RC 20352 (01/23/96)

COMPUTER SCIENCE

IBM Research Report

Algorithms for Incremental Synthesis

Daniel Brand
IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Anthony D. Drumm
IBM AS/400 Division
Rochester, Minnesota

Sandip Kundu
IBM Research Division
Austin, Texas

Prakash Narain
AMD
Sunnyvale, California

IBM Research Division
Almaden . T.J.Watson . Tokyo . Zurich

Algorithms for Incremental Synthesis

Daniel Brand	Anthony D. Drumm	Sandip Kundu	Prakash Narain
IBM Research Division Yorktown Heights, NY	IBM AS/400 Division Rochester, MN	IBM Research Division Austin, TX	AMD Sunnyvale, CA

Abstract:

A small change in the input to logic synthesis may cause a large change in the output implementation. This is undesirable if a designer has some investment in the old implementation and does not want it perturbed more than necessary. We describe a method that solves this problem by reusing gates from the old implementation, and restricting synthesis to the modified portions only.

1. Introduction

Even when an implementation of a design is generated automatically, it is common for a designer to have an investment in the implementation. Examples of such investment are effort to synthesize, expense of physical design, mask generation, any manual changes, or simply designer's time spent understanding the implementation. This investment may be jeopardized if the designer has to modify the specification. Such modifications, commonly called "engineering changes" or ECs, are necessitated by changes in requirements, errors, or efforts to speed up the logic. If the designer synthesizes his new specification then he may get a completely different implementation, because synthesis tries to find a minimal representation of the new function, and a small change in the specification of a function may cause a large change in its optimal implementation. Moreover, synthesis systems contain heuristics, which make some "arbitrary" choices, which further contribute to large changes in the implementation. For this reason it is a common practice to "freeze" a design, meaning that after a design is sufficiently stable, synthesis is no longer used, and any modifications are done manually in both the specification and the implementation.

This is undesirable for three reasons. First, it is very time consuming for a designer to understand which part of the synthesized implementation corresponds to the modified part of his specification. Secondly, it is very easy for the designers to make a mistake in modifying the implementation. Thirdly, the modified implementation may be wrong even if the first two tasks were performed correctly. The reason is that some optimization performed on the old version might be invalid for the new function.

To illustrate the last point consider the circuit in Figure 1. It is the result of expressing a given design as an initial unoptimized network. The rectangle represents some block of combinational logic. The initial network is then optimized into the one shown in Figure 2. Note that one of the connections of the rectangle has been replaced by the constant 0, and the block has been optimized and technology mapped as indicated by the shading. For the sake of this example assume that the inverter and the nand gate remain intact during optimization. Then the designer performs the EC indicated in Figure 3. It involves changing the inverter into a non-inverting buffer. In order to generate a new implementation manually the designer changes the inverter in the old implementation of Figure 2 into a non-inverting buffer creating the new implementation of Figure 4. He did not make any mistake in relating his source to the implementation, but this new implementation is functionally wrong, because introduction of the constant 0 in the old version was correct only in the presence of the inverter. This example shows that a small change in specification may require a larger change in implementation than normally expected by designers.

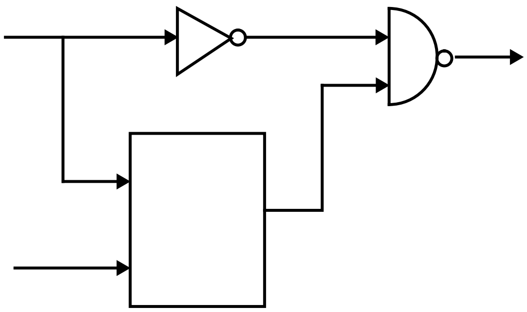

Figure 1: Old specification

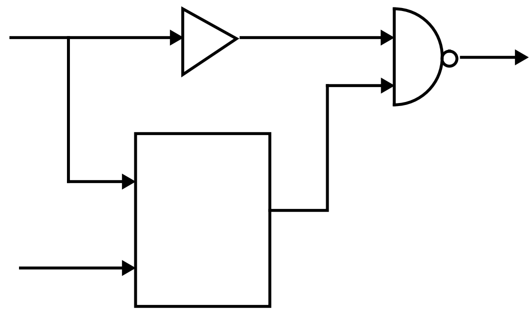

Figure 3: New specification

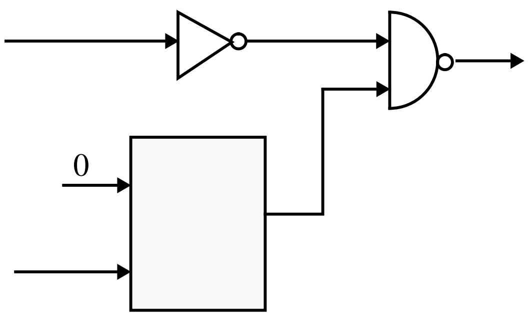

Figure 2: Old implementation

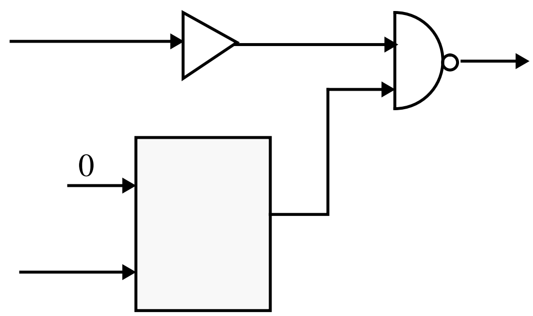

Figure 4: New incorrect implementation

Therefore, several approaches have been proposed for automating the EC process. These approaches differ by their objectives. Some methods restrict themselves to a specific set of ECs [9],[15],[17]. Others have the goal of preserving the final layout, and allowing modifications only around the perimeter of the implementation [11],[19]. There are several methods with the same goal as ours, namely, reusing as many gates from the old implementation as possible. An example is [18], whose method is applicable if the logic structure before and after synthesis is the same, except for technology mapping. The method closest to ours is that of [10], which can handle arbitrary structural differences. The main distinguishing feature of our method is a preprocessing step calculating correspondence between the new specification and the old implementation, which then gives us not only efficiency, but allows greater reuse of the old implementation.

2. Methodology

Regular (i.e., non-incremental) synthesis has the following steps, which are identical for all versions of a design:

```
READ(Specification)
REGULAR_SYNTHESIS
WRITE(Implementation)
```

In case of incremental synthesis we have an old and a new version of a design. The old version is given by Specification0 (e.g., Figure 5) and Implementation0 (e.g., Figure 6), which could have been produced automatically or manually. The new version is given by Specification1 (e.g., Figure 7). We will assume all three to be given as gate networks, which is the normal representation for logic synthesis. Correspondence between primary IOs and other nets is indicated by the common letter forming the name. (This commonality of names is used for exposition only; our programs do not rely on names.) Throughout the paper we will use shading of gates to indicate that they have already been synthesized and mapped into a technology.

If regular synthesis is applied to Specification1 it is likely to produce Implementation1 of Figure 8. While this might be optimal from some points of view, a designer might prefer the implementation of Figure 9. Although that implementation might take larger area than that of Figure 8, it preserves the designer's investment he put into Implementation0 of Figure 6. In particular, designers do not like to see changes in the implementation of outputs unaffected by the EC, e.g., output Z0.

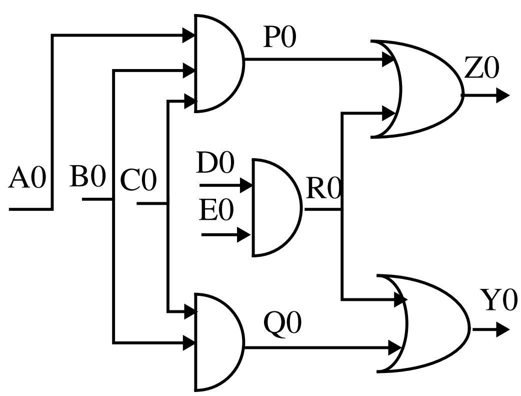

Figure 5: Specification0

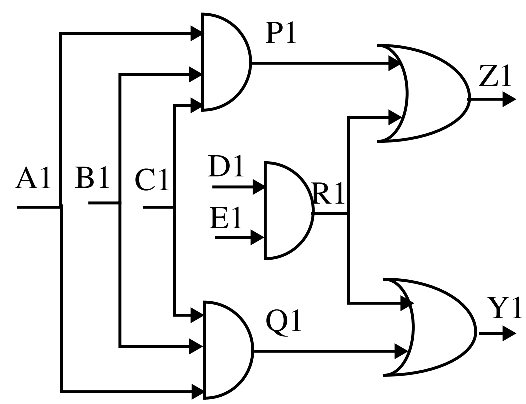

Figure 7: Specification1

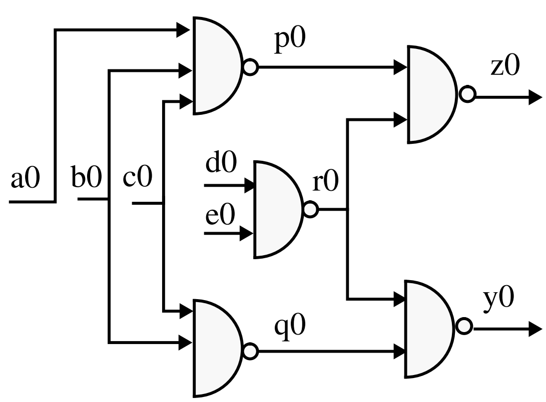

Figure 6: Implementation0

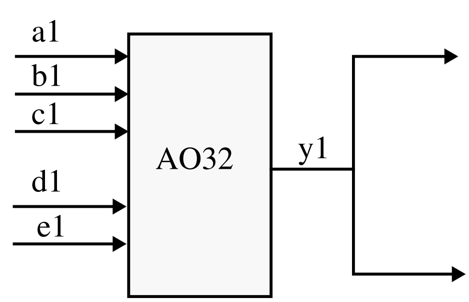

Figure 8: Implementation by regular synthesis

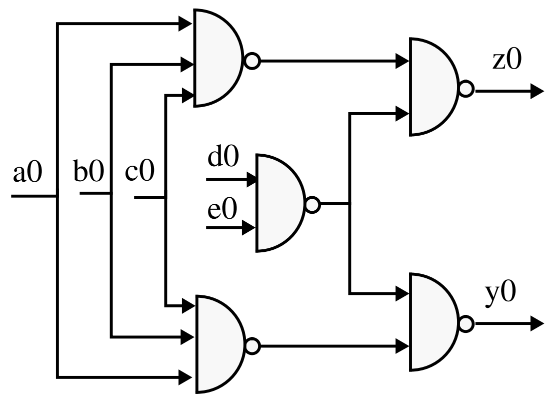

Figure 9: Implementation1 by incremental synthesis

In our approach to incremental synthesis we use the following steps:

FUNCTIONAL_CORRESPONDENCE (Specification0 , Implementation0)
READ (Specification1)
STRUCTURAL_CORRESPONDENCE (Specification0 , Specification1)
LOGIC_REUSE (Implementation0 , Specification1)
REGULAR_SYNTHESIS
POST_PROCESSING
WRITE(Implementation1)

The main subject of this paper are the steps FUNCTIONAL_CORRESPONDENCE (Section 3), STRUCTURAL_CORRESPONDENCE (Section 4), and LOGIC_REUSE (Section 5). Their result is a preliminary implementation of Specification1 (see Figure 10). This preliminary implementation has three kinds of gates -- some gates from Implementation0 (the shaded NAND gates), some gates from Specification1 (the AND gate) and some inverters needed to connect the first two kinds of gates. We will refer to the gates from Implementation0 (shaded in our figures) as “old gates” and all the others as “new gates”. The objective of incremental synthesis is to minimize the number of new gates. In particular, primary outputs that retain their specification should also retain their implementation.

Once this preliminary implementation is formed it is run through regular synthesis (including optimization, technology mapping, etc.) in order to process the new gates. During this regular synthesis all the old gates are marked as protected from any synthesis transformations, and remain unchanged.

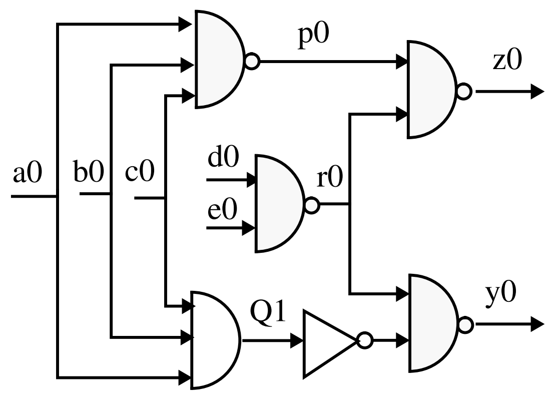

Figure 10: Result of LOGIC_REUSE

Our goal of retaining as much of the old implementation as possible may result in a new implementation that is suboptimal in terms of area, delay, etc. This may or may not be acceptable depending on how much investment already exists in the old implementations. In earlier stages of the design process (for example, before placement) it may be desirable to sacrifice some similarity between the new and old implementation for the sake of a better implementation. In that case we use a post-processing step, where the protection is removed from the old gates for several reasons: A primary output, which retains its old implementation, may nevertheless have different delay because of different loading of some nets. Also some of its faults might become untestable. The post-processing step may be needed to adjust power levels and improve testability.

Our general approach to incremental synthesis tries to identify pieces of logic in Specification1 (e.g., the two OR gates) that can be replaced by pieces of logic from Implementation0 without altering the function of Specification1. This is, in general, a difficult problem, because given a piece of logic in Specification1, it is not clear what would be good candidate logic in Implementation0 for the replacement. It is not clear because there is in general no relationship between Specification1 and Implementation0. First of all, they are functionally different; they may even have different primary IOs. Secondly, they are structurally different because synthesis tends to make drastic logic restructuring.

In order to derive a correspondence between Specification1 and Implementation0 (see Figure 11), we use Specification0, which is functionally related to Implementation0 (both should implement the same function), and it is also structurally related to Specification1, (in case of only an incremental change to the specification, the two specifications "look" similar). The correspondence between Specification 1 and Implementation0 is then simply obtained by transitivity.

Incremental synthesis needs a "boolean reasoning mechanism" as much as regular synthesis. There are several such mechanisms, for example, truth tables[16], PLAs[6], BDDs[8]. We have chosen to do boolean reasoning using a test generator [13][14] for two reasons. A test generator operates directly on the given gate network, while other mechanisms need to build a special representation, which may be of exponential size. Furthermore a test generator uses both the function and the structure of a network; thus we may be able to manipulate the structure so that the running time does not become excessive even for large functions (see Section 3 and Section 5).

Besides the need to reason about functionality, we also need to discover where the ECs are, which is a

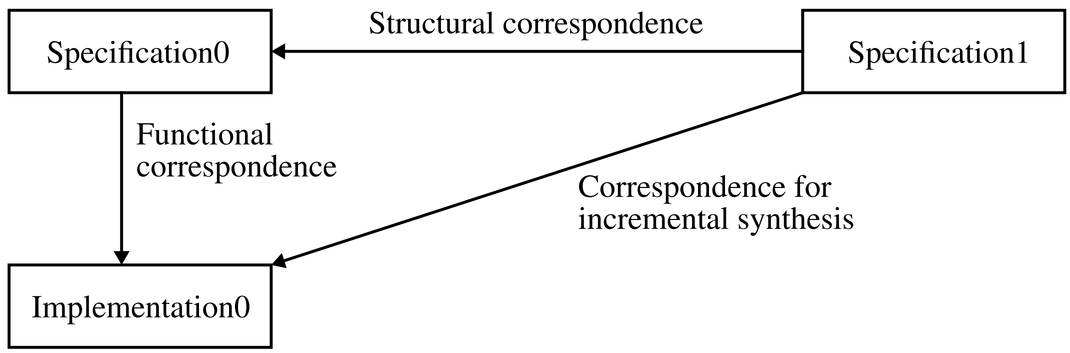

Figure 11: Correspondence composition

purely structural property (see Section 4). After we explain the algorithms we will report experimental results on several examples (Section 6).

3. Functional correspondence

Functional correspondence is calculated between two logic networks F and G, which are Specification0 and Implementation0. The objective of the algorithm is to find for each net f in F a net g in G, so that the function of f can be replaced by the function of g without changing the functionality of F. In the example of Figure 5 and Figure 6 the correspondence is indicated by having the same name, except that nets in Implementation0 are in small type face.The two networks are not treated symmetrically. When this algorithm is used for verification [4], it does not matter which is designated F and which is G. On the other hand, for incremental synthesis we always use F = Specification0, G = Implementation0 because the direction of the resulting correspondence is significant. Also a heuristic described later is based on the assumption that F = Specification0, G = Implementation0. Other than that the algorithm of this section is independent of what F and G represent.

As an example, consider F to be the network of Figure 5 and G to be the network of Figure 6. Then the algorithm should discover that the inverse of the function of p0 can replace the function of P0. Also it should discover that the function of z0 can replace Z0. Similarly r0 can replace R0 and q0 can replace Q0 with negative polarity, while y0 can replace Y0 with positive polarity. Any of these replacements can be performed without affecting the functionality of F.

Before the algorithm starts we need a correspondence between primary IOs of F and G; such correspondence can be usually based on the primary IO names. We also need a correspondence between latches because the algorithm is applicable to combinational logic only. Latch correspondence is a non-trivial problem, but we will not discuss our solution because it is applicable only to our synthesis system[5]. Having found the latch correspondence, we will discuss combinational logic only; in this section the term *primary output* will include latch inputs and *primary input* will include latch outputs.

The approach is based on the following Lemma.

Lemma 1:

Let x be a vector of variables and y be a single variable. Consider arbitrary function f(x), g(x), F(x, y). (Since all three functions depend on x we will drop x from our notation; for example by F(f) we will mean F(x, f(x)).) Then the following two identities hold for all x.

$$F(f) = F(g) \quad \text{iff} \quad F(f \oplus g) = F(0) \tag{1}$$

$$F(f) = F(\bar{g}) \quad \text{iff} \quad F(f \oplus g) = F(1) \tag{2}$$

Proof:

We will be done if we show that (1) and (2) are true for any input pattern x. For a given input x there are four possible combinations of values for f and g. For illustration we will show the case f = 0 and g = 1; all the other cases are similar. Substituting f = 0, g = 1

in (1) we get F(0) = F(1) iff F(1) = F(0)

in (2) we get F(0) = F(0) iff F(1) = F(1)

❑

The lemma tells us how to use a test generator to check whether g can replace f inside F. The right hand side of (1) says that the output of the XOR is not testable for stuck at 0 and the left hand side says that in this case we may replace f by g. Similarly (2) says that if the XOR is not testable for stuck at 1 then we can replace f by $\bar{g}$.

Figure 12 shows two gate networks F and G sharing the same input vector x, and having subfunctions f

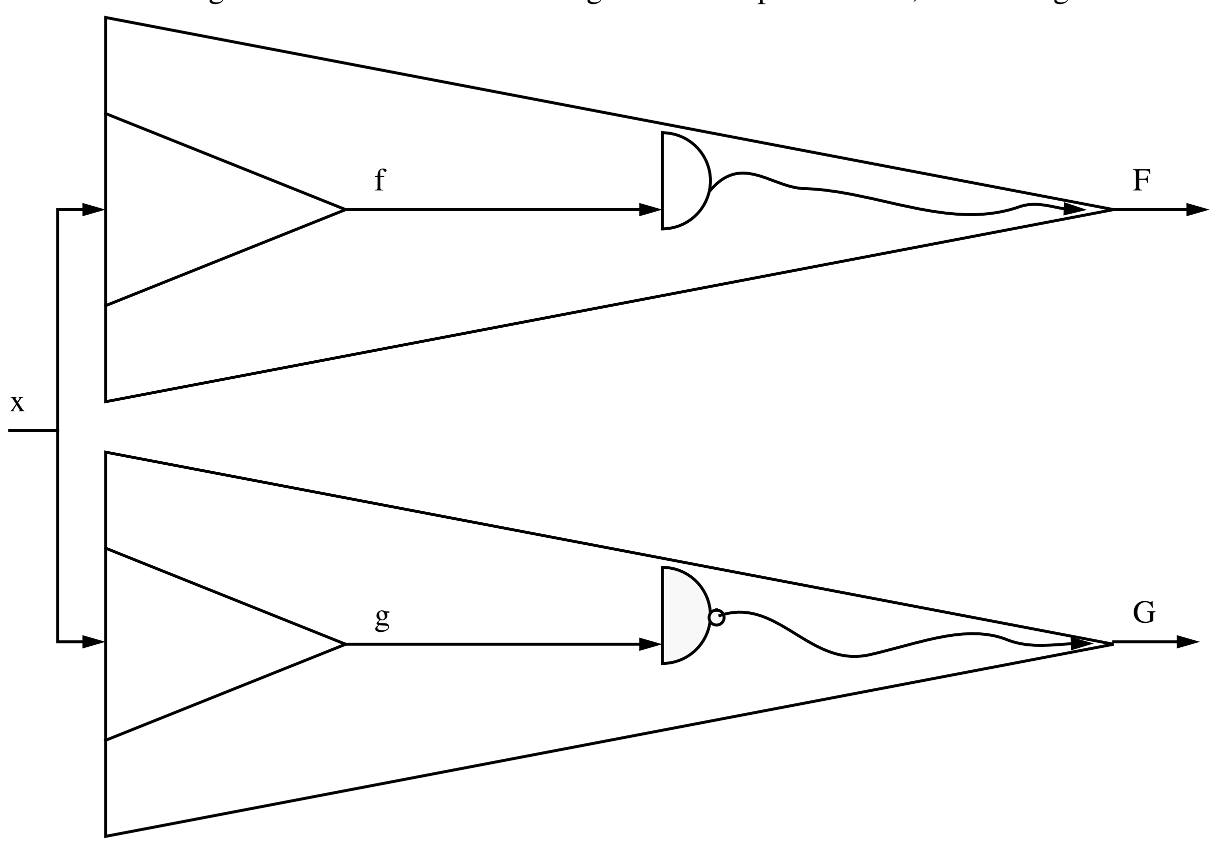

Figure 12. f is an internal function of F

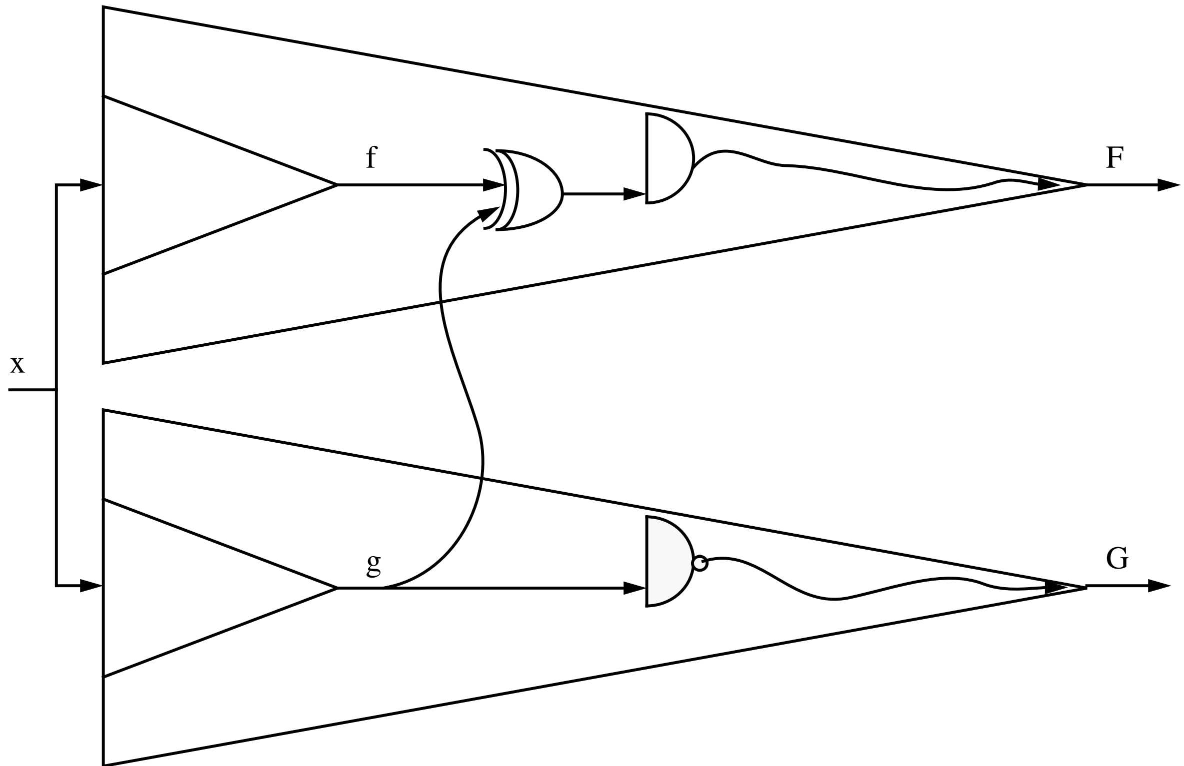

Figure 13. Can g replace f inside F?

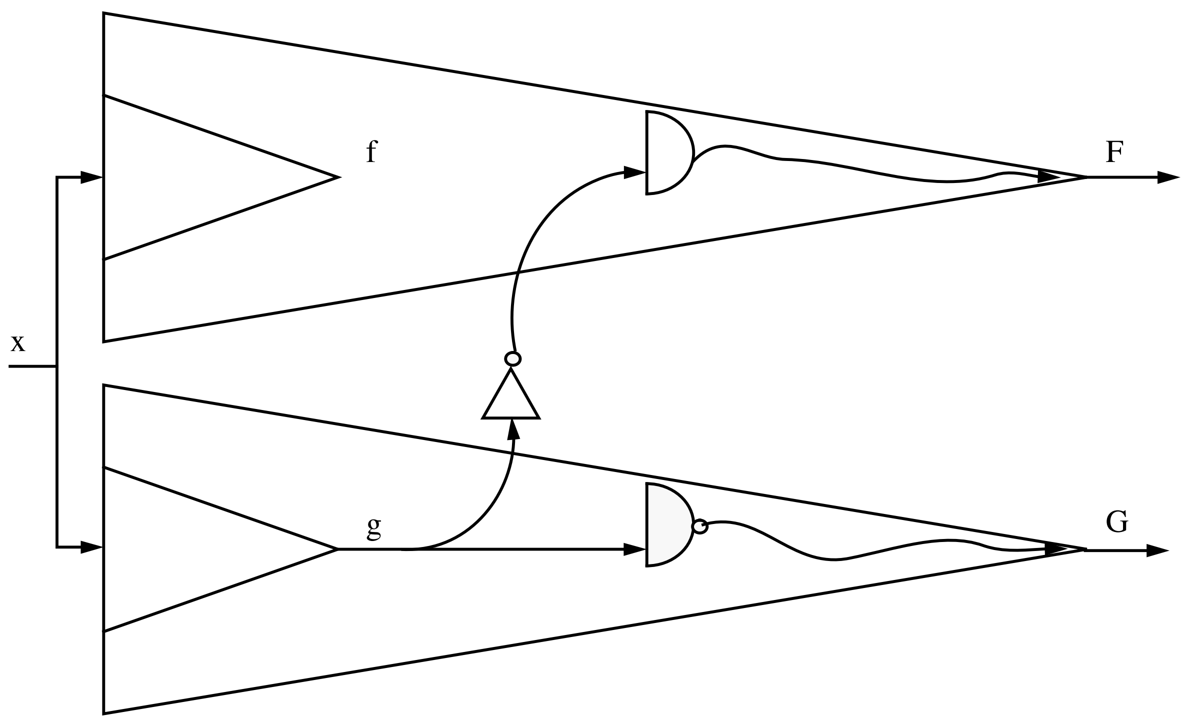

Figure 14. $\bar{g}$ replaces f

and g respectively. We determine whether g may replace f by forming the network of Figure 13, where an XOR gate has been inserted between f and all its fanouts. If the XOR gate's output is not testable for stuck at 1 then we may modify the networks as shown in Figure 14. If the XOR gate's output is not testable for stuck at 0 (rather than 1) then we may modify the networks as shown in Figure 14, except for the inverter. Whenever g or $\bar{g}$ can replace f we record that fact, as that constitutes an element of the sought correspondence. The key to our algorithm is to actually perform the replacement of Figure 14 before asking whether any other functions can be replaced. This reduces the complexity of our algorithm as will be explained next.

Definition 1:

A *miter* consists of a two-input XOR gate, plus the symmetric set difference between the transitive fanins of the two inputs into the XOR gate.

In other words, a miter starts at an XOR gate and extends till primary inputs or till nets shared by both cones of logic. In Figure 13 we have a miter consisting of the XOR gate plus the cones of f and g.

Miters are of concern because test generators tend to perform poorly on miters. While there are test generation strategies specifically designed for miters[12], they are bound to fail if the miter is big enough. Our algorithm is based on the observation that the difficulty for a test generator is dependent mainly on the size of the miter, rather than on the size of the surrounding logic. The key to our approach is to keep applying a test generator to configurations where the size of the miters is small and independent of the size of the design.

The whole process proceeds from inputs of F to its outputs. At each step we have a subfunction f and try to find a subfunction g of G that could replace f. If we find a suitable g we make the replacement, while retaining the functionality of F (Figure 14). At the next step (see Figure 15) the new miter does not extend all the way to primary inputs, but only to g. As a result, while the size of the functions f and g increases, the miter does not, which is essential for keeping the running time short. As we proceed, F keeps resembling G more and more and the process stops at the outputs of F, which should be replaceable by the outputs of G. If an output of G cannot replace the corresponding output of F then the test generator gives an input pattern constituting a counter example to assertion that F and G compute the same function.

The algorithm is given below and is followed by a detailed explanation.

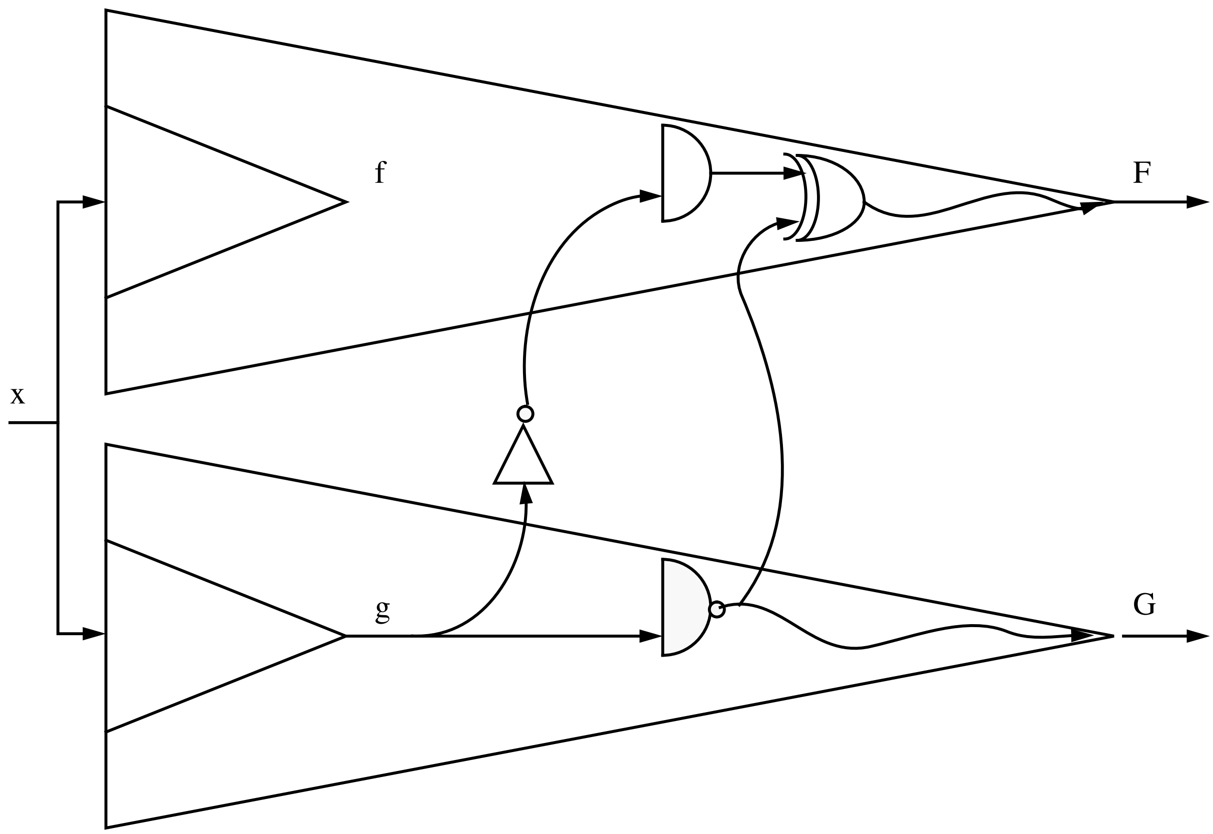

Figure 15. At next step the miter extends only to g

```
0: for each net calculate on which primary inputs it depends
1: form a list fs of nets in F in topological order
2: for each f in fs
       {
3:       form a list gs of some nets in G
4:       for each g in gs
           {
5:           if F(f ⊕ g ) = F(0) then replace f by g
6:           if F(f ⊕ g ) = F(1) then replace f by ḡ
           }
       }
```

For each of the above statements we will give an explanation as well as its computational complexity. Let the number of connections in both F and G be n. We will assume that the number of nets is also of the same order as n.

Statement 0 calculates information used for a heuristic selection of the candidates g in statement 3. For each of the n nets it collects a set of primary inputs whose number could be in the worst case of order n. Therefore its time and space complexity is $O(n^2)$.

Statement 1: For verification [4] not all the nets of F need to belong to fs. On the other hand for incremental synthesis fs consists of all the nets of F because we want a corresponding g to exist for as may nets f as possible. Each corresponding net will be a potential boundary point between logic that can be reused and logic that has to be resythesized. Thus the finer is the correspondence the greater gate reuse can be expect from incremental synthesis.

The topological ordering is done in linear time.

Statement 2: There are n of the nets f.

Statement 3: The list gs consists of candidates who might be able to replace f. The choice of gs has a large effect on performance. If the list gs were too long then we would call the test generator too many times even for nets f that cannot be replaced by any g. On the other hand if gs were too short then we might fail to find a g to replace a particular f, which would make the test generator's job harder later on. Our heuristic forms a list gs giving preference to nets g with a name related to the name of f, followed by nets g that have the same simulation result as f on a small number of random patterns, followed by all remaining nets. In order to minimize the number of calls to the test generator the list gs is pruned as follows:
- We require that each g depend on no more primary inputs than f does.
- We use approximate fault simulation to discard candidates g which are not likely to replace f (as in [1]).

Then the list gs is truncated after k elements; we normally use k = 20. Statement 3 has a worst case complexity O(n) because we may be forced to consider all n nets g. Since this statement is executed n times it contributes $O(n^2)$ to the overall algorithm.

Statement 5 and 6: We call the test generator to determine whether g can replace f. If so, we make the replacement and terminate the loop over gs. This statement contributes only O(n) complexity to the algorithm because we allow the test generator to run only a fixed amount of time independent of the size of the network. For each f the test generator is called at most k times, i.e. a constant number of times.

Thus the overall worst case complexity is $O(n^2)$, but we cannot guarantee to find a replacement g for any f that actually does have a replacement. In order to obtain such a guarantee we would have to allow an unbounded amount of test generation time. In that case the test generator would take an exponential amount of time, exponential in terms of miter size (as long as justification through the miters dominates the test generator's time). Since the miters tend to be large if F and G are structurally very different, the miter size can be considered a measure of structural difference between F and G. In that sense we can say that the algorithm is exponential in the difference between the two networks.

The fact that in our experiments we did not experience an exponential blowup suggests that in synthesized logic it is normally possible to keep the miter size constant and small. This does not imply optimization based on local information only; the test generator does take into account global information.

The result of this calculation is a correspondence between nets of Specification0 and Implementation0. The correspondence assigns to some nets f in Specification0 a net g in Implementation0 together with a polarity. The polarity depends on whether statement 5 or 6 was applied.

4. Structural correspondence

Structural correspondence is calculated between two logic networks; in case of incremental synthesis the two networks are Specification0 and Specification1. The algorithm identifies in what ways the two networks are identical, in what ways they are similar, and in what ways they are completely different. In the example of Figure 5 and Figure 7 the resulting correspondence is indicated by same name, except for the suffix 0 or 1.

Structural correspondence assigns nets in Specification0 to nets in Specification1. Unlike functional correspondence, structurally corresponding nets may have completely unrelated functions. Also unlike functional correspondence we do not require that latch correspondence be given to us; in fact, we have to rely on this algorithm to identify corresponding latches. The problem would be much easier if latch correspondence was given to us by the designer [3], which is actually normally the case for RTL designs. However, if Specification0 and Specification1 are generated by high level synthesis then corresponding latches may have different names; therefore our algorithm must deal with networks containing cycles.

Each of the two networks is represented by a directed graph with vertices and edges. For example, the network of Figure 16 is represented by the graph of Figure 17.There is a vertex for each gate, net, and primary IO. There is an edge for each pin; the edge connects two vertices adjacent in the logic network. Thus the graph may have multiple edges (in case of multiple connections of a net to a gate), but does not have any self loops.

Vertices may have some properties associated with them; for example, a vertex corresponding to a gate has the gate's function. We will refer to these properties as types. The actual interpretation of types is not needed because we will merely need to know whether two types are the same or not. Similarly pins, i.e., edges, have types. For example, a latch may have an input pin of type "data", "clock", "scan-in", etc. An AND gate, say, has all input pins of the same type because they are all commutative.

Suppose that in Figure 16 the box Y represents a latch with data D and clock C. Then in Figure 17 the two input edges of Y have different type, while the two input edges of X have the same type.

The algorithm described in this paper can deal either with gates whose any inputs can be permuted, e.g., AND, or gates whose no inputs can be permuted, e.g., LATCH, but not with gates whose inputs can be permuted with some restrictions, e.g., AO. The way our implementation handles complex gates like AO is very much tied to our specific representation of such gates and therefore will not be described.

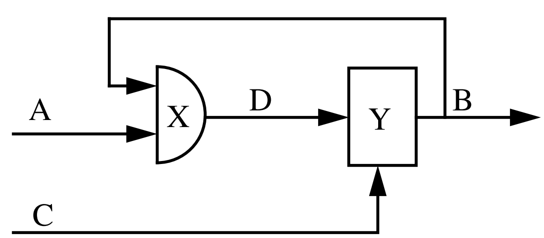

Figure 16: Example of a gate network

We will describe the algorithm as it applies to two directed graphs G and G'. Throughout this section we will use for illustration Figure 18, which is designed to highlight the issues in handling graphs with cycles. The two graphs are not treated in a symmetrical way, but the results do not depend in a significant way on

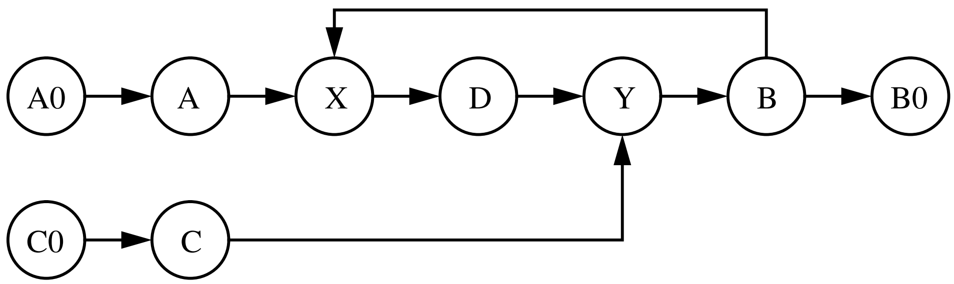

Figure 17: Graph representing the network of Figure 16

which graph represents which version. We will use V, W to denote vertices of G; V', W' to denote vertices of G'; E to denote edges of G; E' to denote edges of G'.

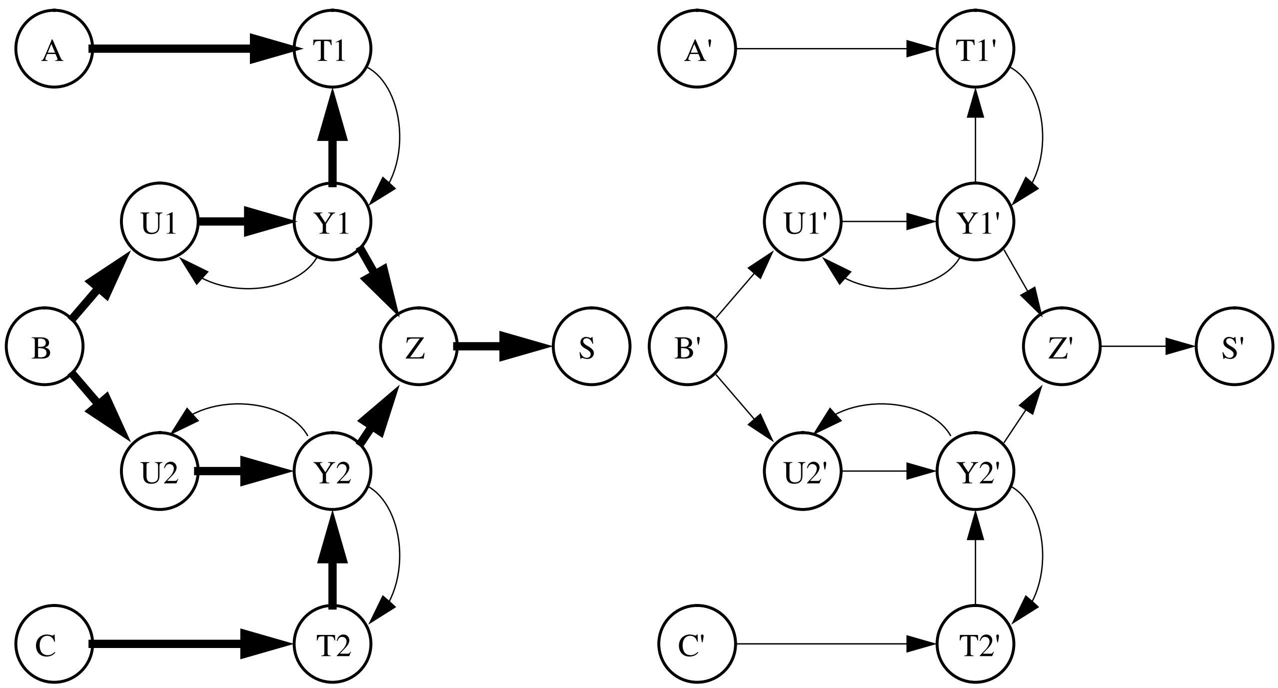

Figure 18: Two isomorphic graphs G and G'

Definition 2:

A *graph* consists of a set of *vertices* and a set of *edges*. For each edge E there is a *source vertex* V and a *sink vertex* W. We will say that E is an *output edge* of V, and W is an *output vertex* of V; E is an *input edge* of W and V is an *input vertex* of W. The *InDegree* of a vertex is the number of its input edges. The *OutDegree* of a vertex is the number of its output edges. A vertex without any input edges is called a *primary input*. A vertex without any output edges is called a *primary output*. Each vertex and edge has an associated symbol, called *type*.

The objective of structural correspondence is to find isomorphism between subgraphs that are isomorphic (Section 4.1), and also find some heuristic correspondence between non-isomorphic subgraphs (Section 4.2).

4.1. Isomorphism

Graph isomorphism is in general a difficult problem with no known polynomial solution. Therefore we simplify the problem by requiring a partial correspondence to be given, and our algorithm extends the initial correspondence to other vertices. Normally the initial correspondence is given for primary inputs and outputs; we will refer to vertices with the initial correspondence as *seeded*.

For example, in Figure 18, Seed(A) = A', Seed(B) = B', Seed(C) = C', Seed(S) = S'. Our goal is to calculate function Corr, which extends Seed and maps, for example, Corr(Y1) = Y1'. Please note that there are two isomorphisms between the two graphs of Figure 18. The given Seed allows only one of them and excludes the isomorphism f, where f(A) = C', f(B) = B', f(C) = A'.

Definition 3:

Given two graphs G, G' and a partial function Seed (from vertices of G to vertices of G'), an *isomorphism* consistent with Seed is a function f mapping vertices of G into vertices of G', mapping edges of G into edges of G', and satisfying
a) f is one-to-one
b) f is a bijection
c) Each vertex V has the same type as f(V)
d) Each edge E has the same type as f(E)
e) An edge E is incident with vertex V iff f(E) is incident with f(V).
f) f(V) = Seed(V) whenever Seed(V) is defined.

The algorithm is performed by the following recursive procedure Isomorphism, which we will first describe briefly and then expand on each of its stages. The procedure Isomorphism is given a function Seed, which we require to be a 1-1 mapping between the vertices of G and G', and it returns an extension of Seed to the other vertices or returns NULL if no isomorphism consistent with Seed exists.

```
Isomorphism(G, G', Seed)

Stage A: Calculate depth-first ordering of G.
Stage B: Calculate InputCorr by one or more passes over G in forward direction.
Stage C: Calculate OutputCorr by one or more passes over G in backward direction.
Stage D: Let Corr be Seed extended by the intersection of InputCorr and OutputCorr.
Stage E: If there are several choices for a vertex, call Isomorphism with
         each of the choices to see if one results in isomorphism.
```

The overall approach of the algorithm can be explained as follows. We propagate the correspondence given by the Seed through the graph (Stages B, C). However, this propagated correspondence captures only local connectivity of individual vertices, and it can be combined into a global isomorphism only if the correspondence is 1-1 (Stage D). If it is not 1-1 then we have to perform brute force exponential search in Stage E. Thus the objective of all the previous stages is to narrow down possible correspondences, so as to reduce the need for Stage E as much as possible. While the narrowing down is essential for good performance it cannot be so aggressive so as to exclude an existing isomorphism; this is the property that will be proved later. We now explain each stage in greater detail.

Stage A:

We perform a depth-first search[2] on G starting from primary inputs. Its purpose is to get a depth-first ordering of all the vertices. There is one pathological situation that we need to take care of first. It can happen that G has no primary inputs or that there are vertices not reachable from primary inputs; in such a case the depth-first ordering would not contain all the vertices. We do not actually require that each vertex be reachable from a primary input, but for our claims below to be true we do need that every vertex be reachable from a seeded vertex. For such sufficiently seeded graphs, without loss of generality, we will assume that any vertex is reachable from some primary input and some primary output can be reached from any vertex. If necessary, this could be accomplished by spliting some seeded vertices into pairs of primary inputs and outputs.

The resulting depth-first ordering is actually a topological order in case the graph is acyclic. In general, it has several properties we will be relying on:
1) Each primary input appears in the ordering before any of its output vertices.
2) Every vertex reachable from primary inputs is preceded by at least one of its input vertices.

In Figure 18 the depth-first ordering is A, B, C, T2, U1, U2, Y1, Y2, Z, T1, S. The thick arrows in the graph G indicate edges whose input vertex precedes the output vertex in the ordering.

This depth-first ordering is used in Stage B. For Stage C we calculate another ordering, namely the depth-first ordering of G with the direction of all edges reversed.

Stage B:

We maintain the following two data structures:
For each vertex W in G, InputCorr(W) is a set of vertices in G', denoted [W_1',..., W_n']. It is initialized to empty.
For each vertex X in G or G', InputBest(X) is a natural number. It indicates the quality of the best correspondence obtained so far involving X. It is initialized to infinity.

Definition 4:

A pair of vertices (W, W') is *sponsored* by k edges iff there are k input edges E_1,..., E_k of W with input vertices V_1,..., V_k respectively, and there are k input edges E'_1,..., E'_k of W' with input vertices V'_1,..., V'_k respectively satisfying the following properties
1) $type(E_i) = type(E'_i)$, for each i,
2) $V'_i \in InputCorr(V_i)$, for each i
3) $V'_i = V'_j$ iff $V_i = V_j$, for each i,j

For example, in Figure 18 assume that InputCorr is same as Seed. Then (U1, U1') is sponsored by 1 edge. In terms of Definition 4, E_1 is the edge between B and U1, V_1 is B, E_1' is the edge between B' and U1', V'_1 is B'. Similarly (U1, U2') is also sponsored by 1 edge.

Stage B iterates through the vertices of G according to the depth-first ordering calculated in Stage A. For each vertex W, Stage B performs the following steps, where the step numbers are used for future reference.

```
if W is seeded then {InputCorr(W) = [Seed(W)];
                     InputBest(W) = 0;
                     InputBest(Seed(W)) = 0;
                    }
else {
    1: Potential(W) = set of all W' such that   type(W) = type(W') and
                                                W' is not seeded
                                                (W, W') is sponsored by any 1 edge
    2: For each W' in Potential(W)
        Discrepancy(W, W') = max(InDegree(W), InDegree(W')) - k
        where k is the largest number such that (W, W') is sponsored by k edges
    3: For each W' in Potential(W)
        InputBest(W) = min(InputBest(W), Discrepancy(W, W'))
        InputBest(W') = min(InputBest(W'), Discrepancy(W, W'))
    4: InputCorr(W) = set of all W' in Potential(W) satisfying both of the following
        1) Discrepancy(W, W') = InputBest(W)
        2) Discrepancy(W, W') = InputBest(W') or
           W has an input vertex with empty InputCorr
 }
```

We now explain Stage B using the example of Figure 18. For each W we describe the computation of the quantities Potential(W), Discrepancy(W,W'), etc., and in Table 1 we show these quantities in tabular form. The second column shows the set Potantial(W) and for each element W' it gives Discrepancy(W,W') separated by a dash.

For the three primary inputs, which are seeded, we get InputCorr(A) = A', InputCorr(B) = B', InputCorr(C) = C', and InputBest is 0 for all of them.

Next we process T2. In Step 1 Potential(T2) is built by considering all the input vertices of T2, such as C. For each of them we consider all their corresponding vertices, such as C'. For each of them we put into Potential their output vertices. Thus Potential(T2) = [T2']. Step 2 calculating Discrepancy(T2, T2') gets k = 1. Therefore Discrepancy(T2, T2') = 1, InputBest(T2) = InputBest(T2') = 1. InputCorr(T2) = [T2'].

Next we process vertex U1 similarly toT2. Potential(U1) = [U1', U2']. Discrepancy(U1, U1') = Discrepancy(U1, U2') = 1. InputCorr(U1) = [U1', U2']. When processing U2, which is symmetrical with U1 we also get InputCorr(U2) = [U1', U2'].

The vertex Y1 has only one input vertex, U1, with non-empty InputCorr, thus Potential(Y1) = [Y1', Y2']. Same as with the previous vertices we get Discrepancy(Y1, Y1') = Discrepancy(Y1, Y2') = 1, InputCorr(Y1) = [Y1', Y2'].

When processing vertex Y2 we obtain a different result than for Y1. This difference is caused by the depth-first ordering, which is not symmetrical. While only one input vertex of Y1, namely U1, has been processed already, both input vertices of Y2 have been processed, which will allow us to narrow down the possible correspondences of Y2. Potential(Y2) = [Y1', Y2']. Discrepancy(Y2, Y1') = 1 because only one input of Y2, namely U2 corresponds to an input of Y1'. In contrast Discrepancy(Y2, Y2') = 0 because Y2 has two inputs U2 and T2 both corresponding to inputs of Y2'. Thus InputBest(Y2) = 0 and InputCorr(Y2) = [Y2'].

For Z we will get Potential(Z) = [Z', T1', T2', U1', U2']. Discrepancy(Z, T1') = 1 because only one input of Z, namely Y1, corresponds to an input of T1'. Similarly Discrepancy(Z, U1') = 1 and Discrepancy(Z, U2') = 1. Discrepancy(Z, T2') is also 1, but for a different reason. Both input vertices of Z correspond to input vertices of T2'. However, only one edge can be used to sponsor (Z, T2') because of condition 3 in Definition 4. When calculating Discrepancy(Z, Z') we have a small optimization problem because InputCorr(Y1) has two elements. Suppose that we try to sponsor the correspondence between Z and Z' by the correspondence between Y1 and Y2'. Then we get k = 1 only; we cannot use the correspondence between Y2 and Y2' to co-sponsor (Z, Z') because of condition 3 in Definition 4. However, if we try to sponsor (Z,Z') by the correspondence between Y1 and Y1' then we can also use the correspondence between Y2 and Y2', yielding k = 2. Therefore Discrepancy(Z, Z') = 0. (This optimization problem of looking for the largest k is a major difference from the acyclic case [3], where it is not necessary.) Thus InputCorr(Z) = [Z'].

When processing T1, all of its inputs already have InputCorr calculated. We get Potential(T1) = [Z', T1', T2', U1', U2']. Discrepancy(T1, T1') = 0, while Discrepancy with the other members of Potential is 1. Therefore InputCorr(T1) = [T1'].

W	Potential(W) - Discrepancy	InputBest(W)	InputCorr(W)
A		0	A'
B		0	B'
C		0	C'
T2	T2'-1	1	T2'
U1	U1'-1 U2'-1	1	U1' U2'
U2	U1'-1 U2'-1	1	U1' U2'
Y1	Y1'-1 Y2'-1	1	Y1' Y2'
Y2	Y1'-1 Y2'-0	0	Y2'
Z	Z'-0 T1'-1 T2'-1 U1'-1 U2'-1	0	Z2'
T1	Z'-1 T1'-0 T2'-1 U1'-1 U2'-1	0	T1'

Table 1. Results of Stage B

We have passed through the graph once calculating for each vertex InputCorr, as the first approximation to the desired isomorphism f. Due to the loops in the graph it is advantageous to repeat the process we just illustrated. In our implementation we iterate the steps of Stage B in an event-driven fashion until no new refinements in InputCorr are possible. If we did that in the example of Figure 18 then we could propagate correspondence along the backward edges, which we could not take advantage of during the first pass. Furthermore, in all subsequent passes InputCorr is non-empty for every vertex, thus we can use InputBest(W') to prune the correspondences (see condition 2 of Step 4). However, in this example we will not continue with Stage B because if we did, InputCorr of each vertex would become a singleton, which would give us the desired isomorphism, and we could not illustrate subsequent stages.

Stage C:

It works the same way as Stage B except that the sense of all the arrows is reversed. In our example we start from the primary output S and derive
OutputCorr(S) = [S']
OutputCorr(Z) = [Z']
OutputCorr(Y1) = [Y1', Y2']
OutputCorr(Y2) = [Y1', Y2']
OutputCorr(T1) = [T1', T2']
OutputCorr(T2) = [T1', T2']
OutputCorr(U1) = [U1', U2']
OutputCorr(U2) = [U1', U2']

We cannot get output correspondence refined any more than that because output correspondence does not make use of seeds on primary inputs, and those seeds are the only thing preferring one of the two isomorphisms over the other.

Stage D:

Stage D calculates a partial function Corr assigning vertices of G' to vertices of G. Whenever we assign Corr(W) = W' then every isomorphism between G and G' consistent with Seed must assign W' to W. Corr is calculated by taking the common vertices of InputCorr and OutputCorr for each W.

```
1: Corr = Seed
2: until no change keep iterating over vertices W whose Corr(W) is undefined
        {
3:        Common =  InputCorr (W) ∩ OutputCorr (W)
4:        if Common is empty then return NULL /*there is no isomorphism*/
5:        InputCorr(W) = Common
6:        OutputCorr(W) = Common
7:        if Common consists of a single vertex [W']
               {
8:              Corr(W) = W'
9:              delete W' from InputCorr(V) and OutputCorr(V) for all V ≠ W
               }
        }
```

In our example, we get
Corr(A) = A, because of statement 1
Corr(B) = B, because of statement 1
Corr(C) = C, because of statement 1
Corr(T1) = T1' because of statement 8
Corr(T2) = T2' because of statement 8
Corr(Y2) = Y2' because of statement 8
Corr(Y1) = Y1' because of statement 9 with V = Y1, W = Y2, then statement 8 with W = Y1
Corr(Z) = Z' because of statement 8
Corr(S) = S' because of statement 8

Corr remains undefined for U1 and U2 and therefore we need Stage E.

Stage E of procedure Isomorphism:

Stage E is needed if InputCorr of some vertex W contains more than 1 element. Stage E does a brute force search, picking each element of InputCorr(W) in turn and calling the procedure Isomorphism recursively to determine if there is an isomorphism consistent with the choice for W.

```
if Corr is defined for all vertices of G then
        if Corr is an isomorphism then return Corr
                                  else return NULL
Let Corr be undefined for W
Let InputCorr(W) = [W1',...,Wk']
for each i
  {
     Corr(W) = Wi'
     f = Isomorphism(G, G', Corr)
     if f is not NULL then return f
  }
return NULL
```

In our example, suppose that we chose W = U2. The first iteration sets Corr(U2) = U1' and the recursive call returns NULL because there is no isomorphism consistent with the given seed. The second iteration succeeds returning an isomorphism.

It should be clear that if the procedure Isomorphism does not return NULL then its result is an isomorphism because that property is checked explicitly before returning any correspondence. However, we have to prove that the procedure Isomorphism will find an isomorphism whenever one exists.

Lemma 2:

Suppose that every vertex of G is reachable from a primary input, that every primary input of G is seeded, and that there exists an isomorphism f between G and G' consistent with the given Seed. Then InputCorr(W) contains f(W) for all W processed by Stage B. Moreover, InputBest(W) = number of input edges of W, whose source vertices have empty InputCorr at the time W is processed by Stage B.

Proof:

The proof is by induction on the number of steps to perform Stage B. The claim of Lemma is clearly true for primary inputs, which are seeded. So assume that W is not a primary input and that the lemma is true for all vertices processed before W. Let E_1,..., E_n be all the input edges of W such that their source vertices V_1,..., V_n have non-empty InputCorr. By property 2) of depth-first ordering and by induction hypothesis, n > 0. Let the number of remaining input edges, whose source vertices have empty InputCorr, be m.

As a result of Step 1 of Stage B Potential(W) includes f(W) because by induction hypothesis, InputCorr(V_1) includes f(V_1), which is an input vertex of f(W) (by definition of isomorphism). Similarly please note that using the induction hypothesis and the fact that f is an isomorphism, all the edges E_1,..., E_n sponsor (W, f(W)). Therefore the number k computed by Step 2 must satisfy $k \geq n$. On the other hand none of the remaining m edges can be used to sponsor (W, f(W)) and therefore k = n. Thus
Discrepancy(W, f(W)) = max(InDegree(W), InDegree(f(W))) - k = m.

We will be finished if we can show that in step 4 both conditions are satisfied for including f(W) in InputCorr(W). Since none of the m edges can be used to sponsor any (W, W'), we get InputBest(W) = m, which makes condition 1) of step 4 satisfied. Condition 2) is automatically satisfied if m > 0. If m = 0 then Discrepancy(W, f(W)) = InputBest(f(W)) = 0.

❑

Theorem 1:

Suppose that every vertex of G is reachable from a primary input, suppose that every primary input of G is seeded, and suppose that there exists an isomorphism between G and G' consistent with the given Seed. Then the procedure Isomorphism will return an isomorphism between G and G' consistent with Seed.

Proof:

The proof is by induction on the number of non-seeded vertices. If this number is 0, then the procedure Isomorphism returns Seed itself, which by the premisses of the Theorem is an isomorphism. Now consider a given Seed and assume the theorem to be true for all seeds defined on more vertices than Seed itself. By Lemma 1 at the end of Stage B InputCorr(W) will contain f(W) for all isomorphisms f. By symmetry we can apply Lemma 1 to Stage C and get that OutputCorr(W) also contains f(W) for all isomorphisms f.

We have to argue that Stage D will not discard any isomorphism. First of all, the intersection of InputCorr(W) and OutputCorr(W) also contains all the f(W) and therefore steps 5 and 6 of Stage D do not discard any isomorphism.

Step 9 of Stage D will not discard any isomorphism for the following reason. Whenever Common consists of a singleton W', all isomorphisms map W to W'. Therefore no isomorphism assigns W' to any $V \neq W$ and hence W' can be removed from InputCorr(V). For the same reason, whenever Step 8 defines Corr(W), all isomorphisms consistent with Seed are also consistent with Corr.

In Stage E let i be the smallest index such that there is an isomorphism f mapping f(W) = W_i.Then the i-th invocation of Isomorphism receives input, which is more defined than the given Seed and still has an isomorphism consistent with it.Therefore by induction hypothesis the i-th invocation of Isomorphism returns an isomorphism.

❑

4.2. Non-isomorphism

Up to now we have concentrated on using the algorithm for finding an isomorphism. However, in case of incremental synthesis the two given graphs are normally not isomorphic. Incremental synthesis needs two pieces of information; it needs to know which primary outputs and latches have isomorphic input cones, and it needs some correspondence even between non-isomorphic portions of the designs. The latter is by necessity heuristic and we cannot prove any properties of the correspondence between non-isomorphic portions.

If the two graphs are not isomorphic the correspondence computed by the algorithm will satisfy condition a, d, f, Definition 3, but may violate the definition of isomorphism in the following ways:
b) A vertex or edge might not get anything to correspond to it.
c) Two vertices with different types may correspond.
e) The incidence property may be violated.

We use a procedure Correspondence(G, G', Seed), which is identical to the procedure Isomorphism except for the following differences. First, it always returns some correspondence, provided some vertices are seeded. Thus step 4 of Stage D does not return NULL when Common is empty; instead it merely stops processing W. Secondly, in step 1 of Stage B Correspondence does not require that type(W) = type(W'); instead if the two types are different it increases Discrepancy by a certain heuristic amount. Thirdly, Stage E does not try every element of InputCorr(W) to see if isomorphism can be derived. Instead it heuristically chooses one such element.

Stage E of procedure Correspondence:

```
Chose W so that Corr(W) is undefined, but InputCorr(W) or OutputCorr(W) is non-empty
If there is no such W then return Corr
Set Corr(W) = W' for some W' in InputCorr(W) or OutputCorr(W)
return Correspondence(G, G', Corr)
```

As we said above, besides computing the correspondence, we also need to know which primary outputs and latches have an EC in their cone. That calculation is performed by the procedure Correspondence before any arbitrary choice is made in Stage E, as will be explained next. For that we define an InputCone of a vertex Y to represent the combinational logic feeding Y. Every input vertex of InputCone(Y) will be either a primary input or a vertex representing a latch.

Definition 5:

For a vertex Y in G or G', *InputCone(*Y*)* is a subgraph. An edge E belongs to InputCone(Y) if Y is reachable from the edge E without passing a vertex representing a latch. A vertex belongs to InputCone(Y) if one of its input or output edges belongs to InputCone(Y). A vertex X is an *input vertex* of InputCone(Y) if no input edge of X belongs to InputCone(Y).

We check which InputCones are isomorphic using the following code executed optionally before stage E of the procedure Correspondence.

```
for every W with Corr(W) = W' defined and satisfying
    W and W' are primary outputs or they represent latches
    for every V in InputCone(W) with Corr(V) defined, Corr(V) is in InputCone(W')
    every vertex X in InputCone(W) or InputCorr(W') has InputBest(X) = 0
    for every input vertex V of InputCone(W), Corr(V) is an input vertex of InputCone(W')
  {
      f = Isomorphism(InputCone(W), InputCone(W'), Corr)
      Corr(V) = f(V) for all V having f(V) defined
  }
```

The checking for isomorphism is done using the procedure Isomorphism, and seeding all those vertices for whom Corr has already been defined. Therefore we would not do the isomorphism checking if for some V in InputCone(W), Corr(V) was defined and was outside InputCone(W'). Also there is no need to check the input cone of W unless all the vertices inside InputCone(W) and InputCone(W') have InputBest of 0, a necessary condition for isomorphism (Lemma 2). (It is not a sufficient condition for isomorphism as shown in Figure 19.)

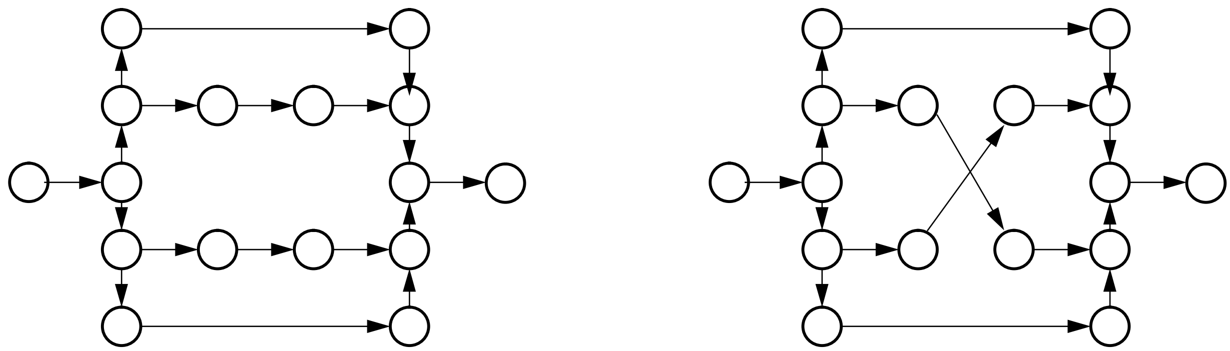

Figure 19. Non-isomorphic graphs whose all vertices have InputBest and OutputBest of 0

We will also not check for isomorphism of InputCone(W) unless its output W as well as all its inputs have Corr defined. The reason is that doing the isomorphism checking without already having a correspondence for all inputs and outputs would take too much time, which is not worthwhile. If the original designer's input was on register transfer level then latches are seeded and therefore Corr will be defined for inputs and outputs of any InputCone. The correspondence for a latch might not be given to us only if the design came through high level synthesis, which generated some latches automatically. The designer is not even aware of those latches and therefore does not require their implementation to be preserved.

After checking input cones for isomorphism we use any isomorphism found to extend Corr and further refine InputCorr and OutputCorr as done in Stage D.

Having described the two procedures Isomorphism and Correspondence, we consider the question of their correctness. For Isomorphism, it is simple. If the two graphs are isomorphic then the procedure Isomorphism will find that as proved in Theorem 1. And if the procedure Isomorphism claims that two graphs are isomorphic then it is so because it was checked explicitly in Stage E.

The question of correctness is more difficult for the procedure Correspondence, which handles (heuristically) non-isomorphic graphs. One requirement is that it identifies isomorphic input cones. Our procedure does that for graphs where latch correspondence can be established as stated above. However, even this simple requirement is not so simple as shown in Figure 20. For the purposes of incremental synthesis we may identify only one of InputCone(X) or InputCone(Y) as isomorphic to its counterpart in G'. Which one, will be determined by the order in which input cones happen to be checked for isomorphism.

The computational complexity of the algorithm is dominated by Stage E of the procedure Isomorphism, which may cause an exponential search for an isomorphisms. A second possibility of an exponential behavior is Step 2 of Stage B, involving a covering problem. In practice neither causes a problem; the most important consideration for performance is to keep the size of InputCorr and OutputCorr down, because that will keep down the size of Potential in Stage B. For that reason in our implementation stages B and C are executed in alternation and Stage D is incorporated inside them, so as to reduce the size of InputCorr and OutputCorr as soon as they are generated.

Figure 20. Non-isomorphic graphs, but both primary outputs have isomorphic input cones.

5. LOGIC_REUSE algorithm

As described in Section 2, LOGIC_REUSE is performed after FUNCTIONAL_CORRESPONDENCE and STRUCTURAL_CORRESPONDENCE. Those two correspondences are composed by transitivity to obtain a correspondence between the nets of Specification1 and Implementation0, see Figure 11. (The correspondences are calculated before LOGIC_REUSE starts and does not change during its execution.)

For correspondence of primary IOs and latches we do not rely on the functional correspondence calculated in Section 3. Instead we use the explicit correspondence mentioned at the beginning of Section 3 (e.g. correspondence of primary IOs is by name). There are two reasons. First of all, if Specification0 and Implementation0 are not functionally equivalent then the algorithm of Section 3 would not relate corresponding primary outputs. Secondly, even if Specification0 and Implementation0 are functionally equivalent, it is possible that a primary output is equivalent to several nets; for incremental synthesis we need to make sure that each primary output is related to its corresponding primary output and nothing else.

When a net X1 in Specification1 corresponds to a net x0 in Implementation0 then it is likely that x0 (or $\overline{x0}$) can functionally replace X1. We could be assured of the legality of such a replacement if Specification0 and Specification1 were identical and if the replacements were carried out in the order calculated by functional correspondence. However, since the two specifications are not identical we must check it explicitly and make the replacement only if it does not change the function of Specification1.

LOGIC_REUSE is done in four phases. Phase 0 makes sure that unmodified input cones preserve their implementation. Phase 1 deals with primary inputs and latches, Phase 3 deals with primary outputs and latches, while Phase 2 handles the combinational logic in between. We will first explain the algorithm only in the simplest case, where there are no latches and the new and old versions have the same primary IOs.

Phase 0 uses the information from structural correspondence calculation as to which input cones are isomorphic. In our example of Figure 5 and Figure 7, InputCone(Z0) and InputCone(Z1) are isomorphic. The implementation of the isomorphic cones is marked as “committed”. This marking is applied to every input pin in the implementation cone and prevents the disconnection of the pin by later stages of the algorithm. In our example of Figure 6 all the input pins in InputCone(z0) are marked as committed, which guarantees that the whole cone will remain as is. In the sequel we will remove from our figures those gates that lie in InputCone(z0) or InputCone(Z1) only, not because they are removed in our implementation, but merely to simplify our figures.

Phase 1 places Specification1 and Implementation0 together, sharing primary inputs only (see Figure 21). The resulting network still has two sets of primary outputs.

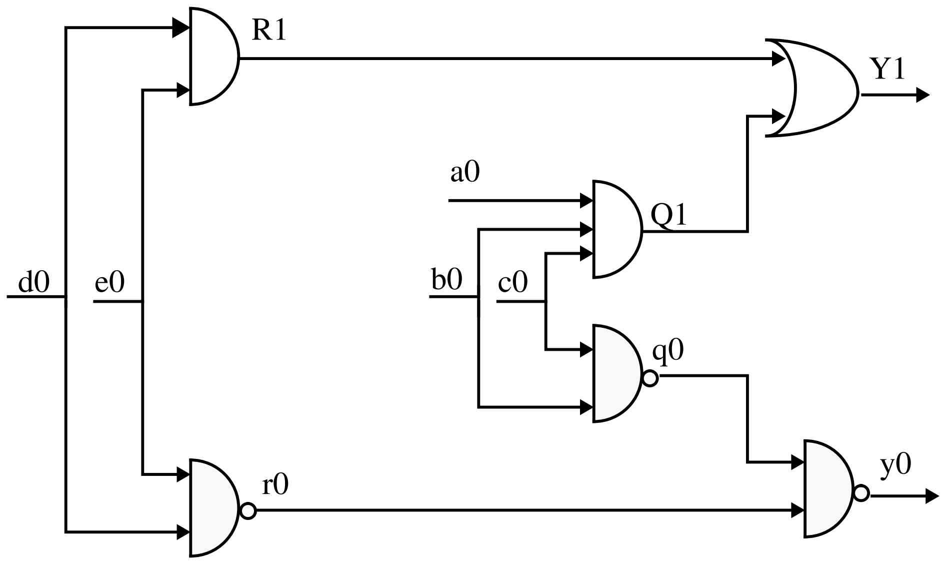

Figure 21. Result of Phase 1

Phase 2 takes the two pieces of logic sharing primary inputs only, and introduces more sharing between them. The result of Phase 2 is a network, where Specification1 and Implementation0 are interconnected, and some pieces of logic are left unused. Phase 2 may change the functionality of the primary outputs for Implementation0 (y0), but it must preserve the functionality of the primary outputs of Specification1 (Y1).

Phase 2 proceeds through the nets of Specification1 in topological order from inputs to outputs. Given a net X1 in Specification1, the composition of structural and functional correspondence gives us a corresponding net x0 in Implementation0. We check whether x0 may replace X1 without changing the functionality of Specification1 (using Lemma 1).
Case A: If the answer is "yes", then the net x0 (or $\overline{x0}$) replaces X1 at all its sinks.
Case B: If the answer is "no", then the net X1 (or $\overline{X1}$) replaces x0 at all its uncommitted sinks.

Case A (the desirable case) represents the situation of logic X1 that was not modified and thus can use the gates of x0. The cone of X1 becomes unnecessary. Case B represents logic X1 that was modified and thus cannot use the gates of x0. In Case B, replacing x0 with X1 makes it likely that as the algorithm proceeds it will encounter Case A further on, as will be illustrated later.

In our example, we first apply Case A to R1 and r0 with negative polarity, see Figure 22. This step effectively reused the gate r0. It is connected through an inverter because the functional correspondence between R0 and r0 is negative.

Then we apply Case B to Q1 and $\overline{q0}$ (see Figure 23). It has the effect of replacing the gate q0 with Q1 plus an inverter. This is because the EC involves the gate q0, which cannot be reused.

Lastly we apply Case A to Y1 and y0 (see Figure 24). Please note that originally (Figure 21) we could not do that because Y1 and y0 had different function. As a result of applying Case B to Q1 and $\overline{q0}$, Y1 and y0 now have the same function. This is an example, how Case B enables Case A later on thus effectively limiting the effect of the EC (to gate q0 only).

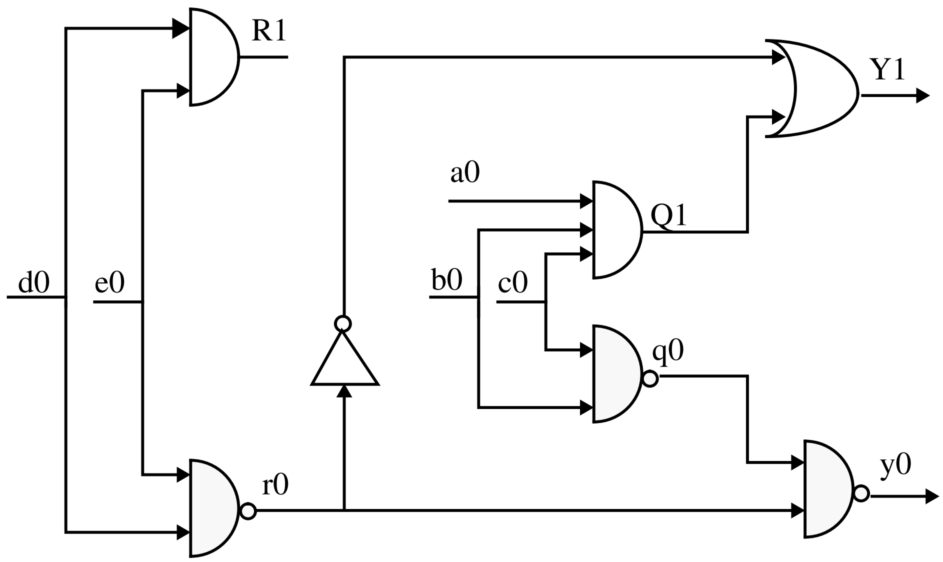

Figure 22. After applying Case A to R1 and $\overline{r0}$

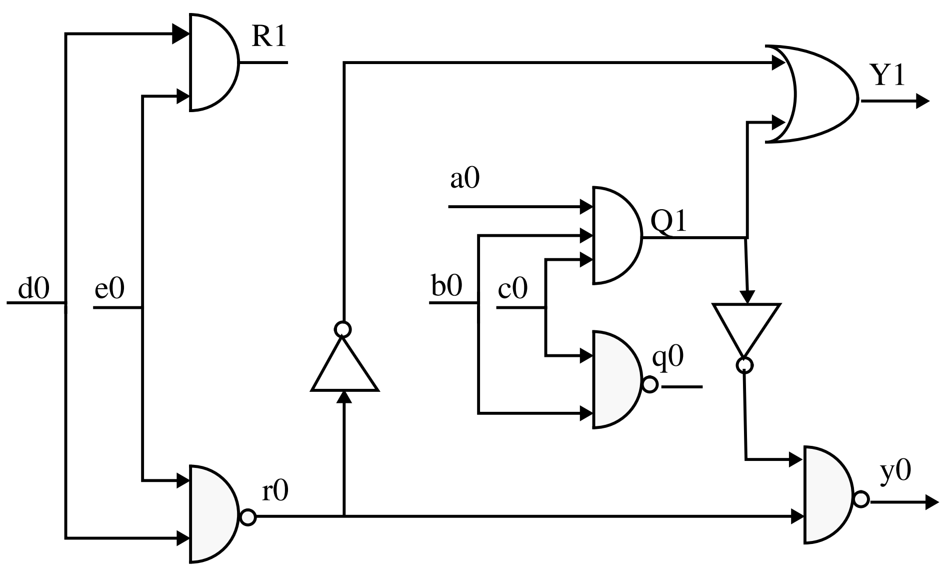

Figure 23. After applying Case B to Q1 and $\overline{q0}$

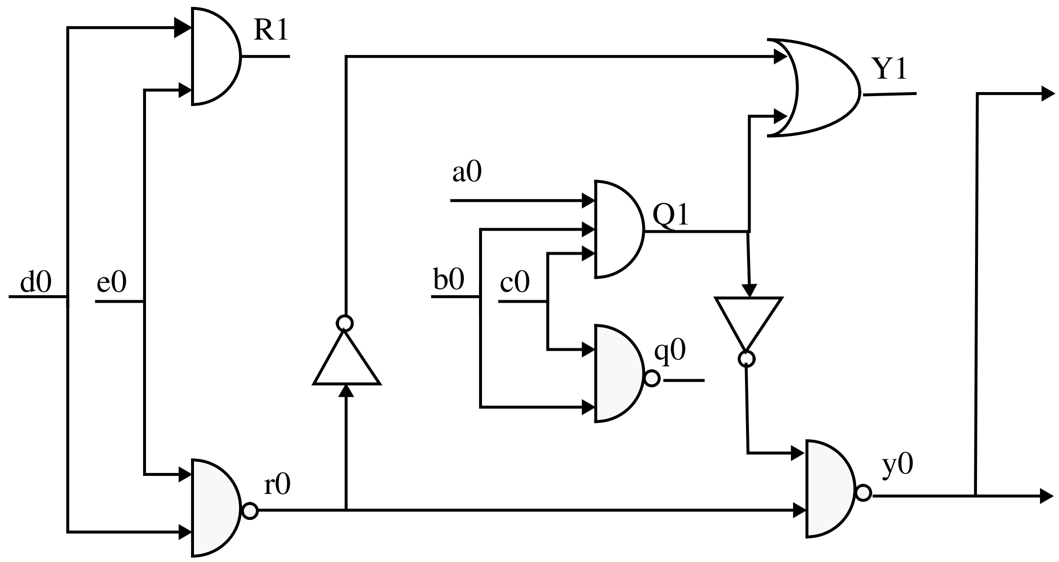

Figure 24. After applying Case A to Y1 and y0

After Case A the functionality of Specification1 is preserved because it was checked explicitly before applying Case A. After Case B, it is certainly possible for the functionality of Implementation0 to change, which is all right. However, the way we described it above, it is also possible for the functionality of Specification1 to change, which is not all right. The reason is illustrated in Figure 25. It shows the situation after applying Case A to U1 and u0. If we then applied Case B to V1 and v0 we would change the functionality of Specification1, which we cannot allow. The cause of the problem is the fact that proceeding in topological order in Specification1 does not imply that the corresponding nets in Implementation0 will be in topological order.

To guarantee that Case B will never change the functionality of Specification1 we use the same "com-

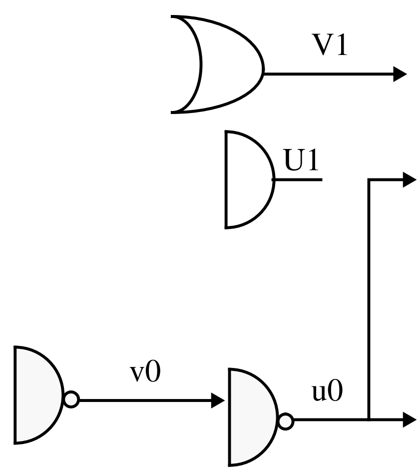

Figure 25. After applying case A to U1, u0 we cannot apply Case B to V1, v0

mitting" mechanism mentioned above for isomorphic cones. After performing Case A on some U1 and u0 we commit all the new connections of u0 as well as the whole input cone u0. That means, all those input pins are marked as committed and may not be changed.

Figure 24 shows the network after Phase 2 is finished. We are still left with two sets of primary outputs (Y1 and y0). Each pair of corresponding primary outputs ends up connected to the same net (y0 in Figure 24). This property is guaranteed because we made sure that primary outputs correspond to each other and no other nets. Phase 3 simply deletes all the primary outputs of Specification1. The result is a network with the functionality of Specification1 and with as many gates of Implementation0 as introduced by Case A.

The result can be seen in Figure 10, where we deleted unused gates and reintroduced the gates in InputCone(z0). This has to be run through regular synthesis, where all the shaded gates are protected from any transformations. The result is shown in Figure 9.

We now discuss some more complicated situations. Our general approach tries to handle all complications (the vast majority of which involve latches) in Phase 1 and 3. Phase 2 remains unchanged. For some of these complications there is no unique correct solution and the designer needs to chose the one most appropriate for his methodology.

Different primary IOs in Implementation0 and Specification1

If the designer adds to Specification1 a primary input or output X non-existent in Implementation0 then Phase 1 and 3 simply keep X; there is no corresponding old primary IO to merge it with.

Now consider the opposite situation when the designer removes from Specification1 a primary IO X that exists in Implementation0. If X is a primary output then it is deleted during Phase 3. If X is a primary input then it is deleted during Phase 1 and its net is connected to an arbitrary constant (0 or 1). Normally Specification1 will have no use for that primary input X, and the constant net will become irrelevant. However, it is also possible that the new implementation reuses the logic with the constant. That is not a problem, provided it is later mapped into the proper technology.

Latches

We try to treat latches as inputs and outputs of combinational logic, but there are some important differences. One difference is that latches contain some boolean functions; for example, data and clock are combined together, it is common for a latch to have several data inputs that are ANDed or ORed inside, also latches tend to have several outputs (different polarities, master slave, scan). We view a latch as a primitive delay element with some surrounding logic. Examples of surrounding logic are clock and data ANDed together, or inverters for some outputs. We can handle any kind of surrounding logic with an important restriction -- we require that each latch output is the output of the delay element, possibly inverted; we could not handle latches whose outputs involve some boolean combination of the latch inputs.

During Phase 1 we disconnect the output X1 of any latch R1 that has a matching latch r0 in Implementation0, and we make X1 come from r0 (inverted if r0 provides a different polarity that R1). If R1 does not have a matching r0 (R1 is a new latch non-existent in the old version) then we leave it as is (same as newly added primary IO). We also delete any latch r0 in Implementation0 that does not have a matching latch in Specification1, and connect its output to a constant (same as primary input deleted in new version).

At the end of Phase 2 it may not be true that corresponding latches will end up functionally equivalent. This can happen because of the boolean functions packaged inside latches. Therefore in Phase 3 we have to check whether the delay element inside R1 computes the same function as the delay element inside the corresponding r0. If the answer is “yes” then we can delete R1. If the answer is “no” then we have to delete r0 and make the output come out of R1 (inverted if necessary).

Some technologies have cells containing several latch bits. For our purposes they can be treated same as single bit latches with one exception. If one bit of a multi-bit latch cannot be used in the new version then we do not reuse any of the others either.

Implementation0 and Specification0 are not functionally equivalent

This can happen in two ways. First consider the case that Implementation0 is wrong and the designer uses incremental synthesis to correct it. In that case we omit Phase 0 so that even unmodified primary outputs are assured to get the functionality of Specification1.

Another possibility is that Implementation0 is intentionally different from Specification0. A typical example are modifications for testability. A designer may add extra control points involving latches that did not exist in Specification0. Naturally, the designer does not want to add the same testability logic in every new version.

Recall that we said above that we delete latches r0 that do not have a corresponding latch R1 in Specification1. This applies only to those r0 which do have a corresponding latch R0 in Specification0. We do keep a latch r0 if it has no corresponding R0 nor R1, because r0 was added to Implementation0 and hence we assume that the designer wants it in Implementation1 as well.

The algorithm is run as described above including Phase 0, which ensures that primary outputs with isomorphic cones in both specifications will use the logic in Implementation0 whether it is functionally equivalent or not. Thus the testability structures will be preserved in cones unaffected by the EC, but they will have to be reinserted into modified cones.

Scan lines

Both boundary scan and internal scan lines present special problems, because our algorithm treats them the same as other logic. Up to now we were making sure that the logic of Implementation1 was correct in a combinational sense; however, the correctness of scan lines involves more than combinational properties. For example, suppose that Specification0 has two latches connected in a scan line as A0-B0, Specification1 has the corresponding latches A1-B1, and Implementation0 has latches a0-c0-b0. The result of our algorithm contains latches a0, b0, c0, where both b0 and c0 get their scan input from a0, which is clearly not a correct scan line. Therefore we use a program that corrects the scan line afterwards, while trying to preserve it as much as possible. In examples like the one above, it is not clear what the designer meant, therefore we sometimes make heuristic choices as to the scan line ordering.

Clock lines

Our algorithm will preserve the old clock distribution network whether the new version has the same number of latches or not. If the number of latches changes then it may be necessary to completely redo the clock network. In contrast to the case of scan lines, we do not have any special programs that would try to reuse parts of the old clock network.

6. Experimental results

All of the circuits for our experiments were taken from production parts, with the exception of C6288, which was taken from [7] and a random change was introduced. (We included C6288 so that the reader can relate the other pieces of logic to something that has been discussed in the literature.) Examples A1 and A2 are identical, except that in the case of example A1, Specification0 and Specification1 were run through different preprocessing. Examples D1, D2, D3 are three variations on the same basic design.

Table 2 compares the size of the EC in the Specifications to its impact on the implementation obtained by incremental synthesis (including regular synthesis of modified logic and the post processing step). Designers expect a large difference in the implementation only if there is a large difference in the specifications. We measure the size of the EC in terms of lines of VHDL (column SOURCE LINES). We report the total number of lines in Specification0 (column old), how many of them appear in Specification1 (column reused) and how many new or modified lines appear in Specification1 (column new). For the implementation (column IMPLEMENTATION GATES) we report analogous numbers, except in terms of gates.

Two of the examples, namely A1 and C, stand out as having an excessive percentage of new gates. In example C this is caused by the fact that percentage of changed lines and gates is not always a good measure of reuse. When counting lines of VHDL we include blank lines, comment lines etc., while such "non-executable" objects do not exist in the implementation. If measured by the number of new gates per new line of VHDL, example C is not excessive.

In contrast, example A1 points to a genuine weakness of this whole approach. In case of A2, the designer's VHDL was fed directly into incremental synthesis, which therefore saw a mere one line difference between Specification0 and Specification1. In case of A1, Specification1 was optimized before given to incremental synthesis, while Specification0 was not. Thus the two specifications looked completely different to incremental synthesis. An incremental synthesis method that would consider only the function and not the structure of the specifications should perform equally well on examples A1 and A2, because Specification1 has the same function for both A1 and A2. Our method does rely on the structure of the specifications, and in particular requires that the new function be realized by inserting portions of Specification1 into Implementation0. We do that for two reasons. One is efficiency. The other is the need to use incremental synthesis to realize not a different function, but a different structure -- designers often change their specification to obtain a faster design rather a new functionality.

NAME	SOURCE LINES				IMPLEMENTATION GATES			
	old	reused	new	% new	old	reused	new	% new
A1	7315	7314	1	0.1%	973	800	251	25.8%
A2	7315	7314	1	0.1%	973	952	3	0.3%
B	1440	1439	1	0.1%	2676	2675	2	0.1%
C	2514	2492	51	2.0%	940	895	134	14.2%
D1	12792	12785	19	0.1%	4199	4137	107	2.5%
D2	12804	12795	23	0.2%	4124	4077	93	2.2%
D3	12792	12779	41	0.3%	4199	4152	64	1.5%
C6822	6735	6734	1	0.1%	2210	2210	0	0.0%

Table 2. Amount of reuse in Specification and Implementation

In Table 3 we consider the question of penalty (area and delay slack) incurred by using incremental synthesis. Under the column "incremental" we report the result of using incremental synthesis, including regular synthesis of modified logic and the post processing step. Under the column "regular" we show the result of synthesizing the whole Specification1 by regular synthesis only. The column "% increase" shows the penalty (when positive) or the benefit (when negative) of using incremental synthesis.

In Table 4 we compare CPU times (in seconds) for the two ways of synthesizing Specification1. In case of incremental synthesis we report separately the time to calculate functional correspondence because that can be done before Specification1 exists and thus need not impact the turn around time. The column "incremental" includes CPU time for structural correspondence as well as LOGIC_REUSE. The column "regular" reports the time needed by regular synthesis and the post-processing step.

NAME	AREA (CELLS)			SLACK (NS)	
	incremental	regular	% increase	incremental	regular
CARB1	11498	10245	12.2%	-0.8	-0.8
CARB2	9806	9949	-1.4%	-0.8	-0.9
GP	6622	6621	0.1%	-1.7	-1.7
FXDCD	3238	3092	4.7%	-0.9	-0.8
CCTL1	40116	39491	1.5%	-1.0	-1.0
CCTL2	39938	38937	2.5%	-1.2	-0.9
CCTL3	39715	38927	2.0%	-1.0	-0.9
C6822	7018	7018	0.0%	-5.9	-5.9

Table 3. Quality of results with incremental synthesis vs. regular synthesis only

NAME	INCREMENTAL SYNTHESIS				REGULAR ONLY
	functional	incremental	regular	total	
CARB1	232	124	698	1054	1110
CARB2	232	403	234	869	855
GP	128	244	315	687	806
FXDCD	26	40	207	273	561
CCTL1	3718	1168	1275	6161	4485
CCTL2	3588	1449	1244	6281	4381
CCTL3	3718	1283	1192	6193	4381
C6822	51	108	209	368	1458

Table 4. CPU times (sec)

7. Conclusions

We have presented an automated way of performing incremental synthesis. Its main benefit lies in reducing the design cycle, which happens in several ways. First, it automates the implementation of specification changes late in the design cycle. Secondly, it allows a designer to make modifications (e.g. speed up) in one area without changing the implementation of areas he is already happy with. And thirdly, it preserves the information from the old version (e.g. net names), which helps in analyzing the new implementation.

8. Acknowledgments

D. Kung, L. Stok, L. Trevillyan read the manuscript and made many helpful suggestions. We are especially grateful to Andreas Kuehlmann for valuable discussions. Lakshmi Reddy made an important contribution to the test generator, on which this approach is based.

9. References

[1] M.Abramovici, P.R.Menon, D.J.Miller, "Critical Path Tracing: An Alternative to Fault Simulation", *IEEE Design and Test,* Vol.1, February 1984, pp. 83-93.
[2] A.V. Aho, J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley 1977.
[3] D. Brand, "The Taming of Synthesis", *Proc. of International Workshop on Logic Synthesis*, RTP, May 1991.
[4] D. Brand, "Verification of Large Synthesized Designs", *Proc. of ICCAD*, November 1993, pp. 534-537.
[5] D. Brand, R. Damiano, L. van Ginneken, A. Drumm, "In the Driver's Seat of BooleDozer", *Proc. of ICCD*, Oct. 1994, pp. 518-521.
[6] R. Brayton, G. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms*

for VLSI Synthesis, Kluwer Academic Publishers, 1984.
[7] F. Brglez, P.Pownall, R. Humm, “Accelerated ATPG and Fault Grading via Testability Analysis”, *IEEE International Symposium on Systems and Circuits*, June 1985, pp. 695-698.
[8] R.E. Bryant, “Graph Based Algorithms for Boolean Function Manipulation”, *IEEE Transactions on Computers,* Vol. 35, Aug. 1986, pp. 677-691.
[9] P.Y. Chung, I.N. Hajj, “ACCORD Automatic Catching and CORrection of Logic Design Errors in Combinational Circuits”, *Proc. of International Test Conference,* September 1992.
[10] M. Fujita, T. Kakuda, Y. Matsunaga, “Redesign and Automatic Error Correction of Combinational Circuits”, *Proceedings of the IFIP TC10/WG10.5 Workshop on Logic and Architecture Synthesis*, North-Holland, May 1990, pp. 253-262.
[11] M. Fujita, Y. Matsunaga, K.C. Chen, “On Application of Boolean Unification to Combinational Logic Synthesis”, *Proc. of ICCAD*, November 1991, pp. 510-513.
[12] P.Goel, “An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits”, *IEEE Transactions on Computers*, Vol. C-30, March 1981, pp. 215-222.
[13] R.P.Kunda, P.Narain, J.A.Abraham, B.D.Rathi, “Speed up of Test Generation using High Level Primitives”, *Proc. of 27th Design Automation Conference*, June 1990, pp. 594-599.
[14] S.Kundu, L.H.Huisman, I.Nair, V.S.Iyengar, L.N.Reddy, “A small Test Generator for Large Designs”, *Proc. of International Test Conference*, Sept. 1992, pp. 30-40.
[15] J.C. Madre, O. Coudert, J.P. Billon, “Automating the Diagnosis and the Rectification of Design Errors with PRIAM”, *Proc. of ICCAD*, November 1989, pp. 30-33.
[16] S. Muroga, Y. Kambayashi, H.C. Lai, J.N. Culliney, “The Transduction Method -- Design of Logic Networks Based on Permissible Functions”, *IEEE Transactions on Computers*, Vol 38, No. 10, October 1989, pp. 1404-1424.
[17] I. Pomeranz, S.M. Reddy, “On Diagnosis and Correction of Design Errors”, *Proc. of ICCAD*, November 1993, pp. 500-507.
[18] T.Shinsha, T. Kubo, Y. Sakataya, J. Koshishita, K. Ishihara, “Incremental Logic Synthesis Through Gate Logic Structure Identification”, *Proc. of DAC*, June 1986, pp. 391-397.
[19] Y. Watanabe, R.K. Brayton, “Incremental Synthesis for Engineering Changes”, *Proc. of ICCD*, November 1991, pp. 40-43.