A Data Path Synthesis Method for Self-Testable Designs

Christos A. Papachristou, Scott Chiu, and Haidar Harmanani

Department of Computer Engineering
Case Western Reserve University
Cleveland, Ohio 44106

ABSTRACT
A high level synthesis for testability method is presented whose objective is to generate self-testable RTL designs from data flow behavioral descriptions. The approach is formulated as an allocation problem based on an underlying structural testability model and its connection rules. Two allocation techniques have been developed to solve this problem: one based on an efficient heuristic algorithm that generates cost-effective designs, the other based on an integer linear program formulation that generates optimal designs. The allocation algorithms have been implemented and several benchmark examples are presented.

1 Introduction

Advances in VLSI technology have complicated not only the design process but also the testing problem of such chips and systems. To alleviate the testing problem, several design for testability (DFT) techniques were proposed [McCl85, EiWi77, KoMZ79]. However, there are some penalties for using DFT techniques at the system level. These penalties include, among others, hardware overhead, additional design cost for DFT insertion, and a possible circuit performance degradation [KiTH88].

To handle the demands for short turn-around time, a number of high level synthesis systems has been proposed [McFa90]. However, only few researches address the testability problem [Been90, Catt89, AbBr85, GeEl88]. In essence, there are currently two approaches to this problem in high level synthesis: The first one [Been90, Catt89] uses *a macro template* which ensures that DFT is used as a building block for structural level system design. The testing objective is achieved by handling each macro cell separately. The second approach [AbBr85, GeEl88] uses a *testability insertion backend* where a post-synthesis module for the synthesis system is added for converting a register transfer level design into a testable one. A redesign loop will be initiated if the design can not fulfill the testability constraints supplied by the user. Note that none of these techniques support testability at the behavioral level. However, recent advances in high level synthesis made the DFT consideration at higher levels imperative in order to keep pace with the capability of such systems.

This paper presents a new approach to high level synthesis with self testability. The motivation for this work is the need to cover the void between the fields of high-level synthesis and design for testability by avoiding a post-synthesis process which may require costly design modifications. Thus, we aim to bridge the complete separation of the design process from the DFT process by integrating them within the framework of high-level synthesis. Our method is based on a new formulation of the data path allocation problem that maps a scheduled data flow graph (DFG) of the behavior onto a data path which is self testable without the need for post-synthesis modification or hardware insertion. The main features of our method are:

- A formulation of the behavioral synthesis problem to include *structural testability.* This is based on data flow entities which correspond one-to-one to testable blocks.
- Two allocation techniques that map a given data flow behavior into a self testable data path design under technology dependent cost consideration. The first is based on an efficient *graph heuristic algorithm* while the second method is based on an *integer linear program formulation (ILP).*

In section 2 we develop the underlying structural testability model of our synthesis method. In section 3 we discuss the fundamental aspects of testable data flow synthesis. In sections 4 and 5 we present the specific testable allocation techniques, based on a graph-heuristic and on an integer linear program formulation, respectively. The results are discussed in section 6.

2 Structural Testability of Data Paths

We have chosen our test methodology as Built-In Self-Testing (BIST) with random pattern stimulus and multiple input signature analysis. Under this assumption, we base our design approach on the concept of structural testability of data paths which follows from the notion of I-path introduced by [AbBr85]. The main idea of BIST is to reconfigure the data path, during test mode, into a number of self testable *Combinational Logic Blocks* (CLBs). A CLB is testable if, for each CLB port, there exists a circuit path TPGR → CLB → MISR, called I-path, where TPGR is a test pattern generator register and MISR is a multiple input signature register. In the BIST technique, the TPGRs and MISRs are allocated to existing data path registers with the aim of generating an appropriate set of I-paths containing all CLB input ports. Thus, by definition, a data path is *structurally testable* if there exists an I-path for each CLB port of the data path. It may not be always possible to achieve structural testability of a given data path without post-synthesis modification because of insufficient number of registers, or unsuitable register connection structures.

We introduce a specific testable structure or template, called *the Testable Functional Block (TFB),* defined as follows: A TFB consists of three types of components organized in three distinct layers. Input MUXes are in the first

This work is supported in part by the Semiconductor Research Corporation (SRC) under contract 90-DJ-147

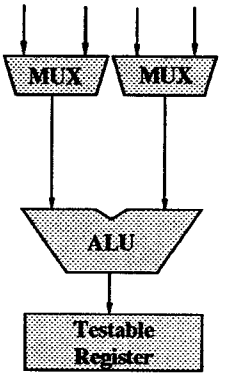

Figure 1: Basic TFB Structure

layer, feeding a CLB in the second layer, whose output goes to Test Registers (TPGR, MISR, or BILBO) in the third layer. Hence, in its simplest form, a TFB consist of an ALU, two MUXes connected to the ALU inputs and an output Test Register, Figure 1. Clearly, each MUX input port is also a TFB input port whereas the TFB output port is the test register output.

The reason for introducing the TFB structure is two fold: 1) it is directly related to the I-path structures and hence it can be used as a basic component of testable data paths; 2) there is a direct relationship between TFBs and behavioral synthesis flow graphs that will be established in the next section.

The connection rules for TFBs to form structurally testable data paths are quite simple: 1) connect each input port of a TFB to a distinct output port of other TFBs or to input registers of the data path; 2) connect the output port of a TFB to the input port of another TFB. The restriction is that there are no *self loops*, Figure 2, within the same TFB I/O port, because it is not possible to form an I-path structure for an input port located in such a self loop without degradation of test data [HuPe87]. This is an important point and it is stated more formally as follows:

- *A data path composed of TFBs that follow the TFB connection rules is structurally testable.*

It should be noted that in our structural testability model we do not consider the the testability of individual data path modules such as adders, and multipliers. However, we assume that the testability parameters of these modules such as test pattern generation length and random fault coverage are known from an extended cell library that includes

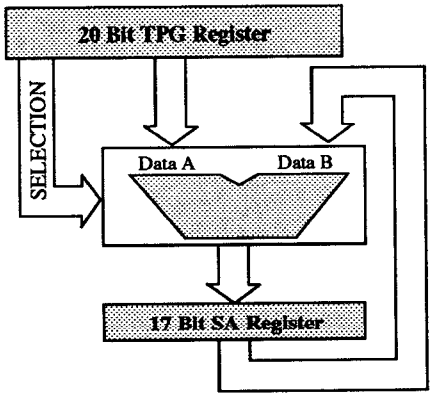

Figure 2: A Self loop

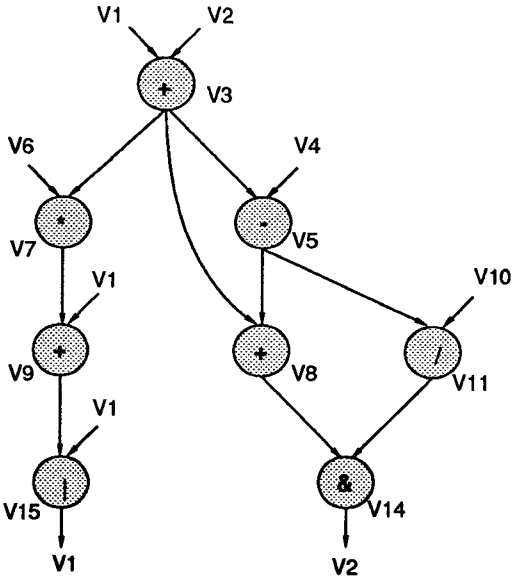

Figure 3: An example DFG [TsSi86]

these parameters. Note, that if the fault coverage is not satisfied, we will have to consider a modification at the logic level; however, this will be taken care of by a logic synthesis system, and is beyond the scope of this paper.

3 Testable Data Flow Synthesis

Our synthesis technique employs a *scheduled* data flow graph (DFG) as input behavior and produces a testable data path while keeping the cost as small as possible. In what follows we describe the fundamental concepts of our approach, in reference to the scheduled DFG of Figure 3. The basic idea of our method is to merge data flow variables, operations and connections variables *simultaneously* using a single merging process. The motivation is as follows. In most of the existing allocation techniques [TsSi86] there is a lack of merging coordination, i.e., merging of variables, operations and connections are performed in separate. This, however, has two disadvantages: First, the allocation leads to irregularly structured data paths that are difficult to test, i.e., needing the insertion of shadow registers for test purposes. Second, the merging order may adversely affect the cost—doing register merging first may increase the overhead of operation merging, and vice versa.

3.1 Formulation

Our proposed allocation merging scheme is based on the concept of actions and actors introduced next. Consider a DFG node associated with the variable instance V, its corresponding operation $O(V)$, and life span $L(V)$. Then the *action*, $A(V)$, of V is the 3-tuple:

$A(V) = [V; O(V); L(V)]$

For the example in Figure 3, the action of V_3 is: [V_3; *Addition*; 1 3]. It follows from this definition that each DFG node represents an action whose output edge(s) is (are) labeled V. The input edges of $A(V)$ are connected to the outputs of other actions.

An *actor* is generated by merging two or more actions. The intuitive rationale is to avoid the dislocation of variable instances from their corresponding operations, keeping them close together during the entire allocation process. This proximity is preserved topologically in the generated data path and it contributes to its structural testability. The specific objective of our scheme is to allow the mapping of actors of the DFG into testable functional blocks,

TFBs, (defined previously) which are the building blocks of the testable data path generated by the proposed allocation.

There are two conditions that should be satisfied for the successful merging of two actions $A(V_1)$ and $A(V_2)$:

1. The life spans $L(V_1)$ and $L(V_2)$ should not overlap.
2. Neither V_1 nor V_2 should be an input edge (variable) to the actions $A(V_2)$ or $A(V_1)$, respectively.

The reason for the first condition is to avoid resource conflicts that occur from merging two conflicting variables into the same register. The condition also prevents the merging of two concurrent operations into the same functional unit. This constraint will result in our basic TFB model, i.e., only one register at the third layer. The reason for the second condition is to avoid self looping in the generated actors which will lead to self looping in the TFBs of the data path produced by the proposed allocation. In fact, the second condition is responsible for satisfying the TFB connection rules, described earlier, and hence preserving the structural testability of the synthesized data path.

The first constraint discussed above could be in fact relaxed by allowing the merger of conflicting variables but not operations. The register conflict will be resolved by allowing more registers at the third layer of our TFB model. Another approach to reduce the effect of the first constraint on the final cost would be by the insertion of dummy nodes into the scheduled DFG. The dummy nodes will be inserted so that to cut the long life spans that may prevent further mergings. Note that the dummy nodes correspond to NO-OPs which do not add any operation merging cost, and preserve the structural testability style. This technique has been used and proved to be very effective in the design examples (section 6).

Formally, two actions that can be merged under the previous conditions are called *compatible*. Compatible actions form an *action sequence* in the order of the control step assignment of their corresponding variable instances. Merging the actions of an action sequence A_1, A_2, generates an *actor* denoted by $A(V_1, V_2)$ or simply A_1A_2, and defined by the 3-tuple:

[*Variable Sequence*; *Operation*; *Life Cycle*]

For example, the *actor* generated by merging the actions A_7 and A_8 in Figure 3 is:

A_7A_8 = [V_7 V_8; ***multiplication, addition***; 2 3]

3.2 General Optimization

The allocation merging scheme guarantees structural testability by mapping actors onto TFBs. In fact, an initial testable data path structure can be immediately generated by direct mapping of the DFG actions into TFBs. To reduce the cost of the synthesized data path it is desirable to find a minimal number of actors of the given DFG. This leads to testable data path synthesis under technology-independent cost. A more realistic approach is to synthesize the data path under technology-dependent cost that includes among other parameters area, delay, and testability overhead. There is a large number of actors that can be generated but fortunately we only need a much smaller number since there exists a covering relationship among these actors, defined as follows:

1. An actor is covered by, or is a *subactor* of, another actor B, if the action sequence comprising B is a subsequence of that of A.
2. A *prime* actor in a DFG is *not* a subactor of any other actor.

For example, in Figure 3 A_3A_{11} is a subactor of $A_3A_{11}A_{15}$ which is a prime actor.

The optimization strategy of our allocation scheme can be formulated in a manner similar to the classical covering problem, as follows:

1. find all prime actors in a scheduled DFG.
2. find a minimal or near-minimal collection of prime actors covering all actions in the DFG.

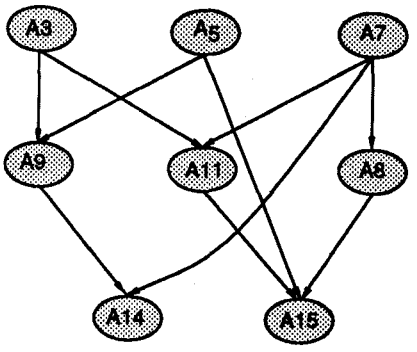

Figure 4: Module allocation graph for the Facet DFG

The finding of all prime actors is facilitated by the notion of the *leader*, defined as the first-in-order action of a prime actor. Thus, A_3 is the leader of $A_3A_{11}A_{15}$. The following statement is true:

- **Statement:** *The leaders of prime actors are mutually incompatible.*

The importance of the above statement is that it allows us to significantly reduce the computation of prime actors by finding first their leaders via the following two-step process.

- The top level actions of a scheduled DFG are leaders.
- Those actions that are mutually incompatible to the top level actions, and are mutually incompatible themselves, are also leaders.

The finding of all prime actors is performed by a multiple-tree data structure, labeled *Module Allocation Graph*, such that the leader actions are placed in the graph roots and the rest of actions are hierarchically placed in the graph nodes preserving the schedule of the DFG. Thus every graph path represents a prime actor. The module allocation graph for the example DFG of Figure 3 is illustrated in Figure 4.

The prime actors for our running example are listed under their leaders below:

$A_3A_9A_{14}$	$A_5A_9A_{14}$	$A_7A_{11}A_{15}$
$A_3A_{11}A_{15}$	A_5A_{15}	$A_7A_8A_{15}$
		A_7A_{14}

The next step of the optimization process is to find a minimal set of prime actors covering the actions of the scheduled DFG. This problem can be formulated and solved as a covering table problem. In general, the covering table problem is NP-complete but there are several efficient heuristic solutions [Chri75]. In our case, we have the extra advantage that the covering table rows, i.e., the prime actors, are conveniently grouped in terms of their leaders, and this observation significantly accelerates the finding of a solution. The covering table (Table 1) for our running example (Figure 3) has the following minimal solution:

A_3A_{11}; $A_5A_9A_{14}$; $A_7A_8A_{15}$

	A_3	A_5	A_7	A_8	A_9	A_{11}	A_{14}	A_{15}
$A_3A_9A_{14}$	1				1		1	
$A_3A_{11}A_{15}$	1					1		1
$A_7A_{11}A_{15}$			1			1		1
$A_7A_8A_{15}$			1	1				1
A_7A_{14}			1				1	
$A_5A_9A_{14}$		1			1		1	
A_5A_{15}		1						1

Table 1: Cover table for the Facet example

3.3 Cost Considerations

Because of technology-dependent design constraints, it is more realistic to optimize the total actor area cost required during the allocation. The area cost of each actor is approximated as follows:

$$C(A) = C(R_A) + C(O_A) + C(TOV_A) + C(MUX_A)^{\dagger}$$

where:
$C(A)$ is the area cost of actor A.
$C(R_A)$ is the area cost of R_A, the register of actor A.
$C(O_A)$ is the area cost of O_A, the operation set of A.
$C(TOV_A)$ is the testability overhead of actor A.
$C(MUX_A)$ is cost of the multiplexers of actor A.

The testability overhead cost increases the cost of R_A by $C(TOV_A)$ and the exact amount depends on whether R_A is configured as a test pattern generator register (TPGR), multiple input signature register (MISR), or BILBO as well as the bit-length and the original design or implementation of the register. The operation set cost $C(O_A)$ varies depending on the complexity of the actor operation set. For example,

Cost$\{addition, division\}$ > Cost $\{addition, subtraction\}$

It is possible to derive first-cut estimates of the above cost factors within a certain technology and logic-level design libraries. There is an additional cost for an actor A to account for multiplexer cost, $C(MUX_A)$, which is due to the two multiplexers in the TFB implementation of A.

For the purpose of this paper, we have derived first-cut estimates of the above cost factors based on a commercial library. In the next sections, we propose two methods to solve this problem. One is based on a graph-oriented heuristic (section 4); the other method is based on an integer linear program formulation (section 5).

4 Allocation by a Graph-Heuristic Algorithm

Each time two nodes are merged, the total number of TFBs in the data path is reduced by one. Reducing the number of TFBs by itself is not sufficient to minimize the data path cost. The module allocation graph described earlier (Figure 4) contains all the information about the actors and prime actors of the DFG. We have developed an incremental algorithm for merging the actors on the module allocation graph which takes into account the reduction in cost when making a decision concerning which nodes (actors) are to be merged.

4.1 Gain and Loss in Actor Merging

Each merge between two actors A and B corresponds to a move in the design space from point S_1 to S_2, where S_1 and S_2 are the states of the data path before and after the merge. The gain $G(A, B)$ is defined as the difference between the cost of the two states: $G(A, B) = Cost(S_1) - Cost(S_2)$. More specifically, the gain $G(A, B)$ here measures the reduction in the data path cost as a result from merging node A with node B. Taking into consideration the previous cost formulation we have:

$$G(A, B) = C(A) + C(B) - C(A, B)$$

where $C(A, B)$ is the cost of the merged actors AB and $C(A)$, $C(B)$, are the costs of A and B, respectively.

When two nodes A and B are merged together, some edges are removed from the module allocation graph as a result. The reason is due to the fact that the compatibility relation is not transitive. Consider the module allocation graph of Figure 4 for example. If we merge compatible nodes A_3 and A_9 then the edge (A_5A_9) should be removed from the module allocation graph because A_5 is not compatible with A_3, although it is compatible to A_9. To preserve compatibility, we also need to generate new edges in the module allocation graph as a result of merging. For example, we need to generate the edge (A_5A_{14}) if we merge A_3 and A_9. Thus, any node merging in the module allocation graph requires edge adjustments according to the compatibility relations.

The *loss L(A, B)* is the sum of the gains of all edges that were removed as a result of merging A and B, except the gain of the edge (A, B). The loss can be calculated as follows:

$$L(A, B) = G_{out} + G_{in} - 2 * G(A, B)$$

where:
G_{out} = the sum of Gains of all edges going out of A.
G_{in} = the sum of Gains of all edges going into B.

The loss gives an indication of how the current merge will affect future merges and it is more global than the gain concept because it takes into consideration the neighboring nodes. We remark that each edge in the module allocation graph can be associated with both a gain and a loss value.

Our merging strategy is as follows. We merge nodes level by level, starting from the top level in the module allocation graph. Of course only compatible nodes, i.e., those connected by a directed edge are mergeable. By merging levels i and $i+1$ we mean that we merge nodes of level i with nodes at level $i+1$ until no edges are left between the two levels. The merging is performed on the basis of some well formed heuristics, described shortly, and the gain and loss values. In every successful merging of nodes A_i and B_{i+1} at levels i and $i+1$, respectively, we replace these nodes by a new node (actor) AB_{i+1} at level $i+1$. As explained earlier, this requires replacement of a number of edges in the module allocation graph to account for the merging, and it also requires computation of the new gain and loss values. In what follows we provide some definitions to help the explanation of the heuristics and allocation algorithm.

Let $Out(i, i+1)$ denote the number of nodes in level i which have outgoing edges to nodes in level $i+1$.

Let $In(i, i+1)$ denote the number of nodes of level $i+1$ which have incoming edges from nodes at level i.

- **Statement** — *By merging levels i and i+1, the maximum number of merges is upper bounded by n, where* $n = min\{Out(i, i+1); In(i, i+1)\}$

4.2 Heuristics

Each positive gain we obtain from the module allocation graph has the effect of reducing the data path cost by the amount of that gain. By maximizing the amount of gain we obtain from the module allocation graph, we minimize the data path cost. Our strategy in maximizing the gain can be summarized as follows: At each stage, we select a pair of nodes that, when merged, will result in a minimal negative effect on future merges. We believe that this strategy works better than a greedy approach that tries to get as much gain as possible at each stage without accounting for future merges. There are two negative effects that are likely to occur:

1. One or more edges are removed, and the gain associated with those edges can no longer be obtained by the future merges.
2. The number of possible future merges is reduced as a result of removing some edges, and thus removing node compatibilities.

We discuss next three heuristics that will guide our algorithm in selecting the pair of nodes that, when merged, will have as small negative effect as possible on future merges. More specifically:

- **Heuristic 1:** To choose the priority level, when merging two levels i, and $i+1$:
  - $Out(i, i+1) < In(i, i+1)$: merging priority is given to the nodes at level i.

[†]Note that routing cost is not included in the cost formulation at the present time. However, such routing costs based on routing estimators will be included in the future.

- $Out(i, i+1) > In(i, i+1)$: merging priority is given to the nodes at level $i+1$.
- $Out(i, i+1) = In(i, i+1)$: we give the nodes in both levels the same priority.

This heuristic is basically used to determine if merging the most restricted nodes in one level, will have an effect on increasing the number of future merges.

- **Heuristic 2:** Choose the merging order for the nodes in the priority level in descending order of their restrictions. Restriction of a node A is measured by:
  - the number of outgoing edges if A is in level i.
  - the number of incoming edges if A is in level $i+1$.

  Clearly, the *most restricted node* is the one with the minimum number of incoming or outgoing edges. Nodes that are not in the priority level are considered to be least restricted.

  This heuristic aims to reduce the first negative effect. Restricted nodes do not have the flexibility of merging with many other nodes. Thus, by merging those nodes first, we increase the possibility of maximizing the number of future merges.
- **Heuristic 3:** Choose the edge to be merged, as follows. Out of the edges that involve one or more of the most restricted nodes, in both levels, choose the edges with a minimum loss. If there are ties, then among the edges with equal loss choose the edge with maximum gain (further ties are broken arbitrarily).

  This heuristic aims to reduce the second negative effect, as edges with minimal loss are merged first.

The graph-heuristic algorithm has a complexity of $O(N^2)$, making this allocation method attractive to be embedded in fast design exploration tools.

5 Allocation by an Integer Linear Program Technique

High-level synthesis problems are often formulated as mathematical programming problems since they attempt to optimize an objective function that includes among other parameters: area, time, and speed. Although linear programming methods may be time-consuming, they take a more global approach than heuristic techniques and tend to produce more optimal results. Hafer [HaPa83] used mixed integer linear programming to select ALU's; Marwedel [Marw86] used integer programming in the Mimola system in the context of module selection; ALPS [LeHL89] used integer programming driven scheduling, and ADPS [PaKo90] used linear programming in the context of scheduling.

In this section, we present an Integer Linear Programming (ILP) formulation that searches for an optimum solution to our allocation problem, formulated in section 3. The proposed formulation follows directly from the cover table illustrated in Table 1, section 3.

5.1 General Formulation

To obtain optimal-cost solution we need to consider in the ILP not only the prime actors but other actors as well. However, we do not need to include all possible actors because of the following reason.

- **Simplification rule:** *Consider two actors A and B. If A covers B and C(B) ≥ C(A) then we need not consider B in the ILP formulation.*

Thus, we eliminate all actors which are covered by some other actor, but have the same cost. This simplification rule turned out to be very efficient experimentally, reducing the size of the problem between 50% and 70%. This was especially true in the case of digital filters.

Let F^* be the set of all feasible actors obtained by using the simplification rule. We need to generate a subset selection from F^* covering all actions and resulting in a minimal total actor cost.

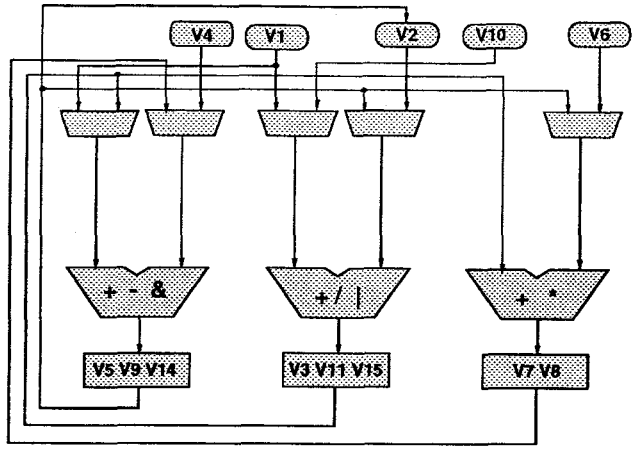

Figure 5: Data path generated from the Facet example

Let N denote the total number of actions (DFG nodes), and M be the cardinality of F^*, and let C_m be the cost associated with an actor m. The cover matrix of F^*, A, is a $M \times N$ (0,1) matrix (Table 1). Below is the ILP formulation:

Constraints:

$$\sum_{i=1}^{M} A_{actor_i, action_j} * X_i \geq 1 \qquad j = 1, 2, ...N \qquad (1)$$

$$X_i = 0 \ or \ 1 \qquad (2)$$

Where

$$A_{actor_i, action_j} = \begin{cases} 1 & \text{if } actor_i \cap action_j \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Cost function to minimize

$$C = \sum_{m=1}^{M} X_m * C_m \qquad (3)$$

Equation (1) ensures that every action in the original DFG is covered, while equation (2) ensures that the selection vector is an integer (0,1) vector. The objective function, equation (3), minimizes the cost factors discussed in section 3.3.

For computational purposes, we note that the optimal solution of our problem must be a vertex solution in the same sense as for linear programming. This makes the ILP formulation fairly easy to solve using the branch and bound method by moving from the continuous optimum to the integer optimum in comparatively few steps [GaNe72].

6 Results

We implemented the proposed allocation techniques, both the graph heuristic and the integer program. The implementations are written in C language, and run on a Sun 4/260 in a Unix environment. The system takes a behavioral description, schedules it using ASAP, ALAP, or forced-directed scheduling (FDS), and then generates the module allocation graph. The system will then run either of two options, ILP or graph heuristic. Next, we discuss four examples from the current literature, for comparison purposes. However, direct comparisons are not possible to make because of the *structural testability* of our design requirements. Our comparisons are based on total area cost that includes the cost factors discussed in section 3.3. For every example, we show the overhead that our testability design may introduce (with respect to other design examples cost); however, this overhead illustrates improvement in area cost when it is negative. It is

System	ALUs	# Reg	# Mux	# Mux In	Cost	Overhead
Ours	(-&+)(*+)(+/|)	8	5	10	915	–
HAL	(/)(-&+)(*|+)	7	6	13	895	2.10 %
Facet	(/)(-&+)(*|+)	8	7	15	950	-3.8 %
Splicer	(/)(-&+) (*|+)	7	4	8	810	11.47 %

Table 2: Results from the Facet example

System	ALUs	# Reg	# Mux	# Mux In	Cost	Overhead
Ours	(-*)(+*)(-*>)	10	6	15	1130	–
HAL	(*)(*)(+)(-)(>)	11	6	13	1125	0.44 %
Splicer	(*)(*)(+)(-)(>)	10	5	11	1040	7.96 %

Table 3: Results from the HAL Differential Equation example

CStep	System	ALUs	# Reg	# Mux	# Mux In	Cost	Overhead
8	Ours	(*), (*), (*) (+), (-), (+-)	6	10	27	1060	–
8	ADPS	(*),(*), (*) (+), (-), (+-)	12	8	29	1345	-26.88 %
9	Ours	(*), (*), (-) (+), (+-)	6	10	24	935	–
9	ADPS	(*),(*) (+-), (+-)	11	6	29	1185	-26.73 %

Table 4: Results from the bandpass filter

System	ALUs	# Reg	# Mux In	Cost	Overhead
Ours	(+), (+), (+), (+) (+), (+*), (+*), (*)	9	32	1460	–
FDS	(*),(*), (*) (+), (+), (+)	12	31	1345	7.19 %
Khailath	(*), (*), (*), (*) (+), (+), (+), (+)	12	32	1490	-2.05 %
FDLS	(*),(*), (*) (+), (+)	12	31	1305	10.61 %
ADPS	(*),(*), (*) (+), (+), (+)	12	32	1490	-2.05 %

Table 5: Results from the wave filter

encouraging to notice that the test overhead did not exceed 11 %. Some of the design examples are:

• **The Facet example:** This is one of the earliest examples, and was first introduced by [TsSi86]. Our system generated the testable data path shown in Figure 5. We compared our results with Facet [TsSi86], HAL [PaKn87], and Splicer [Pang88]. Results comparisons are shown in Table 2.

• **The Differential Equation example from HAL:** This example solves a second order differential equation, and was first introduced by [PaKn87]. We compared our final data path with HAL and Splicer [Pang88]. Results are in Table 3.

• **Sixth-Order Elliptic Bandpass Filter:** This example was first introduced by [TuRa86]. We have assumed that multiplication coefficients are stored in a RAM or ROM. We compared our designs with ADPS, Table 4.

• **Fifth-Order Wave Digital Elliptic Filter:** This example was first introduced by [KuWK85], and popularized later by [PaKn87]. Again, multiplication coefficients are assumed to be stored in a ROM. We compared our results, for control step 17, to ADPS [PaKo90], [PaKn87], and [KuWK85]. Results comparisons are shown in Table 5.

References

[AbBr85] M. Abadir, M.A. Breuer, "A Knowledge-Based System for Designing Testable VLSI Chips," *IEEE Design & Test,* August 1985, pp. 56-68.

[Been90] F. Beenker, R. Dekker, R. Stans, M. Van Der Star, "Implementing Macro Test in Silicon Compiler Design," *IEEE Design & Test,* April 1990, pp. 41-51.

[Chri75] N. Christofides, *Graph Theory: An Algorithmic Approach,* Academic Press, New York, 1975.

[Catt89] F. Catthoor, J. Van Sas, L. Inze, H. De Man, "A Testing Strategy for Multiprocessor Architecture," *IEEE Design & Test,* April 1989, pp. 18-34.

[DeNe89] S. Devadas, R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. on CAD,* July 1989, pp. 768-781.

[EiWi77] E.B. Eichelberger, T.W. Williams, "A Logic Design Structure for LSI Testing," *Proc. 14th Design Automation Conference,* June 1977, pp. 462-468.

[Gajs88] D. Gajski, *Silicon Compilation,* Addisson-Wesley, 1988.

[GaNe72] R.S. Garfinkel, G.L. Nemhauser, *Integer Programming,* New York, John Wiley & Son, 1972.

[GeEl88] C. Gebotys, M. Elmasri, "VLSI Design Synthesis with Testability," *Proc. 25th Design Automation Conference,* June 1988, pp. 16-21.

[HaPa83] L. Hafer, A.C. Parker, "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic," *IEEE Trans. on CAD,* 1(1983), pp. 4-18.

[HuPe87] C.L. Hudson, G.D. Peterson, "Parallel Self-Test With Pseudo-Random Test Patterns," *Proc. International Test Conference,* Sept. 1987, pp. 954-971.

[KiTH88] K. Kim, J.G. Tront and D.S. Ha, "Automatic insertion of BIST hardware using VHDL," *Proc. 25th Design Automation Conference,* June 1988, pp. 9-15.

[KoMZ79] B. Koenemann, J. Mucha and G. Zwiehoff, "Built-In Logic Block Observation Techniques," *Proc. International Test Conference,* October 1979, pp. 37-41.

[KrAl85] A. Krasniewski, A. Albicki, "Automatic Design of Exhaustively Self Testing Chips with BILBO Modules," Proceedings of the International Test Conference, September 1985, pp. 362-371.

[KuWK85] S.Y. Kung, H.J. Whitehouse, T. Khailath, *VLSI and Modern Signal Processing,* Prentice Hall, 1985, pp. 258-264.

[LeHL89] J. Lee, Y. Hsu, Y. Lin, "A New Integer Linear Programming Formulation For the Scheduling Problem in Data Path Synthesis," *Proc. International Conference on Computer Aided Design,* 1989, pp. 20-23

[Marw86] P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Software System," *Proc. 23rd Design Automation Conference,* June 1986, pp. 271-277.

[McCl85] E.J. McCluskey, "Built-In Self-Test Techniques," *IEEE Design & Test,* April 1985, pp. 21-28.

[McFa90] M. McFarland, A. Parker, and R. Compasano. "The High Level Synthesis of Digital Systems," *Proc. of the IEEE,* Vol. 78, No. 2, February 1990, pp. 301-318.

[Pang88] B.M. Pangrle, "Splicer: A Heuristic Approach to Connectivity Binding," *Proc. 25th Design Automation Conference,* June 1988, pp. 536-541.

[PaPM86] A.C. Parker, J. Pizarro, M. Mlinar, "MAHA: A Program for Data Path Synthesis," *Proc. 23rd Design Automation Conference,* June 1986, pp. 461-466.

[PaKo90] C. Papachristou, H. Konuk, "A Linear Program Driven Scheduling and Allocation Method Followed by an Interconnect Optimization Algorithm," *Proc. 27th Design Automation Conference,* June 1990, pp. 77-83.

[PaKn87] P. Paulin, J.P. Knight, "Forced-Directed Scheduling in Automatic Data Path Synthesis," *Proc. 24th Design Automation Conference,* June 1987, pp. 195-202.

[TsSi86] C. Tseng, D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* V. CAD-5, No. 3, pp. 379-395, July 1986.

[TuRa86] L.E. Turner, B.K. Ramesh, "Low Sensitivity Digital Ladder Filters with Elliptic Magnitude Response," *IEEE Trans. on Circuits and Systems,* July 1986, pp. 697-706.