeability()*, evaluates the potential of direct ancestors of the merged nodes for being merged as well. For each pair of operator nodes that are directly connected by data flow output edges as inputs to n_1 and n_2, respectively, and that are compatible and thus could be merged in the future, increment the third measure by one. This cost evaluation accounts for the fact that a merge directly atop the current merge is likely to reduce the connection costs by making decision nodes redundant.

The delay benefit of mapping two operator nodes to one unit, denoted by *delay_benefit()*, is the sum of the two measures described below. If the two operator nodes n_1 and n_2 are merged

into one node n then the benefit in terms of delay of executing this merge, $n = < n_1, n_2 >$, is *delay-operator-costs*(n) := max(*bound_node_delay*(n) - *bound_node_delay*(n_1), *bound_node_delay*(n) - *bound_node_delay*(n_2)). (with *bound_node_delay()* the minimal delay of implementing an operator node as defined in Appendix B). The second measure, *delay-connection-cost(),* corresponds to the maximum of changes in delay due to the new interconnections of the merged node. This takes into account the number and size of decision nodes that have to be inserted to disambiguate between different inputs to multi-functional operator nodes. The *delay_benefit()* function is a local measure of the effect of the operator merge on the delay.

6.5 The Component Mapping Algorithm

The Component Mapping algorithm reformulates the component synthesis problem as a clique partitioning problem on the compatibility graph (CG) [26]. The goal is to find a minimal cost clique partition of the set of operator nodes of CG such that each clique can be mapped to one multi-functional unit. We present two algorithms to solve this problem: (1) the heuristic (or greedy) component mapping algorithm and (2) the branch-and-bound component mapping algorithm.

6.5.1 The Heuristic Component Mapping Algorithm

Input: a flow graph F, a unit table, the area/delay parameters α and β.
Output: a flow graph optimized for area cost and/or for delay.
Algorithm:
Generate a compatibility graph CG for the flow graph F (Sec.6.1).
while (there is an edge e in CG with $benefit(e)$>0)
  (1) Select edge e=$< n_1, n_2 >$ from CG with the largest heuristic $benefit(e)$ (Sec.6.4).
  (2) Map n_1 and n_2 to the same unit by transforming F and CG (Sec.6.2 and 6.3).
end while;
Return the transformed flow graph F.

Figure 11: The Heuristic Component Mapping Algorithm.

The heuristic component mapping algorithm [25] is described in Figure 11. The algorithm is based on the heuristic function, *benefit()*, which associates with each compatibility edge e a measure of the benefit of using this edge for clique merging (Section 6.4). In a greedy fashion, it repeatedly selects the edge e with the largest benefit and utilizes it for operator merging. (Section 6.3). It stops, when no more profitable edge remains.

6.5.2 The Branch-and-Bound Component Mapping Algorithm

The component mapping algorithm described in this section (Figure 12) replaces the greedy method by a branch-and-bound control strategy. The algorithm uses the inclusion (or exclusion) of a compatibility edge as the branching criterion (Section 6.3). At each node in the branch-and-bound search tree, the solution space is partitioned into two sets of potential solutions according to whether or not a given compatibility edge is used in the clique cover. If an edge is (not) used in a solution, then the corresponding pair of operator nodes is (not) being mapped to the same unit.

All partial solutions (leaf nodes in the current search tree) that have been generated and that can lead to a potentially complete solution are maintained in a list, called active-stack. While there is any node left in this list, the algorithm selects one of them by the last-in-first-out scheme. Then the heuristic function, *benefit()*, is used for branch selection, i.e., to determine the next most profitable compatibility edge for merging (Section 6.4). **CSA** selects the edge e with the largest benefit, *benefit(e)*.

Then, it expands the current solution node by generating its two children. The first child uses the selected edge e for operator merging, while the second child eliminates the edge e so that it will not be used for operator merging (Section 6.3). For the generation of partial solutions we utilize the clique property to reduce the compatibility graph as demonstrated in Section 6.2.

If either of the two children cannot lead to a least-cost solution based on the bounding function presented in Appendix B, then it is discarded. A solution is discarded if the bounding function results in a cost larger than or equal to the cost of the best solution that has been found by the algorithm so far (BSF). Remaining children are kept as potential future solutions in the active list. If one of the children represents a complete solution, then it is compared against the current best solution to determine the new best solution. This process is repeated until either no partial solution remains in the active list or the time limit given by the user is exceeded.

CSA traverses the branch-and-bound search space in a last-in-first-out manner. This allows us to backtrack over the solution space rather than having to keep all partially explored solutions. This reduces the amount of storage for intermediate solutions. In addition, this scheme generates an initial 'good' solution fast, i.e., in polynomial time.

Below, we explain the main features of the branch-and-bound component mapping algorithm based on an example. Figure 13 shows the portion of the branch-and-bound search space for the Adder/Subtractor design example (Figure 5) that is being traversed by **CSA**. This bounded design space consists of 7 nodes, while the complete search space consists of 30 solution nodes (see [24]).

Example 6 *The flow graph of the the Adder/Subtractor design that serves as input to the component mapper is depicted in Figure 5.c. From this, a compatibility graph which corresponds to the first*

Input: a flow graph, a unit table, an iteration count limit,
and the area/delay parameters α and β.

Output: a flow graph optimized for area cost and/or for delay.

Algorithm:

```
Transform the flow graph into a compatibility graph (Sec.6.1).
active-stack = { 0 };
BSF = empty; (Best Solution Found)
UP = ∞; (Upper Bound = cost of BSF)
while ((active-stack ≠ ∅) and (iteration-count>0)) do begin
 (1) Pop branching node b (partial design solution) from active-stack.
 (2) Select edge e with the largest benefit() based on the heuristic (Sec.6.4).
 (3) Generate children of b with child_1 = b + e, child_2 = b − e (Sec.6.2 and 6.3).
 (4) Calculate lower bounds of the two children: bound(child_i) (App.B).
 (5) for i = 2 downto 1 do
        if bound(child_i) ≥ UP
        then discard child_i;
        else
            if child_i is a complete solution (Sec. 6.3)
            then
                if cost(child_i) < UP
                then child_i replaces the best solution found,
                    i.e., UP:=cost(child_i); BSF:=child_i;
                else discard child_i;
                end if;
            else push child_i onto active-stack;
            end if;
        end if;
     end for;
 (6) decrement iteration-count.
end while;
Return BSF.
```

Figure 12: The **B&B** Component Mapping Algorithm.

(partial) solution node S1 is derived using the technique discussed in Section 6.1. **CSA** *selects the next branching edge, e1, based on the heuristic function described in Section 6.4. Now, the two children of S1 are created; they are called S2 and S7. The left child S2 is created by using edge e1 in the clique cover. That is, the operator nodes Add-Inc and Subtract-Inc that are connected by edge e1 are merged into one multi-functional operator node (meaning, they are mapped to the same multi-functional hardware unit). Furthermore, the clique property is utilized to reduce the compatibility*

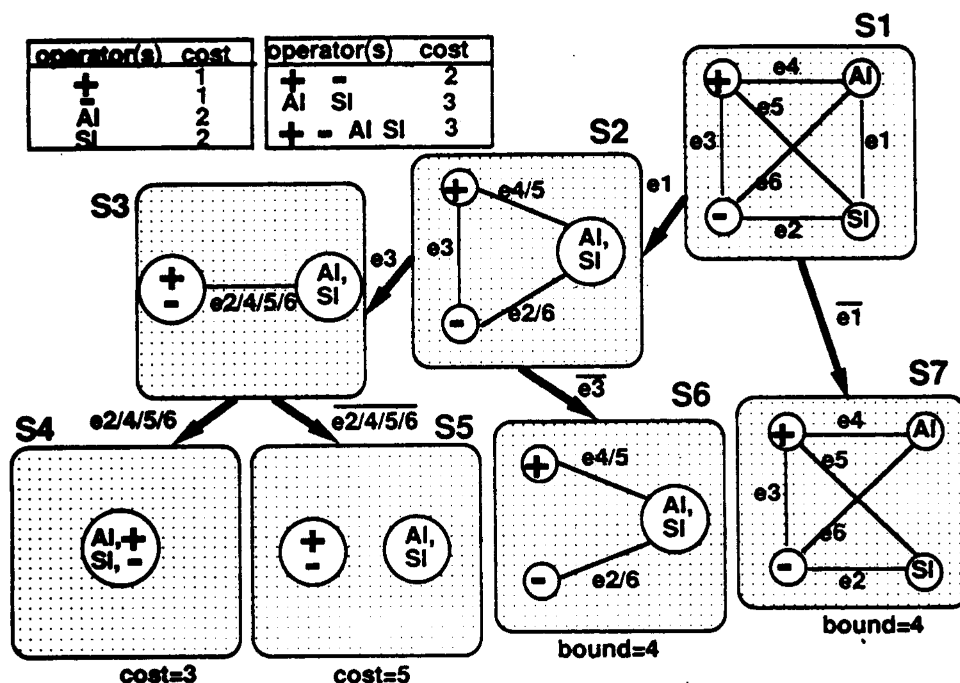


Figure 13: **CSA**'s Search Space for the Adder/Subtractor Design.

graph as demonstrated in Section 6.2. The right child S7, on the other hand, is created by deleting edge e1 from S1. Consequently, the operator nodes Add-Increment and Subtract-Increment will not be merged into one multi-functional operator node in any solution derived from S7. S7 is pushed on the active stack, and **CSA** *continues with S2.* **CSA** *repeats this process until either a complete solution or a solution that can be bounded is found. The first complete solution found by* **CSA** *is S4. At this point, BSF = S4, UP = 3, and active-stack=*$\{S5, S6, S7\}$*. Then, S5 is popped off the active-stack. S5 is also a complete solution. However, it is not as good as S4 and is therefore discarded.* **CSA** *inspects the remaining solutions in the active-list in a depth first manner . Both S6 and S7 can be bounded immediately, since their bound is larger than the cost of S4. The final design returned by* **CSA** *is S4, which corresponds to the merged ECDFG graph shown in Figure 5.d.*

6.6 Evaluation of CSA

There are several observations we would like to make about CSA:

1. The branch-and-bound component mapping algorithm has an exponential worst-case complexity. However, we utilize several techniques to prune the search space and therefore have been able to run the complete algorithm on the typical functional descriptions. In particular, the component mapper uses the unit table to preprune the compatibility graph (Section 6.1), it exploits the clique property to reduce compatibility edges during the branch-and-bound search (Section 6.2), and it applies a bounding function to discard branches of potential solutions from the search tree (Appendix B).
2. Note that the branch-and-bound algorithm is guaranteed to find an optimal solution if given sufficient time. This is desirable as most descriptions of register-transfer level components we have come across thus far are reasonably sized. For high-level synthesis, this is not feasible because descriptions of complete designs are generally large.
3. Lastly, this approach allows for trade-off between the quality of the solution and the computation time. The component mapper can improve on initial good solutions until the time limit is met. If no time limit is set then an optimal solution will be found.

7 EXPERIMENTAL RESULTS

7.1 Experimental Setup

CSA currently runs on SUN3/SUN4 workstations under the UNIX operating system. It consists of approximately 15000 lines of C code not including the VHDL input compiler [13]. We present two different experiments. In the first experiment, we explore the features and limitations of **CSA**. In particular, we compare **CSA**'s performance when using the functionality recognition option versus when not using it. We also study the solution quality (both area and delay) achieved by **CSA** for the heuristic and for the branch-and-bound component mapping algorithm. In the second experiment, we are testing **CSA**'s ability for technology adaptation. Therefore, we experiment with replacing the component-specific information by different libraries.

The two experiments are based on the following collection of typical VHDL descriptions of hardware component(s). Example 1, called Add/Sub, is a variation of an adder/subtractor description [6] with complex functions, such as, Add-and-Increment and Add-and-Decrement. Examples 2 and 3, called Mano1 and Mano2, are two different functional description styles of the arithmetic logic unit proposed by Mano ([15], pg. 337 and pg. 242). Example 4 is a functionally reduced version of the TI 74181 ALU [28]. Example 5 is a divide-by-3328 counter design from the Rockwell-Counter

case study [7]. The sixth example, called Count/Logic, describes a combination of a counting and a logic unit [15], however, with partially permuted inputs. Example 7, called AM2901, corresponds to a functional description of the 2901ALU unit including input multiplexors and latches [17]. The last two design specifications Concur1 and Concur2 describe a combination of concurrently executing components. Concur1 includes several input multiplexors that are shared among the different concurrent units. The components in the Concur2 description receive input data from and write output data to a set of register files.

7.2 Test Series 1

The results of the first experiment are listed in Table 1. In order to explore the features and limitations of **CSA**, we have run the nine examples described in the previous section (first column of Table 1) under a number of different parameter settings. For instance, we run both area and delay optimization as indicated by the α parameter in the second column of Table 1. For $\alpha = 1.0$, **CSA** optimizes for area, and for $\alpha = 0.0$, **CSA** optimizes for performance (Appendix A). Furthermore, we ran **CSA** with or without applying the functionality recognition algorithm as indicated by the **FR** parameter in the third column of the table. The next two columns, labeled **Heuristic CM** and **B&B CM**, display the design results achieved by **CSA** using (1) the heuristic and (2) the branch-and-bound component mapping algorithm, respectively. For each, the solution quality of the design is described by the two measures **Area** and **Delay**. **Area** corresponds to the transistor count of the design and **Delay** to the maximal delay through the design. Both are calculated by the cost function given in Appendix A. The computation time of the respective runs measured in CPU seconds is given in the column labeled **CPU**. The last column, labeled **Designer**, displays the results achieved by the human designer (meaning the best possible design we could find by hand).

One goal of this experiment was the evaluation of the usefulness of the branch-and-bound over the greedy control strategy for the component mapping task. Thus the sixth column of Table 1, labeled **"Imp(provement) B&B"**, indicates the percentage of design quality improvement gotten by **CSA** when using the branch-and-bound versus when using the heuristic component mapping algorithm. The improvement is calculated by the formula $\frac{(quality(heuristic) - quality(B\&B))}{quality(heuristic)}$ with the quality measure being **Area** for $\alpha = 1.0$ and **Delay** for $\alpha = 0.0$. We found that **CSA** improves the design quality in almost half of the thirty six example runs when using the branch-and-bound over when using the heuristic component mapping algorithm (indicated in the sixth column by a percentage larger than zero). It is a quality improvement of 18% percent on the average. For the remaining example runs, the best design was found even without running the complete branch-and-bound algorithm. This improved design quality is achieved at an increased running time of the algorithm. Therefore, **CSA** allows for a trade-off between the quality of the solution and the computation time.

Table 1: A Comparison Matrix of Design Quality and Algorithm Performances.

Examples	α	FR	Heuristic CM Area (trans)	Heuristic CM Delay (ns)	Heuristic CM CPU (sec)	B&B CM Area (trans)	B&B CM Delay (ns)	B&B CM CPU (sec)	Imp B&B	Designer Area (trans)	Designer Delay (ns)
Add/Sub	1.0	yes	172	8	1	172	8	1	0%	172	8
Add/Sub	0.0	yes	172	8	1	172	8	1	0%	172	8
Add/Sub	1.0	no	344	16	2	344	16	7	0%		
Add/Sub	0.0	no	792	19	2	344	16	2	16%		
Add/Sub	**Imp FR**		50%	58%		50%	58%				
Mano1	1.0	yes	578	11	2	408	8	9	29%	408	8
Mano1	0.0	yes	408	8	2	408	8	2	0%	408	8
Mano1	1.0	no	726	19	1	716	18	780	1%		
Mano1	0.0	no	828	19	2	716	18	141	5%		
Mano1	**Imp FR**		20%	58%		43%	56%				
Mano2	1.0	yes	464	15	2	464	15	11	0%	408	8
Mano2	0.0	yes	706	17	1	761	14	23	18%	408	8
Mano2	1.0	no	614	24	2	614	24	98	0%		
Mano2	0.0	no	911	20	2	752	19	122	5%		
Mano2	**Imp FR**		24%	15%		24%	26%				
TI-74181	1.0	yes	706	12	2	411	8	127	42%	411	8
TI-74181	0.0	yes	411	8	2	411	8	2	0%	411	8
TI-74181	1.0	no	910	20	2	789	18	672	1%		
TI-74181	0.0	no	909	19	2	789	18	532	5%		
TI-74181	**Imp FR**		22%	58%		48%	56%				
Rockwell	1.0	yes	2170	56	3	2170	56	12	0%	2170	56
Rockwell	0.0	yes	3223	57	3	2912	33	25	42%	2912	33
Rockwell	1.0	no	2170	56	3	2170	56	12	0%		
Rockwell	0.0	no	3223	57	3	2912	33	25	42%		
Rockwell	**Imp FR**		0%	0%		0%	0%				
Count/Log	1.0	yes	409	11	2	409	11	4	0%	364	11
Count/Log	0.0	yes	634	15	1	543	10	6	33%	406	8
Count/Log	1.0	no	409	11	2	409	11	4	0%		
Count/Log	0.0	no	634	15	1	543	10	6	33%		
Count/Log	**Imp FR**		0%	0%		0%	0%				
AM2901	1.0	yes	1038	18	2	964	18	238	7%	964	18
AM2901	0.0	yes	1132	16	3	1132	16	255	0%	1132	16
AM2901	1.0	no	1038	18	2	964	18	238	7%		
AM2901	0.0	no	1132	16	3	1132	16	255	0%		
AM2901	**Imp FR**		0%	0%		0%	0%				
Concur1	1.0	yes	459	13	2	459	13	2	0%	459	13
Concur1	0.0	yes	459	13	2	459	13	2	0%	459	13
Concur1	1.0	no	685	23	2	685	23	33	0%		
Concur1	0.0	no	831	24	2	1133	23	14	4%		
Concur1	**Imp FR**		33%	46%		33%	43%				
Concur2	1.0	yes	379	8	2	379	8	3	0%	379	8
Concur2	0.0	yes	452	8	2	379	8	2	0%	379	8
Concur2	1.0	no	579	18	2	579	18	91	0%		
Concur2	0.0	no	790	19	1	579	18	453	5%		
Concur2	**Imp FR**		36%	58%		36%	56%				

These experiments further show the importance of the functionality recognition algorithm. For each group of example descriptions, the fifth row labeled **"Imp FR"** indicates the percentage of improvement gotten by **CSA** when using the **FR** option over when not using the **FR** option. This design quality improvement is calculated by the formula $\frac{(quality(not_FR) - quality(FR))}{quality(not_FR)}$. The percentage listed in the **Area** column is gotten by plugging the **Area** measures taken from the rows with the parameter setting $\alpha = 1.0$ into the formula; and the percentage listed in the **Delay** column is gotten by plugging the **Delay** measures taken from the rows with the parameter setting $\alpha = 0.0$ into the formula. For 6 out of 9 example design descriptions, **CSA** improves the design quality by an average of 42% when the **FR** option is used over when the **FR** option is not used. This is true for both the heuristic and the branch-and-bound component mapping algorithm and for both area and delay optimization. For all examples the results produced by **CSA** using the **FR** option are better than or equal to (but never worse than) the designs produced by **CSA** without using the **FR** option.

The designs produced by the human designer (last column of the table) are nearly always equivalent to those of the full-blown **CSA** version, i.e., **CSA** using the **FR** option and the **B&B** component mapping algorithm. The designer only produced a better design for two of the examples, namely, Mano2 and Count/Logic. The designer improved the design for the Mano2 description because s/he recognized the fact that the Mano2 description is equivalent to the Manol description. The Mano2 description uses complex expressions, such as "A + ones-complement(B) + 1" and "A + ones-complement(B)" in place of simpler expressions, such as, "A - B", and "A - B - 1" [15]. The latter functions are directly supported by the ALU in the underlying library (and are stored in the associated functionality table), while the former are not. Note that **CSA**'s results could be improved for this example by adding the respective function patterns to **CSA**'s functionality table. The designer produces a better result for the Count/Logic design description because s/he reversed the inputs of a commutative operation. Note that we have purposely composed this example description by permuting the data inputs for some of the functions of the described component.

7.3 Test Series 2

In this second experiment, we explore **CSA**'s ability for technology adaptation by replacing the component-specific information by different libraries. We ran **CSA** with the three libraries: Genus, Mano, and TTL. The Genus library corresponds to the Generic Component Library GENUS developed at UC Irvine [6]. The Mano library contains a number of the components described in [15] in addition to all components from the Genus library. Finally, the TTL library contains the components of the TTL library [28] in addition to all components from the Mano library. The results presented in the previous section (in Table 1) were generated using the TTL library. Rather than providing complete tables for the experiments using the other two libraries, we present a summary in Table 2. The results presented in Table 2 are based on the **CSA** algorithm with the following parameter settings: **B&B** Component Mapping and α=1.

Table 2: A Comparison Matrix for Design Quality Using Three Different Libraries.

Examples	Using TTL Lib						
	CSA			Designer			Area
Examples	Area	Delay	CR	Area	Delay	CR	Improv.
Add/Sub	172	8	yes	172	8	yes	0%
Mano1	408	8	yes	408	8	yes	0%
Mano2	464	15	no	408	8	yes	12%
TI-74181	411	8	yes	411	8	yes	0%
Rockwell	2170	56	yes	2170	56	yes	0%
Count/Log	409	11	*	364	11	yes	11%
AM2901	964	18	*	964	18	*	0%
Concur1	459	13	yes	459	13	yes	0%
Concur2	379	8	yes	379	8	yes	0%
	Using Mano Lib						
	CSA			Designer			Area
Examples	Area	Delay	CR	Area	Delay	CR	Improv.
Add/Sub	344	20	no	344	20	no	0%
Mano1	408	12	yes	408	12	yes	0%
Mano2	587	17	no	408	12	yes	30%
TI-74181	768	15	no	768	15	no	0%
Rockwell	2170	60	yes	2170	60	yes	0%
Count/Log	409	11	*	364	11	yes	11%
AM2901	1160	21	*	1160	21	*	0%
Concur1	566	15	*	566	15	*	0%
Concur2	452	10	*	452	10	*	0%
	Using Genus Lib						
	CSA			Designer			Area
Examples	Area	Delay	CR	Area	Delay	CR	Improv.
Add/Sub	344	20	no	344	20	no	0%
Mano1	716	24	no	716	24	no	0%
Mano2	654	16	no	654	16	no	0%
TI-74181	826	25	no	826	25	no	0%
Rockwell	2170	60	yes	2170	60	yes	0%
Count/Log	409	11	no	409	11	no	0%
AM2901	1160	21	*	1160	21	*	0%
Concur1	566	15	*	566	15	*	0%
Concur2	452	10	*	452	10	*	0%

Table 2 shows that **CSA** using the TTL library outperforms both **CSA** using the Mano library and **CSA** using the Genus library. Similarly, **CSA** using the Mano library outperforms **CSA** using the Genus library. This is so because the TTL library has a set of richer components than the Mano library, and the Mano library has a set of richer components than the Genus library. For instance, **CSA** using the Mano library is able to reduce the second example description, Mano1, to one unit. On the other hand, **CSA** using the Genus library does not reduce the Mano1 design to one unit. There is no unit in the Genus library which directly supports some of the described functions, e.g., the function "A - B - 1". This function would therefore be implemented by two subtractors in sequence, which considerably decreases the performance of the resulting design. For the Mano1 example description, **CSA** using the TTL library also outperforms **CSA** using the Mano library, even though both libraries have ALUs that directly implement all described functions. The TI74181 component in the TTL library however features a better delay characteristic than the corresponding ALU in the Mano library.

In Table 2, we also present the results of human designers using the three libraries. The last column of the table indicates the percentage of area improvement of the designer's result over **CSA**'s result calculated by $\frac{(\mathbf{Area}(CSA) - \mathbf{Area}(designer))}{\mathbf{Area}(CSA)}$. Given a particular library, **CSA** (with the assumed parameter settings) almost always produced the same result as the designer. This is indicated by a 0% percentage of improvement in the last column of the table. As discussed in the previous section, the designer using TTL components was able to produce a better design for the Count/Logic example by using the commutativity of operators. This is also true for the Mano library. It is not the case for the Genus library, however. Here it is more advantageous to implement the function with the permuted input as an individual component, and hence no input multiplexors can be saved by permuting its inputs. As described in the previous section, the designer improved the design of the Mano2 example by rewriting the input specification, i.e., by replacing the expression "A + ones-complement(B)" by the expression "A - B - 1". This trick allows the designer to improve the design for both the TTL library and the Mano library. For the Genus library, however, this is no longer useful because there is no unit in that library that directly supports the function "A - B - 1".

In this experiment, we furthermore study whether **CSA** is able to recognize and properly reduce the component(s) being described by the functional description. In Table 2, the column labeled **CR** (for component recognition) indicates whether **CSA** was able to reduce the functional description to the described component. The **CR** column can take on the three values: "yes", "no" and "*". **CR**=yes means that proper component recognition took place, while **CR**=no means that component recognition did not take place. Finally, **CR**="*" indicates that the algorithm was able to reduce the description to the described component, but also succeeded in finding an alternative and better design implementation for the given description. We can observe that **CSA** is more likely to reduce functional descriptions to their intended component(s) when given more complex component libraries. In other words, the number of component recognitions (**CR**=yes) decreases for simpler component libraries. For the TTL library, CSA recognizes (and possibly even improves) the design

implementation for 88% of the examples; for the Mano library it is 66% of the examples, and finally for the Genus library it is 44% of the examples. This can be explained with the fact that the libraries contain less complex units and therefore more than one unit is required to implement complex functional descriptions.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have defined a new problem, which we call the component or functional synthesis problem. Functional synthesis maps a functional description of one or possibly several register-transfer level components to an interconnection of components from a given library that perform the same functionality. Most research presented in the literature concentrates instead on the synthesis from behavioral (or algorithmic) descriptions.

Furthermore, we define a general approach for the synthesis from such functional descriptions, and we provide a solution to this problem in form of a two-phase algorithm, called **CSA**. The presented component synthesis algorithm, **CSA**, solves this technology mapping problem on the register-transfer level by automatically synthesizing a given functional description to a near minimal set of micro-architecture components. **CSA** also provides a solution to the *functionality mismatch* problem using a pattern matching scheme.

Our experiments show that in most cases the **CSA** algorithm produces a design that is comparable to that of a human designer. In addition, we found that **CSA** improves the design quality in about half of the example runs when using the branch-and-bound component mapping algorithm over when using the heuristic component mapping algorithm. The improvement in design quality (both area and delay) is 18% on the average. **CSA** therefore allows for a trade-off between the quality of the solution and the computation time. The designer can use **CSA** with the heuristic component mapping algorithm to get a good solution within a very short time. If the solution quality is of importance or if the computation time is not so critical, then the **CSA** should be run using the branch-and-bound component mapping algorithm in order to improve on the initially found solution.

Our experimental results have also shown that designs synthesized using the functionality recognition option are smaller in cost than those synthesized without it. **CSA** was able to improve 66% of the example designs when using functionality recognition; and the improvement in design quality (area and delay) was 42% on the average. Therefore we can conclude that functionality recognition is an essential ingredient of functional synthesis. We have thus succeeded in moving the functionality recognition approach, a fairly common ingredient to logic-level synthesis systems, to a higher level of abstraction by utilizing it at the functional synthesis level.

Lastly, our experiments have shown **CSA**'s ability for component recognition and technology adaptation. Given an appropriate library, **CSA** was able to recognize and properly reduce 88% of the component(s) being described by a functional description.

Future work will address the incorporation of the functionality recognition task directly into the component merging phase in order to solve the two problems of expression tree reduction and of grouping functions to hardware units simultaneously. We may also want to study whether there is any gain in replacing the simple pattern matching procedure used by **CSA** for functionality reduction by a more sophisticated method, such as those used in Boolean technology mapping [14].

ACKNOWLEDGEMENTS. This work was supported by NSF grant MIP-8922851, California MICRO grant #89-057, and contributions from TRW Inc. We are grateful for their support. We also thank Joe Lis for the use of his VHDL graph compiler and members of the UCI CADLAB for helpful discussions.

References

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

[2] R. K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang, MIS: A multiple-level logic optimization system, *IEEE Trans. on Computer-aided Design*, Nov. 1987.

[3] R. K. Brayton, et al., *ESPRESSO IIC: Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Netherlands, 1984.

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

[5] N. D. Dutt, and J. R. Kipps, Bridging High-Level Synthesis to RTL Technology Libraries, *Proc. 28th Design Automation Conf.*, pp. 526–529, 1991.

[6] N. Dutt, GENUS: A Generic Component Library for High Level Synthesis, Tech. Report 89-09, Univ. of Cal., Irvine, 1989.

[7] D. Gajski, J. Lis, N. Vander Zanden, and A. Wu, Synthesis from VHDL: Rockwell-Counter Case Study, Tech. Rep. 90-09, Univ. of Cal., Irvine, 1990.

[8] M. Kahrs, Matching a parts library in a silicon compiler, *Proc. Int. Conf. on Computer-Aided Design,* pp. 169–172, 1986.

[9] K. Keutzer, DAGON: Technology Binding and Local Optimization by DAG Matching, *Proc. 24th Design Automation Conf.*, pp. 617–623, 1987.

[10] J. R. Kipps, and D. D. Gajski, Effects of Mixing Design Styles on the Synthesis of RTL Components, Tech. Rep. 91-42, Univ. of Cal., Irvine, 1991.

[11] G. W. Leive, The Design, Implementation, and Analysis of the Automated Logic Synthesis and Module Selection System, Carnegie-Mellon University, Carnegie Institute of Technology, Ph.D. Thesis, 1981.

[12] G. W. Leive and D. E. Thomas, A Technology Relative Logic Synthesis and Module Selection System, *Proc. 18th Design Automation Conf.*, pp. 479–485, 1981.

[13] J. S. Lis and D. D. Gajski, Synthesis from VHDL, *ICCD*, pp. 378–381, 1988.

[14] Mailhot, F. and De Micheli, G., Technology Mapping Using Boolean Matching and Don't Care Sets. *Proc. European Conf. on Computer-Aided Design,* pp. 212 – 216, 1990.

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

[16] M. C. McFarland, A. C. Parker, and R. Camposano, Tutorial on High Level Synthesis, *Proc. 25th Design Automation Conf.*, pp. 330–336, June 1988.

[17] J. Mick and J. Brick, *Bit-Slice Micro-Processor Design,* McGrawHill, 1980.

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

[19] B. M. Pangrle, Slicer: A Heuristic Approach to Connectivity Binding, *Proc. 25th Design Automation Conf.*, pp. 536–541, June 1988.

[20] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity,* Prentice Hall, Inc., Englewood Cliffs, N.J., 1982.

[21] N. Park 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, pp. 356–370, March 89.

[22] G. P. Paulin and J. P. Knight, Scheduling and Binding Algorithms for High-Level Synthesis, *Proc. 26th Design Automation Conf.*, pp. 1 – 6, 1989.

[23] Rundensteiner, E. A., and Gajski, D. D., "A Design Representation Model for High-Level Synthesis", Information and Computer Science Department, Uni. of California, Irvine, Tech. Rep. 90-27, Sep. 1990.

[24] Rundensteiner, E. A., Gajski, D. D., and Bic, L., "Component Synthesis from Functional Descriptions", Information and Computer Science Department, Uni. of California, Irvine, Tech. Rep. 90-24, Aug. 1990; also *Proc. Int. Conf. on Computer-Aided Design,* pp. 208 – 211, Nov. 1990.

[25] E. A. Rundensteiner, D. D. Gajski, and L. Bic, Technology Mapping for Register Transfer Descriptions, Tech. Rep. 89-42, Univ. of Cal., Irvine, Dec. 1989.

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

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

[28] *The TTL Data Book for Design Engineers,* Texas Instruments Incorporated, 2nd Ed., 1967.

[29] *VHDL Language Reference Manual*, Addison Wesley, 1988.

[30] K. Wakabayashi and T. Yoshimura, A Resource Sharing and Control Synthesis Method for Conditional Branches, *Proc. Int. Conf. on Computer-Aided Design,* pp. 62 – 65, 1989.

APPENDIX

A APPENDIX: THE COST FUNCTION

A.1 The Overall Cost Function

We approximate the quality of design solutions in terms of a weighted sum of design area and design delay, while most other related research [26] focuses on area optimization. The cost of a graph G (a complete solution) is approximated by

$$cost(G) = \min_{M_i \in M}(\ \alpha \times area_cost(G/M_i) + \beta \times delay_cost(G/M_i)).$$

The constants α and β, with α, $\beta \in [0{:}1]$ and $\beta = 1.0 - \alpha$, correspond to the relative importance of area versus delay optimization, and are entered as input parameters to **CSA** by the designer. $M_i \in M$ corresponds to a *direct* mapping of the design into hardware components from the given library, where a *direct* mapping means that each operation node is mapped to a different component instance (since the sharing of units between operator nodes is taken care of by explicitly creating multi-functional operator nodes in the graph G as shown in Section 6.3). The function $area_cost(G/M_i)$ corresponds to the area cost of the design when using the mapping M_i, while the function $delay_cost(G/M_i)$ corresponds to the delay cost of the design when using the mapping M_i.

A.2 The Area Cost Function

We approximate the hardware cost (*area_cost*) of design solutions by transistor counts. A significant difference to [26] is that our cost function is parameterized by the chosen component library (captured by the unit table U). U contains the function **op-cost**: U $\times$ Integer $\rightarrow$ Integer, that maps a unit type and its bit width $(u, bw(u))$ to a cost value. For units with operators, such as plus, minus, and comparison, this function is (bit-width $\times$ log(bit-width) $\times$ some unit constant). For units with multiplication or division operators, the function is (bit-width^2 $\times$ some unit constant). Since each operator node n_i is mapped to a functional unit $u \in U$, the cost of an operator node n_i can be expressed by **op-cost**$(M_i(n_i), b)$ with b the bit-width of n_i assuming the mapping M_i. A multi-functional operator node n_i requires a decoder node v_j that captures the function select logic in the form of a truth table. Its cost, denoted by **decoder-cost**(v_j), corresponds to the number of transistors needed to implement the decoding. Interconnection costs are approximated by the cost of the decision nodes $d_i \in D$ in the design representation, since a decision node will be mapped to either a multiplexor or a bus. Thus, the increased control and wiring complexity that results from resource sharing is included in the cost optimization. This is calculated by a piece-wise linear function, **conn-cost**$(choices(d_i), bit-width(d_i))$, which maps the number of choices of a decision node d_i and the bit width of its data operands to a cost value.

Assuming the mapping M_i of G, the total area cost of a graph G is approximated by

$$area_cost(G/M_i) = \sum_{n_i \in N}(\textbf{op-cost}(M_i(n_i), bw(n_i)))$$

$$+ \sum_{v_j \in V \text{ and } v_j \text{ is a decoder-node}}(\textbf{decoder} - \textbf{cost}(v_j))$$

$$+ \sum_{d_i \in D}(\textbf{conn-cost}(choices(d_i), bw(d_i))).$$

CSA is not concerned with register binding. Thus, the cost of storage elements will be constant over all possible designs produced by **CSA**, and can be ignored in this cost function.

A.3 The Delay Cost Function

We approximate the performance (*delay_cost*) of design solutions in terms of the maximal delay through the design. Our cost function is again parameterized by the chosen component library (captured by the unit table U). U contains the function **delay**: U × Integer → Integer, that maps a unit type and its bit width $(u, bw(u))$ to a delay value. Since each operator node n_i is mapped to a functional unit $u \in U$ by a mapping M_i, the delay of an operator node n_i can be expressed by $\textbf{delay}(M_i(n_i), bw(n_i))$ with b the bit-width of n_i. A multi-functional operator node n_i requires a decoder node v_j to select the desired function. Its delay, denoted by $\textbf{decoder-delay}(v_j)$, corresponds to the longest delay path through the decode logic. The interconnection delay is approximated by the delay attributed to the decision nodes $d_i \in D$ in the design representation, since a decision node will be mapped to either a multiplexor or a bus. The delay of a decision node d_i, $\textbf{conn-delay}(choices(d_i), bit-width(d_i))$, is thus a function of the number of its choices and the bit width of its data operands.

Assuming the mapping M_i of G, the total delay cost of a graph G is approximated by

$$delay_cost(G/M_i) = \max_{path\ p_i \in G}($$

$$\sum_{(n_j \text{ on } p_i) \text{ and } (n_j \in N)}(\textbf{delay}(M_i(v_j), bw(v_j)))$$

$$+ \sum_{(v_j \text{ on } p_i) \text{ and } (v_j \text{ a decoder-node})}(\textbf{decoder} - \textbf{delay}(v_j))$$

$$+ \sum_{(d_j \text{ on } p_i) \text{ and} (d_j \in D)}(\textbf{conn} - \textbf{delay}(choices(d_j), bit - width(d_j)))).$$

where p_i is a path in G as defined in Section 3.2.

B APPENDIX: THE BOUND FUNCTION

B.1 The Overall Bounding Function

A bounding function determines a lower bound on the cost of a partial solution, i.e., a lower bound on all complete solutions that can be derived from the given compatibility graph via further operator merging (Section 6.3). This can then be used to bound the search space of the branch-and-bound algorithm. Assume that we associate with each node G in the search space the value $cost_{min}(\mathrm{G})$ = min{ cost(F): F is a complete and feasible solution node in the subtree of G }, or ∞ otherwise. Then the bounding function, *bound()*, must fulfill the following rules [20]:

Requirement 1: *bound(G)* $\leq$ $cost_{min}(G)$ for all nodes G in the search tree.

Requirement 2: *bound(G1)* $\geq$ *bound(G2)* if *G1* is a child of *G2*.

The first requirement states that *bound(G)* is a lower bound on the cost of *any* design implementation of G that can be derived using further operator merging. This guarantees that the bounding function is a true lower bound on the cost function, and it ensures that you won't prematurely discard any potentially useful solution. The second requirement ensures the monotonicity of the bounding function. In other words, the estimated bound of a partial solution is always smaller than or equal to the estimated bound of any possible extension of that partial solution. Thus, if you make a decision to discard the solution branch based on the current value of the bounding function, then the knowledge of any solution further down the search branch would not have proven the current decision wrong. If a function fulfills these requirements, then the following is guaranteed: if the lower bound on a partial solution cost exceeds any complete solution cost found so far, then the partial solution extensions are nonoptimal and may be discarded.

The definition of the bounding function used by **CSA** is given next. The function is a weighted sum of area and delay bounds of a graph G approximated by

$$bound(G) = \alpha \times area_bound(G) + \beta \times delay_bound(G)$$

with α and β equal to the relative importance of area versus delay optimization as defined in a previous section. The function *area_bound(G)* is defined to be the bound on the area cost for any design implementation of G, while *delay_bound(G)* is defined to be the bound on the delay cost for any design implementation of G.

B.2 The Area Bounding Function

The area bounding function, ***area_bound()***, is defined below, while the delay bounding function, *delay_bound()*, is defined in the next section. Let a partial solution G correspond to the compatibility graph *CG*. We denote the minimal cost of implementing an operator node by *bound_node_area()*, i.e., *bound_node_area(n_j)* $= \min_{M_k \in M}(\textbf{op-cost}(M_k(n_j), bw(n_j)))$. A node n in *CG* is ***isolated*** if no compatibility edges are connected with this node. We denote an input decision node of an isolated node n by $D(n)$. Let a subgraph of operator nodes connected (not necessarily completely) by compatibility edges be called a cluster, and let *CLUSTER* stand for the set of all maximal clusters of *CG*. An isolated node is not considered to be a cluster of size 1. Denote the size of a cluster c by $|c|$. Then define the function *ADDON: CLUSTER* $\rightarrow$ Integer as follows. For $c \in$ *CLUSTER*, if there is one node *n1* $\in c$ with the number of compatibility edges connected to *n1* less than $|c| - 1$, then let $n2$ be a node in c with *n1* $\neq n2$ that is not connected to *n1* and set *ADDON(c)* = min (*bound_node_area(n_1)*, *bound_node_area(n_2)*). Otherwise *ADDON(c)* = 0. Then the area bounding function, *area_bound*(), is defined by

$$area_bound(G) = \textstyle\sum_{isolated\ nodes\ n_i \in N}(bound_node_area(n_i))$$

$$+ \textstyle\sum_{isolated\ nodes\ n_i \in N}(\textbf{conn-cost}(choices(D(n_i)), bw(D(n_i))))$$

$$+ \textstyle\sum_{c \in CLUSTER}(\ \max_{n \in c}(bound_node_area(n)) + ADDON(c)).$$

This definition of the area bounding function is best explained by an example.

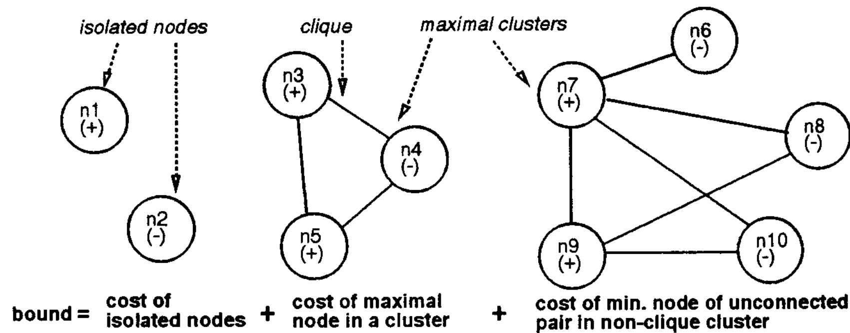


Figure 14: A Bounding Function Example.

Example 7 *In this example we show how the area bounding function is evaluated for the compatibility graph CG shown in Figure 14. We don't give the associated ECDFG graph, and therefore can only bound the operator hardware costs of the partial design. Assume that all operators have a bit width of*

one. Assume the following hardware costs for the ten operator nodes n_i: bound_node_area(n_i) = 15 if i is an even number and 10 if i is an odd number for i = 1, ..., 10. Then bound(CG) is calculated as follows. Nodes n1 and n2 are isolated nodes. Therefore, bound(CG) is incremented by 10 for n1 and 15 for n2. CG has two maximal clusters, namely, n3 to n5 form one cluster C1 and nodes n6 to n10 form a second cluster C2. Thus, bound(CG) is incremented by 15 for C1 and by 15 for C2. Cluster C1 is a clique, therefore no ADDON is calculated for C1. C2 is not a clique. Therefore, each pair of its nodes that is not connected by compatibility edges will never be mapped to the same hardware unit. Let us pick the pair n8 and n10. Hence, bound(CG) can be increased by the lesser cost of the two nodes, which is ADDON(C2) = min (bound_node_area(n_8), bound_node_area(n_{10})) = min(15,15) = 15. In total, bound(CG) = 70.

B.3 The Delay Bounding Function

The delay bounding function, *delay_bound()*, is defined below. Again let a partial solution G correspond to the compatibility graph CG. Assume all definitions given above. In addition, we define a decision node to be *stable* if its data inputs are either operator nodes or data access nodes, and none of its input operator nodes share compatibility edges among one another. Next, we define the function *bound_node_delay:* $V \rightarrow$ Integer with V the set of vertices of G as follows:

$$bound_node_delay(v_j) = \begin{cases} \min_{M_k \in M}(\mathbf{delay}(M_k(v_j), bw(v_j))) & v_j \in N \\ \mathbf{decoder-delay}(v_j) & v_j \in V \; and \\ & v_j \; is \; decoder-node \\ \mathbf{conn-delay}(choices(v_j), bw(v_j)) & v_j \in D \; and \\ & v_j \; is \; stable \\ 0 & otherwise \end{cases}$$

with **delay**(), **decoder-delay**(), and **conn-delay**() the delay measures discussed above. We use p_i to denote a path in G as defined in Section 3.2. The total delay bound of a graph CG is approximated by

$$delay_bound(G) = \max_{p_i \in G}(\textstyle\sum_{v_j \; on \; p_i}(bound_node_delay(v_j)))).$$

This definition of the delay bounding function is explained by an example below.

Example 8 *In this example we show how the delay bounding function is evaluated on the design shown in Figure 8. For this example assume that all operator and decision nodes can be implemented with a maximal delay of 2. The decision node d_2 is stable because its data input nodes n_6 and n_7 are not mergeable with one another, i.e., bound_node_delay(d_2)=2. The decision nodes d_1 and d_3 are however not stable. Consequently, bound_node_delay(d_1) = 0 and bound_node_delay(d_3) = 0. The delay bounding function therefore returns the maximum path delay value of 4 based on the paths* <

n_6, $d_2 >$ and $< n_7$, $d_2 >$. The cost function, on the other hand, would return the maximum path delay value of 6 by counting in the decision nodes d_1 and d_3 on the paths $< n_3$, d_3, $d_1 >$ and $< n_4$, d_3, $d_1 >$, if the design were complete.

B.4 The Correctness Of The Bounding Function

Before proofing the correctness of the bounding function, we will explain the interplay between the area and delay bounding functions via an example. In Figure 15, the X-axis measures the area and the Y-axis the delay of designs. Let the point PD with the coordinates (Ax, Dx) correspond to a partial design with *area_bound(PD)=Ax* and *delay_bound(PD)=Dx*. Then the bounding function for the partial design PD will result in the point PD', with *bound*(PD) = $\alpha \times Ax + \beta \times Dx$. This point PD' indicates that all solutions derivable by **CSA** from the partial solution PD will be in the solution space in the top-right shaded area above the point PD'. This then implies the cutoff function $\alpha \times \mathrm{Ax} + \beta \times \mathrm{Dx} = \alpha \times \mathrm{X} + \beta \times \mathrm{Y}$, depicted by a dark diagonal line in the figure. To the left of the line are all design solutions which are at least as good as PD' or better. Therefore if **CSA** finds one complete solution to the left of the cutoff line then the partial solution PD and with it all solutions derivable from PD can be discarded.

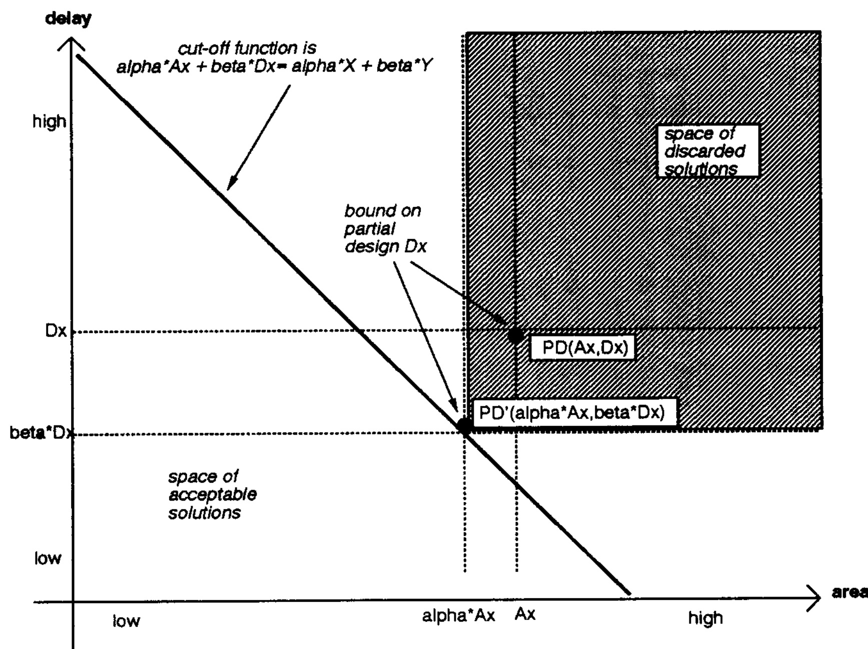


Figure 15: Bounding Function for the Area/Delay Trade-off.

In the following, we first show that the area bounding function is a true lower bound on the area cost function and that the delay bounding function is a true lower bound on the delay cost function. These two proofs can then be used to show that the total bounding function, *bound()*, is a true lower bound on the total cost function, *cost()*.

B.4.1 The Correctness of the Area Bounding Function

Theorem 1 *The area_bound() function is a lower bound on the area_cost() function.*

Proof: Below, we show that the *area_bound()* function fulfills the two bounding requirements given above, and therefore is a true lower bound on the *area_cost()* function.

Requirement 1: *area_bound(G)* $\leq$ $area_cost_{min}(G)$.

The bound function meets this requirement for the following reasons. No further merging and therefore no further cost improvement is possible for isolated nodes. Therefore the first two terms of the *area_bound()* function correspond to the area cost of all *isolated* nodes and their connections. To get a tighter bound, we add the third term to the area_bound function. This term corresponds to the area cost of the most expensive unit within each cluster. This utilizes the following fact: for two operator nodes $n1$ and $n2$, if $n1$ implements a subset of functions of $n2$ then there is a mapping M_i, such that, **functionality**$(M_i(n_1)) \subseteq$ **functionality**$(M_i(n_2))$ and **op-cost**$(M_i(n_1), bw(n_1)) \leq$ **op-cost**$(M_i(n_2), bw(n_2))$. In short, adding more functions to an operator node cannot decrease its area cost. If the cluster is a clique then all operators within the cluster might be mapped to the same hardware unit component (given favorable costs of the multi-functional unit) and thus the cost cannot be further bounded. If the cluster is not a clique then the bounding function is increased by another term, called ADDON. This is so, since such a cluster will be mapped to two or more components.
q.e.d.

Requirement 2: *area_bound(G2)* $\geq$ *area_bound(G1)* if *G2* is a child of *G1*.

It can easily be seen that each further merge of operator nodes can only increase the area bound of a design. This implies that the bounding function also preserves this *monotonicity principle*. **q.e.d.**

B.4.2 The Correctness of the Delay Bounding Function

Theorem 2 *The delay_bound() function is a lower bound on the delay_cost() function.*

Proof: Below, we show that the *delay_bound()* function fulfills the two bounding requirements given above, and therefore is a true lower bound on the *delay_cost()* function.

Requirement 1: $delay_bound(G) \leq delay_cost_{min}(G)$.

The bound function meets the first requirement for the following reasons. No reduction and therefore no further delay improvement is possible for *stable* decision nodes. It is however possible that due to operator merging non-stable decision nodes can be optimized away. For this reason, the delay bound function considers only the delay of stable interconnection units while the delay cost function accounts for the delay of all interconnection units. On the other hand, the delay bound function includes the delays of all operator nodes. This is based on the following assumption: for two operator nodes $n1$ and $n2$, if $n1$ implements a subset of functions of $n2$ then there is a mapping M_i, such that, $\mathbf{delay}(M_i(n_1), bw(n_1)) \leq \mathbf{delay}(M_i(n_2), bw(n_2))$. Or short, adding more functions to an operator node cannot decrease its delay. This implies that if two operator nodes n_1 and n_2 are merged into one operator node n, then the delay of the new node is larger or equal to the delay of either of the two original nodes, i.e., max(*bound_node_delay(n1)*, *bound_node_delay(n2)*) $\leq$ *bound_node_delay(n)*. The merged node n would then lie on both $n1$'s and $n2$'s original paths, and hence the delay bound may increase but not decrease due to operator merging. **q.e.d.**

Requirement 2: $delay_bound(G2) \geq delay_bound(G1)$ if *G2* is a child of *G1*.

The principle that more complex multi-functional operator nodes have larger or equal delays to simpler operator nodes implies that the bound on the delay of operator nodes can only be increased due to operator merging. Similarly, decision nodes may become *stable* due to operator merging, which may further increase the delay cost. Consequently, the bounding function preserves this *monotonicity requirement.*

q.e.d.

B.4.3 The Correctness of the Total Bounding Function

Theorem 3 *The bound() function is a true lower bound on the cost() function.*

Proof: In this proof, we show that the *bound()* function fulfills the two bounding requirements given above, and thus, is a true lower bound on the *cost()* function.

Requirement 1: $bound(G) \leq cost_{min}(G)$.

$$bound(G)$$

$$= \alpha \times area_bound(G) + \beta \times delay_bound(G)$$

$$\leq \alpha \times area_bound_{min}(G) + \beta \times delay_bound_{min}(G)$$

$$\leq cost_{min}(G).$$

q.e.d.

Step 2 of the proof is based on the fact that the *area_bound(G)* is a bound on the area cost for *any* design implementation of G (Theorem 1) and that *delay_bound(G)* is a bound on the delay cost for *any* design implementation of G (Theorem 2). Step 3 utilizes the fact depicted in Figure 15, namely, that any complete solution derived from G will at best have both the smallest area *and* the smallest delay.

Requirement 2: $bound(G1) \geq bound(G2)$ if *G1* is a child of *G2*.

$$bound(G2)$$

$$= \alpha \times area_bound(G2) + \beta \times delay_bound(G2)$$

$$\leq \alpha \times area_bound(G1) + \beta \times delay_bound(G1)$$

$$= bound(G1).$$

q.e.d.

Step 2 of the proof uses the fact that the functions *area_bound()* and *delay_bound()* have been shown to preserve the *monotonicity requirement* in Theorems 1 and 2, respectively.

3 1970 00882 6601

AUG 0 5 1996