RC 20127 (9/24/95)
Computer Science

IBM Research Report

Don't Cares in Synthesis: Theoretical Pitfalls and Practical Solutions

D. Brand, R. A. Bergamaschi, L. Stok

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

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

Don't Cares in Synthesis: Theoretical Pitfalls and Practical Solutions

D. Brand R. A. Bergamaschi L. Stok
IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

Abstract

Effective use of don't cares requires solving several theoretical and practical problems. The theoretical problems are caused by a need to have all tools in a methodology use a consistent semantics of don't cares, so as to guarantee correctness of the final implementation. Several common meanings of "don't care" will be considered and their respective conditions for design correctness will be derived. Several practical problems involving one kind of don't care, where design correctness can be guaranteed, will be discussed. These practical issues include: specifying don't cares in a language description, deriving them during high-level synthesis, and optimizing logic in their presence. Experimental results showing the impact of don't cares on logic quality are presented.

1 Introduction

Current digital designs are done in a range of abstraction levels. In most cases the design is described in a hardware-description language and may involve behavioral, register-transfer (RT) and logic or gate-level partitions. Current synthesis systems operate mostly at the RT level and gate level, but high-level synthesis techniques are gradually being incorporated into existing systems. This differentiation of abstraction levels has important effects on specification, synthesis and quality of results. The use of don't-care conditions has implications on the whole design process, including methodology, specification, synthesis and verification. This paper analyzes several issues related to don't cares from a theoretical as well as a practical perspective.

A typical example of a don't care situation, which will be referred to in later sections, is shown in Figure 1. The **instruction decoder** in Figure 1 has two outputs, *ALU_control* and *is_legal*. The latter is a one bit signal indicating whether the instruction is legal. While the logic inside *Instr_decode* is supposed to be synthesized in isolation, the designer is familiar with the rest of the design and therefore can tell synthesis that *ALU_control* is irrelevant for illegal instructions. Synthesis should then use this don't care information in optimizing *Instr_decode*.

In this paper, a "don't care" will mean information allowing more than one correct implementation. For example, the way *is_legal* is used in gating the output of the ALU in Figure 1 allows more than one correct implementation of *Instr_decode*, and therefore information about the gating is an example of a don't care.

The following problems will be discussed in this paper:

1. *How to specify don't cares* . In the case when a design is specified through PLA tables, specification of don't cares has been done in the form of separate "don't care cubes", as done

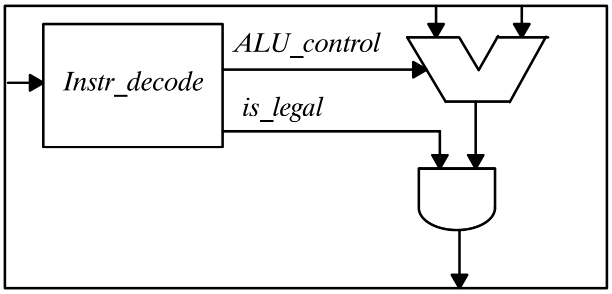

Figure 1: A design containing the partition *Instr_decode*.

for example by Espresso [1]. This approach is limited because it uses a two-level representation and therefore it is not efficient if a large number of cubes is required. Moreover, PLA tables are normally used to represent control logic, but not datapath. This precludes the ability to pass don't-care conditions from the datapath to the controller and back, possibly restricting optimizations.

Normally designs are not given in terms of PLA tables, but rather in the form of high-level and register-transfer-level (RTL) languages, which allow for the concise description of large designs, commonly including control and datapath in the same partition. In this context it becomes important to be able to describe large don't-care sets spanning control and datapath. Section 5 considers three common don't-care specification methods.

2. *How to derive implicit don't cares from the input description.* As the abstraction level of descriptions moves towards RT and behavioral levels, the design space grows; consequently it is expected that the don't-care set also grows. Therefore, it is important to be able to extract don't-care conditions during RTL and high-level synthesis [2, 3, 4]. The Become system [2] is able to derive limited don't-care information based on *If-Then-Else* or *Case* statements. In [3] behavioral don't cares related to registers, multiplexers and ALUs were derived by high-level synthesis. In [4], don't cares were used to minimize the dependency of the control logic on late input signals in order to minimize delay. In all these approaches, the don't-care set was represented as cubes (two-level representation) which were passed to Espresso, Nova [5] or Mustang [6] for logic optimization and state encoding.

   Section 5.3.2 deals with don't cares implied by the description and the semantics of the language and a subset of those created by high-level synthesis. Namely, this paper looks at don't cares implied by the use of multi-bit variables and unreachable states created by scheduling. As an example of a don't care related to multi-bit variables, consider the case where an input is declared as an integer ranging from 0 to 4. This input requires 3 bits, however, by definition it will not assume all values representable in 3 bits. Thus, values 5, 6 and 7 can be used as don't-care conditions. There are other types of behavioral don't cares created by high-level synthesis, which have been presented in [3] and are not the subject of this paper.

3. *How to define the meaning of don't care.* The design language needs to define the behavior of a *correct* implementation. A rigorous definition of the term "don't care" is important because on one hand there are many different interpretations of that word, and on the other

it is essential that the various tools in a methodology use the same interpretation. Section 3 contains two existing formalism (partial functions and Boolean relations) as well others used in the rest of the paper.

4. *How to verify the design specification in the presence of don't cares .* Throughout the paper it is assumed that a design is divided into several partitions, and there is some don't-care information associated with both the design and each partition. While synthesis is performed one partition at a time, design verification (usually simulation-based) is performed on the whole design and must include verifying the don't care information. This paper is independent of how design verification is performed, but it is assumed that design specification is correct. In other words, correctness of an implementation is always defined with respect to its specification, whose correctness is assumed.

5. *How to represent the don't cares in an efficient way for synthesis.* Conciseness of the representation becomes increasingly more important because interactions between synthesized partitions increase as designs become more complex. Existing representations include cubes [1] and BDDs [7], both of which may exhibit exponential growth in relation to an equivalent gate network. The representation used in this paper (Section 4) uses gate networks, so as to avoid restrictions on the size of don't cares . Each of the methods in Section 5 describes how the explicit and implicit don't-care conditions are translated into a network representation suitable for logic synthesis.

6. *Given a don't-care representation, how to take advantage of it during logic synthesis.* This issue has been the subject of most of the literature on don't cares [1, 8, 9, 10, 11] and is closely related to the above issue of don't-care representation. The approach in this paper (Section 6) relies on a test generator, as that is designed to operate on a gate network directly.

7. *When correct implementations of all the partitions are put together, will the result be a correct implementation of the whole design?* In the absence of any don't cares, correctness of the whole design is ensured by correctness of each partition. This desirable property is harder to satisfy in the presence of don't cares, and conditions for achieving it are given for each of the three specification methods in Section 5.

To clarify the scope of this paper the following meanings of "don't care" will not be discussed because they do not imply any choices in implementation.
– An input is sometimes called a don't care if a function does not depend on it. That is a property of a function without introducing any choices.
– For the same reason a don't care literal of a cube is not the subject of this paper.
– Some design languages allow expressions of the form `if (a[0 to 2] = "0-1")` , where '-' is called a don't care. Such a don't care is merely a syntactic convenience, which does not introduce any choices of implementation; it is used to remove `a[1]` from the test. However, if '-' can be assigned to variables and propagated through the design then it will be treated in Section 5.2.
- The expression "don't care" sometimes denotes information given to a static timing analyzer directing it to ignore certain false timing paths. While false timing paths will be a subject of this paper, it will not be referring to any designer directive identifying specific paths.
- So called "internal" or "implicit" don't cares [8] are an important issue for synthesis systems based on PLA optimization [12], and they can be viewed as implying some choices in implementing

a PLA. Since the system in this paper is not based on PLA optimization, it will not consider internal don't cares. Only external don't cares will be treated. The term "external don't cares" will not be used because it is commonly used in a context excluding Boolean relations [11], whereas the use of the word "don't care" in this paper will include Boolean relations.

This paper deals with combinational circuits and combinational don't cares only. Sequential don't cares [13] are not considered. Therefore the terms primary inputs and outputs will include outputs and inputs of memory elements.

2 Terminology

$\mathcal{B}$ denotes the set $\{0,1\}$. However, this paper will also consider multi-valued logic (e.g., non-Boolean 'X', '-', etc.). The letters s, t will denote values, while $\overline{s}, \overline{t}$ will denote vectors of values.

Definition 1 *:*
An $N \times M$ **relation** *F on a set S is a subset of $S^N \times S^M$. ($F(\overline{s})$ denotes the set of all output vectors related to the input vector $\overline{s} \epsilon S^N$).*
A **function** *F (written $F : S^N \rightarrow S^M$) is a relation such that for all $\overline{s}$, $F(\overline{s})$ has at most 1 element.*
A function is called **total** *if $F(\overline{s})$ has exactly 1 element for each $\overline{s}$; otherwise F is partial.*
A **projection** *of a function $F : S^N \rightarrow S^M$ on the i-th output is a function $f : S^N \rightarrow S$ where $f(\overline{s})$ is the value of the i-th coordinate of the M-dimensional vector $F(\overline{s})$.*
A function or relation will be called Boolean iff it is on the set $\mathcal{B}$.

The term "network of gates" will be used throughout the paper. Informally a network is an interconnection of nodes, each representing a function of several inputs and several outputs. Networks will be used in several contexts. A "partition" (e.g., the logic representing the inside of the block *Instr_decode* in Figure 1) will refer to a network being synthesized. A "design" (everything in Figure 1) will denote the amount of logic being verified (simulated). A design may be too large to be synthesized at once and therefore it is divided into several partitions. Formally "partition" and "design" are both networks; these terms will be used selectively to guide the reader to the right context.

Definition 2 *:*
A **network** *is a quintuple – a set of* **nodes***, a set of* **pins***, a set of* **edges***, a sequence of nodes called* **primary inputs***, and a sequence of nodes called* **primary outputs***, all satisfying the following constraints:*
- For each node there is a sequence of **input pins** *and a sequence of* **output pins***. (Either sequence may be empty.)*
- Each edge has a **source pin** *and a* **sink pin***. The source pin must be an output pin of some node and the sink pin must be an input pin of some node (possibly the same node).*
- Each input pin is the sink pin of exactly one edge, called the **incoming edge***, and each output pin is the source pin of one or more edges, called the* **outgoing edges***.*
- The sequence of primary inputs consists of nodes with no input pins and one output pin. The sequence of primary outputs consists of nodes with no output pins and one input pin.

Figure 3 is an example of a network with six nodes, four edges, two primary inputs named a and b and two primary outputs named c and d. The shape of two of the nodes suggests that they have the inverter function, although Definition 2 does not talk about nodes having any function. The function of nodes and the interpretation of a network are defined below.

Definition 3 *:*

An **interpretation** *of a network on a set S is an assignment of a (possibly partial) function to each node other than primary inputs and outputs. A node with N input pins and M output pins must be assigned a function $S^N \rightarrow S^M$. An* **interpreted network** *is a pair [network, interpretation]. (Note that a node with no input pins is either a primary input or it is assigned a constant function).*

Definition 4 *:*

A function $f : S^N \rightarrow S$ **depends** *on its first variable iff there exist values $s_1 \epsilon S, s_2 \epsilon S, \overline{t} \epsilon S^{N-1}$, so that $f(s_1, \overline{t})$ and $f(s_2, \overline{t})$ are both defined, and $f(s_1, \overline{t}) \neq f(s_2, \overline{t})$.*

Analogously, dependence can be defined on any input variable other than the first. For a function $f : S^N \rightarrow S^M$, the i-th output depends on the j-th input if f projected on the i-th output depends on the j-th input. If a function is assigned to a node, then an output pin i depends on an input pin j of the node if the function of the i-th output depends on the j-th input.

Definition 5 *:*

For two functions $f_1 : S^N \rightarrow S^M$ and $f_2 : S^N \rightarrow S^M$, f_1 **has no more dependencies** *than f_2, provided whenever the i-th output of f_1 depends on the j-th input then so does the i-th output of f_2.*

As explained later, it will be important that an implementation f has no more dependencies than its specification f_U.

Definition 6 *:*

A **topological order** *of an interpreted network is an assignment of integers to edges with the following properties:*

– Only edges sharing the same source pin may be assigned the same integer.

– If an output pin $OutPin$ depends on an input pin $InPin$ then the integer assigned to any edge attached to $OutPin$ must be higher than the integer assigned to the edge attached to $InPin$.

This paper considers only interpreted networks with a topological order.

Definition 7 *:*

If a network with N primary inputs and M primary outputs is interpreted on a set S of values, and has a topological order then it **computes** *a function $F : S^N \rightarrow S^M$ defined by the following procedure. Let $\overline{s} \epsilon S^N$ be any given input vector. First assign a function to each primary input node – it will be the corresponding constant from the input vector $\overline{s}$. Then compute a value for each edge in topological order. The values computed for primary outputs constitute $F(\overline{s})$. But if during the computation the value of any edge is undefined then $F(\overline{s})$ is also undefined.*

Although not in the scope of this paper, it can be proven that even if a network has several topological orders, all of them give rise to the same function.

Figure 3, Figure 4, Figure 5 are examples of interpreted networks with topological order. Before one can define the function computed by the networks of Figure 4 or Figure 5 the function of P in Figure 3 must be defined. That function is then assigned to the node named P in Figure 4 and Figure 5.

3 Formalism

It is essential that all the tools used in a methodology, such as simulation, high-level synthesis, logic synthesis, implementation verifier, etc, have the same understanding of the design language semantics in general and don't cares in particular. The common way of avoiding misunderstanding is to define a common formalism for representing the semantics.

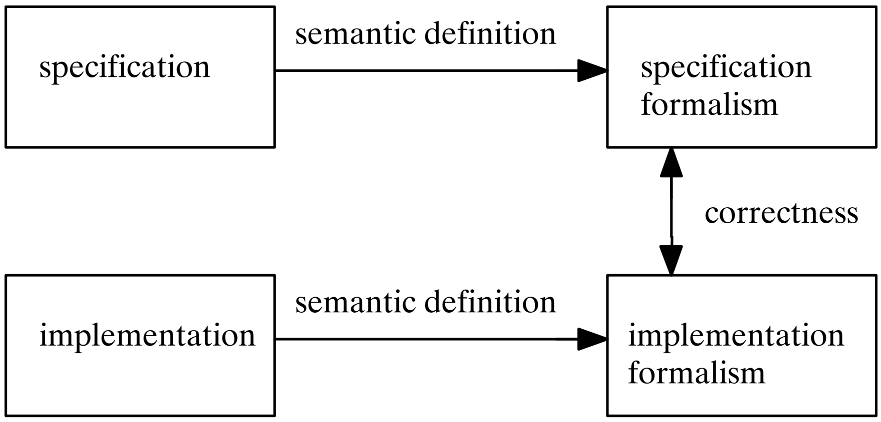

Figure 2: Correctness of implementation is defined by correctness of two formalisms.

Let's consider Figure 2 from the point of view of implementation verification. It has to determine whether a given netlist (e.g. in EDIF) is a correct implementation of a given specification (e.g., in VHDL [14]). The meaning of that task is given by a formalism for both the implementation and the specification, and by some correctness relationship between the two. Normally there is no choice in the formalism for the implementation – it is a total boolean function given by simulation semantics. However, when defining a design language one has the choice of defining both the formalism for the specification as well as the correctness relation between the two formalisms.

In the absence of any don't cares the specification is represented also by a total boolean function and the correctness requirement is simply that the two functions be identical. In the presence of don't cares this simple definition of correctness is not applicable because the very essence of don't cares is supposed to allow more than one correct implementation. This section is concerned with the problem of defining correctness in the presence of don't cares .

Not all definitions of correctness are equally desirable. Any definition of correctness should satisfy the trivial property of declaring an implementation correct whenever it happens to be identical to the specification.

Another important property is "replaceability", which is normally taken for granted in designs without don't cares, and which can be explained as follows. A definition of correctness applies to each partition as well as to the whole design. Suppose that the specification of one partition is replaced by its implementation and suppose that the resulting design with the one partition replaced is correct with respect to the original design. In this case, the partition specification is said to be replaceable [15] by its implementation. In a methodology which infers correctness of the whole design by verifying the implementation of each partition separately, it is essential that correctness of a partition implies replaceability. Section 5 will establish conditions for correctness to imply replaceability in the presence of don't cares.

The most common don't-care formalism is that of a partial function. In the instruction decoder example in Figure 1, the specification would be given by two partial functions – one for *ALU_control* and one for *is_legal*. The don't-care information can be represented by making the *ALU_control* function undefined for illegal instructions.

Definition 8 *:*

A function f is **correct** *with respect to a partial function f_U iff $f_U \subseteq f$.*

Partial functions are inadequate to describe some don't care situations [11], because an implementation is free to output any value for each undefined input pattern. For example, in Figure 3 there is a partition with inputs a, b and outputs c, d. That partition is embedded inside a design in Figure 4 that XORs the outputs c, d. This gives some freedom in the implementation; for example, both inverters can be removed without changing the functionality of the whole design. There is no partial function capable of specifying all the correct replacements for P without also allowing incorrect replacements.

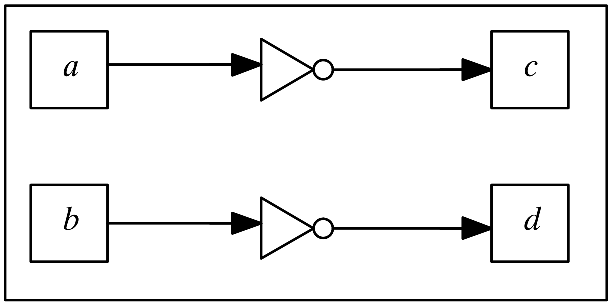

Figure 3: A partition P to be optimized.

To see that, suppose there is a partial function Q that specifies ALL the correct implementations of P. For input (0,0), Q would have to allow the output (1,1) because that is the response of the implementation in Figure 3 on input (0,0). For the same input (0,0), Q would also have to allow the output (0,0) because that is the response of the implementation with both inverters removed, which is also correct. The existence of two correct output patterns force Q to be undefined on (0,0). But that allows an implementation generating output (1,0) on input (0,0), which is not correct.

For this reason Boolean relations have been introduced [11]. Using a Boolean relation R, all the correct implementations can be exactly specified as follows:

$R(0,0) = \{(0,0), (1,1)\}$
$R(0,1) = \{(0,1), (1,0)\}$
$R(1,0) = \{(0,1), (1,0)\}$
$R(1,1) = \{(0,0), (1,1)\}$

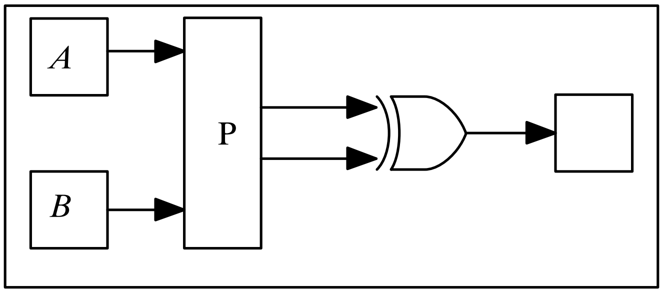

Figure 4: Partial functions are not adequate to describe all correct implementations of P.

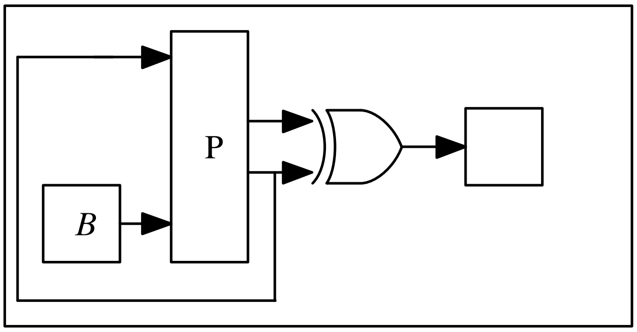

Figure 5: Boolean relations are not adequate to describe all correct implementations of P.

Definition 9 *:*
A function f is **correct** *with respect to a Boolean relation f_U iff $f \subseteq f_U$.*

Boolean relations are also inadequate to describe some don't-care situations because an implementation function is free to chose the result on one input pattern independently of its choices for other input patterns. For example, suppose that the partition of Figure 3 is embedded in the network of Figure 5. This gives some freedom in the implementation; for example, the partition P could be implemented by a constant function always outputting (1,0) independently of the inputs a,b. This is because whether P is implemented using the function in Figure 3 or using the constant function, the design in Figure 5 still outputs the constant 1. There is no Boolean relation capable of specifying all the correct implementations, without also allowing incorrect ones.

To see that, suppose there is a Boolean relation R that specifies ALL the correct implementation functions of P. R(0,0) must include (1,1) because (1,1) is the response of the implementation in Figure 3 on input (0,0). For all inputs, R must include (1,0) because that is the response of the constant function discussed in the previous paragraph. As a result, R would have to allow function f to output (1,1) on input (0,0), and output (1,0) on the other three input patterns. If this f is assigned to node P in Figure 5, then for input $B = 0$ the new design will oscillate (i.e., it will not have a topological order), whereas the original design did not oscillate. Thus the requirement that R allow ALL correct implementations forces it to allow an incorrect one as well.

The formulation in this paper is not limited to partial functions or Boolean relations. The meaning of a don't-care situation will be defined simply by giving the criterion for an implementation to be considered correct. Such a criterion defines a set of functions, which is the specification formalism in Figure 2. Replaceability will be defined based on a given notion of correctness.

The examples in Figures 4 and 5 consisted of a network *Design* with a distinguished node P. Each node of *Design* computed some function; let f_U (Figure 3) be the function of P. The assignment of a function to each node of *Design* determined a function F_U computed by the whole *Design*. For the purpose of illustration, F_U is considered the only correct implementation function of the whole design. If P is given a different function f, which causes the whole design to compute a (possibly different) function F, then replaceability implies that f can replace f_U if $F = F_U$.

Replaceability is important when a design is divided into several partitions synthesized separately. After synthesizing all the partitions, their implementations are combined into an implementation of the whole design. There are several ways of making sure that this resulting implementation behaves as specified. One approach is to simulate the implementation, but that is less efficient than simulating the unsynthesized design specification. Another approach is to simulate the design specification before synthesis, and then formally verify that the implementation is correct with respect to the specification. But designs tend to be too large for existing verifiers to handle. Moreover, it is difficult to find the possible causes of implementation errors using either approach.

Therefore a common practice is to eliminate completely the verification of the whole design implementation. Instead, the implementation of each partition is verified, and if they are all correct, it is assumed that their combination will result in a correct design. Therefore with this approach it is essential to make sure that correctness of all the partitions implies correctness of the whole design, i.e., that correctness implies replaceability. However, as it will be shown, it is not easy to devise a design language that allows don't cares and whose definition of correctness implies replaceability.

Definition 10 *:*

An **embedding** *of a function $f : S^N \rightarrow S^M$ is a triple $[Design, Partition, I]$, where $Design$ is a network containing a node $Partition$ with N input pins and M output pins. I is an interpretation of $Design$ with two properties:*

- The node $Partition$ must be assigned the function f.

- The interpreted network [Design, I] must have a topological order.

Definition 11 *:*

Let $f : S^N \rightarrow S^M$ and $f_U : S^N \rightarrow S^M$ be two functions, and let $[Design, Partition, I_U]$ be an embedding of f_U. Let I be the same interpretation as I_U except that $Partition$ is given the function f instead of f_U. Then f is a **safe replacement** *for f_U in the embedding $[Design, Partition, I_U]$ iff*

– any topological order of $[Design, I_U]$ is also a topological order of $[Design, I]$, and

– if $[Design, I_U]$ has a topological order then the function computed by $[Design, I]$ is correct with respect to the function computed by $[Design, I_U]$.

The term "correct" in the above definition will be defined for each kind of don't cares in Section 5, which will then give a meaning to "safe replaceability" particular to that kind of don't care.

Definition 12 *:*

A function f is a **universal replacement** *for f_U iff f is a safe replacement for f_U in any embedding of f_U.*

Safe replaceability is dependent on a particular design. In contrast, universal replaceability is defined independently of any design. Once an implementation f is a universal replacement for a specification f_U then f may replace f_U in any design where f_U might be embedded. That means, for a new design one has to perform only design verification (using f_U). Once that is successful f may be used in place of f_U without any need for more verification.

4 Representing Don't Cares

In order for logic synthesis to take advantage of don't cares they have to be represented in a form that is concise and suitable for logic optimization. As stated in the introduction, this work uses gate networks (as opposed to cubes or BDDs) because of their expressive power and conciseness. Section 5 shows how these networks are generated for each don't-care specification method, and Section 6 explains how they are used during optimization.

For each don't-care specification method the designer input is formalized as a function f_U, which might be total or partial, Boolean or multi-valued. From this function f_U two networks are generated – **care network** and **don't-care network** (see Figure 6). The care network computes some total boolean function f_0, which is one correct implementation, and can be chosen arbitrarily from the set of all correct implementation functions. The don't-care network is used to specify how the care network can be changed from computing f_0 into computing some other function f, which is also correct. The two networks together will be called the **merged network**.

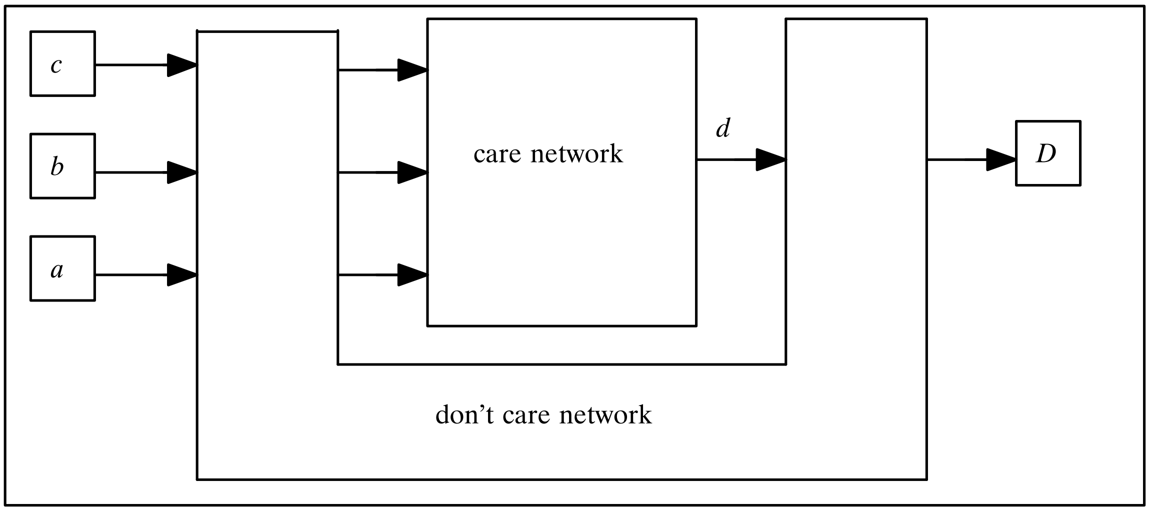

Figure 6: The most general configuration of a merged network.

For illustration, consider the following logic expression with primary inputs `a, b, c` and primary output `d`. The designer states that

```
d = a  and  b  and  c;
```

but he does not care about the value of d whenever

```
a = b = 1
```

The care network also has primary inputs `a, b, c`, primary output `d`, and it is built so as to compute the function f_0 `= a and b and c`. The don't-care network has primary inputs `a, b, c, d`, primary output `D` and computes

```
if (a and b) then D = 0; else D = d;
```

The assignment of 0 to `D` is arbitrary and could be replaced by any other constant. The important thing is that the propagation of `d` to `D` is gated by the care condition. The don't-care network is always constructed so that replacing f_0 by any correct function f will preserve the functionality of the merged network. Conversely, only a correct f will be able to replace f_0 without changing the functionality of the merged network.

In the previous example, the primary inputs `a, b, c` into the care network do not need to be processed by the don't care network. But in general the don't-care network may be quite arbitrary as long as the merged network of Figure 6 has a topological order.

5 Specifying Don't Cares

Three kinds of don't cares will be considered. For each kind of don't care, the following will be discussed: 1) how to specify the don't cares or derive them from a design language, 2) the meaning of the don't care in terms of correctness of an implementation, 3) the conditions for safe and universal replaceability, and 4) how to construct the care and don't care networks for synthesis.

5.1 The Whole Design Provided To Synthesis

In this scenario, synthesis is allowed to examine the whole design and use that information while optimizing an individual partition. Consider the instruction decoder in Figure 1. The designer provides a network with one possible implementation of *Instr_decode* (f_U) and allows synthesis to look at the rest of the design embedding *Instr_decode*. The fact that the ALU outputs are used only in case of legal instructions is sufficient for synthesis to perform optimizations on *Instr_decode* without any other don't care information from the designer.

The care and don't care networks (Section 4) are constructed very simply. The care network is the partition computing f_U itself (thus $f_0 = f_U$), and the don't care network is the rest of the design.

The formalism (Section 3) for representing the specification consists of a total boolean function f_U (for the partition being synthesized) plus a particular embedding of f_U. Correctness is defined with respect to this embedding.

Definition 13 *:*
A function $f : \mathcal{B}^N \rightarrow \mathcal{B}^M$ *is* **correct** *with respect to a function* $f_U : \mathcal{B}^N \rightarrow \mathcal{B}^M$ *in a given*

embedding [$Design, Partition, I_U$] of f_U iff
– any topological order of [$Design, I_U$] is also a topological order of [$Design, I$], and
– if [$Design, I_U$] has a topological order then the function computed by [$Design, I$] is identical to the function computed by [$Design, I_U$].
(As in Definition 11, I is obtained from I_U by assigning f to Partition.)

Correctness has been defined exactly to imply safe replaceability (where correctness of a whole design requires identical functionality). However, since it is defined relative to a specific embedding, it cannot imply universal replaceability.

While this method of specification is both convenient and powerful, it has several methodology problems. First, synthesis will be slowed down by examining the entire design. Secondly, verification of the synthesized partition must take into account the don't care information, that is, the whole design. That may be too large of a network for existing verifiers to handle.

Another problem is the lack of universal replaceability. Partitions cannot be changed independently of each other – a change in one partition requires resynthesis of all others. Therefore this don't care specification method is effective only when safe replaceability is sufficient, and verification problems can be solved.

The reason for the lack of universal replaceability is the fact that don't care information is represented externally to the partition being synthesized. Therefore the next two methods will have the don't care information as integral part of the partition.

5.2 Non-Boolean Values

In some design languages it is suggested that don't cares be expressed using special non-boolean values, e.g. X. In the instruction decoder example (Figure 1), *ALU_control* would be assigned X in case of an illegal instruction.

A simple example of a partition using an X value is the following:

```
Partition1:     if (a = 1) then b = 0; else b = X;
```

with primary input a and primary output b. When $a = 1$ then the designer "cares" that b be 0, but if $a = 0$ then the designer does not care what b is.

The semantics of such multi-valued logic is normally defined by a simulator. In this paper a so called "conservative" simulator is assumed, as is used for instance in VHDL, where simulation semantics is defined for a network, whose nodes have been assigned VHDL operators. The semantics is given by a simulation table for each operator. Therefore a specification is assumed to be given by a network interpreted on a set S of values. Following Definition 7, a partition then computes some function $f_U : S^N \rightarrow S^M$. The set S is assumed to contain the boolean values 0,1, among others. In the above example of *Partition1*, $S = \{0,1,\text{X}\}$ and the network for f_U is obtained in the usual way from the textual representation.

To our knowledge there is no general agreement on how to define correctness of an implementation (which is a boolean function) with respect to a multi-valued specification. In this paper correctness will be defined by assuming a given rule as to which boolean values can replace non-boolean values appearing on primary outputs. For example, consider the following values $S = \{0, 1, \text{L}, \text{H}, \text{X}\}$; one might specify that L appearing on primary output must be implemented by 0, H must be implemented by a 1 and X may be implemented by either 0 or 1. For notational convenience this information will be assumed given as a partial order $\prec$ (read "less defined than") on the set S of values. In this example, the partial order would be (L $\prec$ 0, H $\prec$ 1, X $\prec$ 0, X $\prec$ 1). Note that an implementation is not allowed to generate 1 where specification generated 0 because the partial order does not contain 0 $\prec$ 1.

The order $\prec$ can be extended in the usual way to a vector of values component-wise. Then in terms of Section 3 the meaning of non-boolean values is as follows.

Definition 14 *:*
A function $f : S^N \rightarrow S^M$ is **correct** *with respect to a function $f_U : S^N \rightarrow S^M$ iff for any $\overline{s} \epsilon S^N$, $f_U(\overline{s}) \preceq f(\overline{s})$.*

With the above partial order there are two correct implementations of *Partition1* (in addition to *Partition1* itself)

```
if (a = 1) then b = 0; else b = 0; or
if (a = 1) then b = 0; else b = 1;
```

To illustrate the problem of replaceability consider

```
Partition2: { if (b = 0) then c = 0; else c = 1;
              if (b = 1) then d = 0; else d = 1;
            }
```

with primary input b and primary outputs c, d. Since only Boolean values are involved, *Partition2* has only one correct implementation, namely itself.

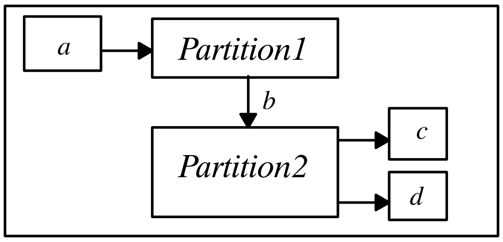

Figure 7: Design with no correct boolean implementation

Consider the design in Figure 7 consisting of *Partition1* feeding *Partition2*. The design has primary input a and primary outputs c, d. In computing the function F_U of the whole design, subject to VHDL semantics, both expressions "(X = 0)" and "(X = 1)" evaluate to `false`. Function F_U can then be expressed by

```
if (a = 1) then {c = 0; d = 1;}
          else {c = 1; d = 1;}
```

Suppose that *Partition1* is replaced by its first correct implementation. Then the new design computes the function

```
if (a = 1) then {c = 0; d = 1;}
           else {c = 0; d = 1;}
```

And if *Partition1* is replaced by its second correct implementation then the design computes the function

```
if (a = 1) then {c = 0; d = 1;}
           else {c = 1; d = 0;}
```

It can be seen that by replacing *Partition1* by either of the two correct implementations an incorrect design results. Thus neither implementation is a safe or universal replacement for Partition1. This is a significant problem because with existing design languages it is not easy for a designer to determine whether there even exists an implementation of his specification that would yield a correct design. Therefore, it becomes important to develop the necessary and sufficient conditions for correctness to imply universal replaceability.

Definition 15 :
A set $\mathcal{G}$ of functions is **closed** *iff it has the following property. Consider any network and any interpretation using functions from $\mathcal{G}$ so that there is a topological order. Then the function computed by the interpreted network is also in $\mathcal{G}$.*

A given closed set of functions, called *AllF*, is assumed. *AllF* is assumed to contain (among others) all constant functions. Informally, *AllF* consists of all the primitive (multi-valued) operators of a design language (AND, OR, NOT, COMPARE, etc.) plus all the functions that can be built out of them. For example, if a design language allows only the values {0,1} and only the primitive operators AND, OR then *AllF* consists of all the positive functions. If a design language allows the values {0, 1, X} and the primitive operators AND, OR, NOT then *AllF* consists of all the boolean functions plus some ternary functions; exactly which ternary functions would be included in *AllF* depends on how the primitive operators are defined on X.

Definition 16 :
A function $g : S^P \rightarrow S^Q$ is **monotonic on a set** $T \subseteq S$ *iff $g(\overline{s_U}) \preceq g(\overline{s})$ for any $\overline{s_U}$, $\overline{s}$ satisfying the following two conditions:*
a) $\overline{s_U} \preceq \overline{s}$
b) $\overline{s_U} \in T^P$ and $\overline{s} \in T^P$

For example, assume that a design language defines X $\prec$ 0, X $\prec$ 1 and provides the operators NAND and COMPARE. The latter is defined by COMPARE(0,0) = COMPARE(1,1) = 1, and COMPARE(0,1) = COMPARE(1,0) = 0. Suppose that the two primitive operators are extended to the value X by outputting X whenever any input is X. Then the two primitive operators are monotonic and hence all functions in *AllF* are monotonic. In contrast, suppose that in extending COMPARE from boolean logic to ternary logic it was required to always return a boolean value (as VHDL requires). Then it is not possible to make COMPARE monotonic. To see that, consider the question of how to define COMPARE(X,0). Let g(s) = COMPARE(s,0). If g(X) = 0 then the monotonicity property is violated because X$\preceq$ 0, but g(X)$\not\preceq$g(0). A similar violation occurs if g(X) = 1.

It is important for *AllF* to be monotonic because as the next two theorems show monotonicity is a necessary and sufficient condition for correctness to imply replaceability.

Theorem 1 :

Assume that every function in *AllF* is monotonic on a set S and let $f : S^N \to S^M$ and $f_U : S^N \to S^M$ be two functions in *AllF*. Then f is a universal replacement for f_U iff the following are true:
1) f is correct with respect to f_U.
2) f has no more dependencies than f_U.

Condition 2) is needed to prevent the following situation. A function f_U is defined to be independent of its input variable a, and its output b is always the constant X. An implementation f makes $b = a$. The implementation is correct, but if the partition were embedded in a design where the output b is connected to the input a through an inverter then the design would oscillate (i.e., would not have a topological order).

Proof:
First suppose that f is a universal replacement for f_U.

To prove condition 1) consider an embedding $[Design, Partition, I_U]$ of f_U, where the node $Partition$ receives its inputs directly from N primary inputs, and the outputs of $Partition$ go directly to M primary outputs. Then the function of $[Design, I_U]$ is identical to f_U. Let I be same as I_U, except that I assigns f to $Partition$. Then the function of $[Design, I]$ is identical to f. Thus the correctness of f with respect to f_U follows directly from Definition 12 of universal replaceability.

To prove condition 2) suppose that the i-th output of f_U does not depend on the j-th input; it needs to be proven that the i-th output of f does not depend on the j-th input either. Consider an embedding $[Design, Partition, I_U]$ of f_U. Besides the node $Partition$, the network $Design$ has $N-1$ primary inputs and M primary outputs. Each output pin of $Partition$ is connected to a primary output. Each input pin of $Partition$ is connected to a primary input, except for the j-th input pin. The j-th input pin of $Partition$ is connected directly to the i-th output pin. Since the i-th output of f_U does not depend on the j-th input, the interpreted network $[Design, I_U]$ does have a topological order. By the assumption that f is universally correct with respect to f_U, that topological order must be preserved in the interpretation I that assigns f to $Partition$. Therefore the i-th output of f cannot depend on the j-th input.

Supposing that conditions 1) and 2) hold, it can now be proven that f is universally correct with respect to f_U.

Consider any embedding $[Design, Partition, I_U]$ of f_U and let the interpretation I be as given in Definition 11. Let F_U be the function computed by $[Design, I_U]$. Definition 11 requires that $[Design, I]$ have the same topological order as $[Design, I_U]$ and that the function F computed by $[Design, I]$ be correct with respect to F_U.

Condition 2) of the theorem ensures that any topological order for $[Design, I_U]$ can be also used for $[Design, I]$.

In order to show that F is correct with respect to F_U one has to prove that $F_U(\overline{z}) \preceq F(\overline{z})$ for any $\overline{z}$. By induction on the topological order, the following can be proven for any edge V:

Let v_U be the value of the edge V in $[Design, I_U]$ and v be the value of the edge V in $[Design, I]$; then

$$\overline{v_U} \preceq \overline{v} \tag{1}$$

The proof of the theorem will be completed by proving (1) because $F_U(\overline{z}) \preceq F(\overline{z})$ follows from

(1) applied to primary outputs.

Case i: The edge V is not attached to an output pin of the node *Partition*.
Let g be the function of V in terms of output pins of *Partition*. That means, g is the function computed by the network obtained from *Design* as follows: Delete the node *Partition*, create a primary input node for each output pin of *Partition*, create a primary output node as a sink for V, delete all other primary outputs of *Design* and all nodes feeding only those primary outputs, change all primary inputs into constant nodes – their interpretation is given by the input vector $\overline{z}$.

The function g, being computed by an interpreted network, must be in *AllF*, hence it is monotonic. Consider only those P outputs of *Partition* on which g actually depends and let $\overline{s_U}$, $\overline{s}$ be the values on those P outputs in $[Design, I_U]$ and $[Design, I]$ respectively. Since those P outputs of *Partition* must precede V in the topological order, their values satisfy (1), i.e., $\overline{s_U} \preceq \overline{s}$. By premise of the theorem g is monotonic, hence $g(\overline{s_U}) \preceq g(\overline{s})$, which proves (1).

Case ii: The edge V is attached to an output pin of the node *Partition*. Let $\overline{W}$ be all the input edges of *Partition* on which V depends. Let h be the projection of f on the output V; that is, h is the function that in $[Design, I]$ computes V with inputs $\overline{W}$. The function h_U is defined analogously in $[Design, I_U]$. Let $\overline{w_U}$ be the values on $\overline{W}$ in $[Design, I_U]$ and let $\overline{w}$ be the values on $\overline{W}$ in $[Design, I]$. By condition 1) of the theorem

$$h_U(\overline{w}) \preceq h(\overline{w}). \tag{2}$$

Since all the edges $\overline{W}$ precede V in the topological order, they satisfy the induction hypothesis, i.e.,

$$\overline{w_U} \preceq \overline{w}. \tag{3}$$

By monotonicity of h_U

$$h_U(\overline{w_U}) \preceq h_U(\overline{w}). \tag{4}$$

By composition of (2) and (4), $h_U(\overline{w_U}) \preceq h(\overline{w})$, i.e., $v_U \preceq v$.

QED

To achieve universal replaceability synthesis must ensure condition 1) of Theorem 1 as usual. Synthesis must also ensure condition 2), which is normally not a problem. The design language definition must guarantee monotonicity of all operators. The monotonicity condition is actually necessary as stated below.

Theorem 2 :

Suppose that for every pair of functions $f : S^N \to S^M$ and $f_U : S^N \to S^M$ satisfying conditions 1) and 2) of Theorem 1, f is universally correct with respect to f_U. Then every function in *AllF* is monotonic on the set S.

Proof: The proof requires showing that $g(\overline{s_U}) \preceq g(\overline{s})$, for any function $g : S^P \to S^Q$ in *AllF*, and any values $\overline{s_U}$, $\overline{s}$ satisfying Definition 16. Consider a network *Design* with no primary inputs, with Q primary outputs, and two other nodes. Node *Partition* has no input pins and P output pins. Node G has P input pins and Q output pins. The inputs of G are connected directly to the outputs of *Partition* and the outputs of G are connected directly to primary outputs.

Let f_U be the function with no inputs and constant output $\overline{s_U}$, and f be the function with no inputs and constant output $\overline{s}$. Please note that the two functions satisfy conditions 1), 2) of Theorem 1. Let I be the interpretation assigning f to *Partition* and g to G. Let I_U be the interpretation assigning f_U to *Partition* and g to G. Since by premise of the theorem f is universally correct with respect to f_U, then $g_U(\overline{s_U}) \preceq g(\overline{s})$.

QED

Having considered what is needed for design correctness in general, one can now apply those concepts to don't cares.

Definition 17 *:*

Let T be a set of values on which all the functions in AllF *are monotonic.*
If $t \in T, t \preceq 0, t \not\preceq 1$ then t is called a **synthesis care 0 value**.
If $t \in T, t \not\preceq 0, t \preceq 1$ then t is called a **synthesis care 1 value**.
If $t \in T, t \preceq 0, t \preceq 1$ then t is called a **synthesis don't care value**.
Otherwise t is called a **synthesis error value**.

A synthesis care 0 value must be implemented as 0 and a synthesis care 1 value must be implemented as 1. A synthesis don't care value can be implemented as either 0 or 1. There are two kinds of synthesis error values. One kind are values $t \in T$ for which neither $t \preceq 0$ nor $t \preceq 1$. If a specification ever allows such a value t on a partition output then the specification has no correct implementation. The other kind of synthesis error values are those that are not in T. For those, there may be a correct implementation of the partition, but not of the whole design. (It will be shown how to generate an implementation in Section 5.2.1.)

Unfortunately, current design languages, VHDL and Verilog, do not offer monotonicity on any set T larger than {0,1}. The main reason are statements like "if (a = b) ...", which existing design languages require to select either the then-clause or the else-clause. As discussed above, as long as comparison for equality is required to yield a boolean value even for non-boolean inputs, monotonicity cannot be achieved.

Theorem 1 is contingent upon Definition 14 of correctness. Replaceability could be ensured if correctness were defined differently, for example, as follows. For every input and output port of the specification f_U, a correct implementation would have to have several binary ports. These binary ports would encode the values that can pass the single port of the multi-valued function f_U. This way correctness would imply universal replaceability, but one would not be able to express any don't cares. Implementations would no longer be free to replace X with 0 or 1. Moreover, the expense of encoding the non-boolean values would be prohibitive.

Consequently, no design methodology can have all of the following desirable properties:

1. Use of existing design languages (e.g., VHDL, Verilog)
2. Guarantee of design correctness
3. Separate synthesis of partitions
4. Separate verification of partitions
5. Existence of synthesis don't care values

Some methodologies give up property 4, forcing the simulation of the whole design after implementation. Some methodologies (such as the one used in this paper) give up requirement 5 and do not use X as a synthesis don't-care value. (Instead, it uses other ways of specifying don't cares, as shown in Section 5.3).

5.2.1 Care and Don't-Care Networks

Regardless of how a methodology deals with the replaceability problem, there is a need to generate implementations that are correct with respect to multi-valued specifications. This section explains how to construct the care and don't care networks, which yield all correct implementations for a given multi-valued function $f_U : \mathcal{S}^N \rightarrow \mathcal{S}^M$. The care network computes a function $f_0 : \mathcal{B}^N \rightarrow \mathcal{B}^M$, which is correct with respect to f_U in the boolean domain only. According to Definition 14, a correct implementation has to accept non-boolean values because it considers the situation of embedding an implementation together with other partitions generating non-boolean values. However, in practice all partitions are implemented, such that when given Boolean inputs they produce Boolean outputs only. Therefore it is sufficient for the implementations to be correct on Boolean values only.

The construction of the care and don't care networks will be explained using the example of *Partition1* and *Partition2* of Section 5.2. They have to be combined into one larger partition (see Figure 8), which is to be synthesized as a whole, because as showed in Section 5.2 it is not possible to synthesize them separately and obtain a correct design.

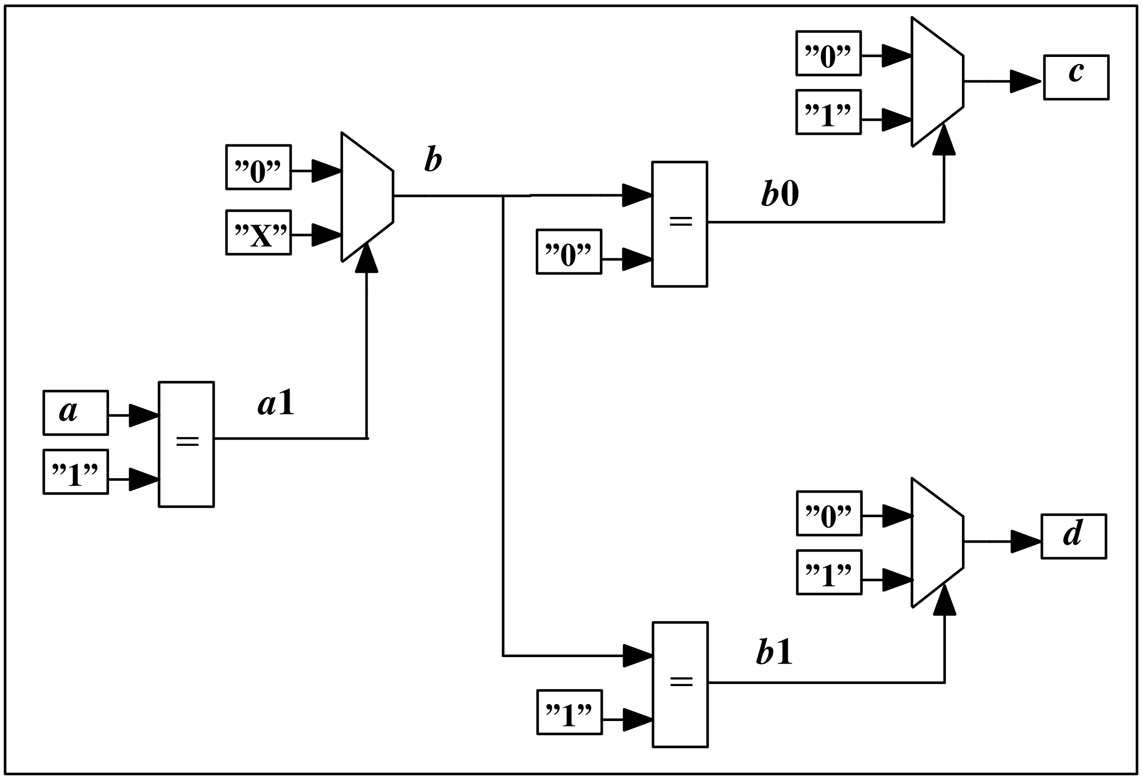

Figure 8: Network for *Partition1* and *Partition2* merged.

The network computing f_U is the interpreted network *Spec*, as shown in Figure 8. The care and don't care networks, which together form the merged network (Figure 9), are generated as follows. The merged network should have as many primary inputs and outputs as *Spec*. Each primary output of the merged network is the result of ANDing an output of the care network with a condition expressing that the output is a synthesis care value.

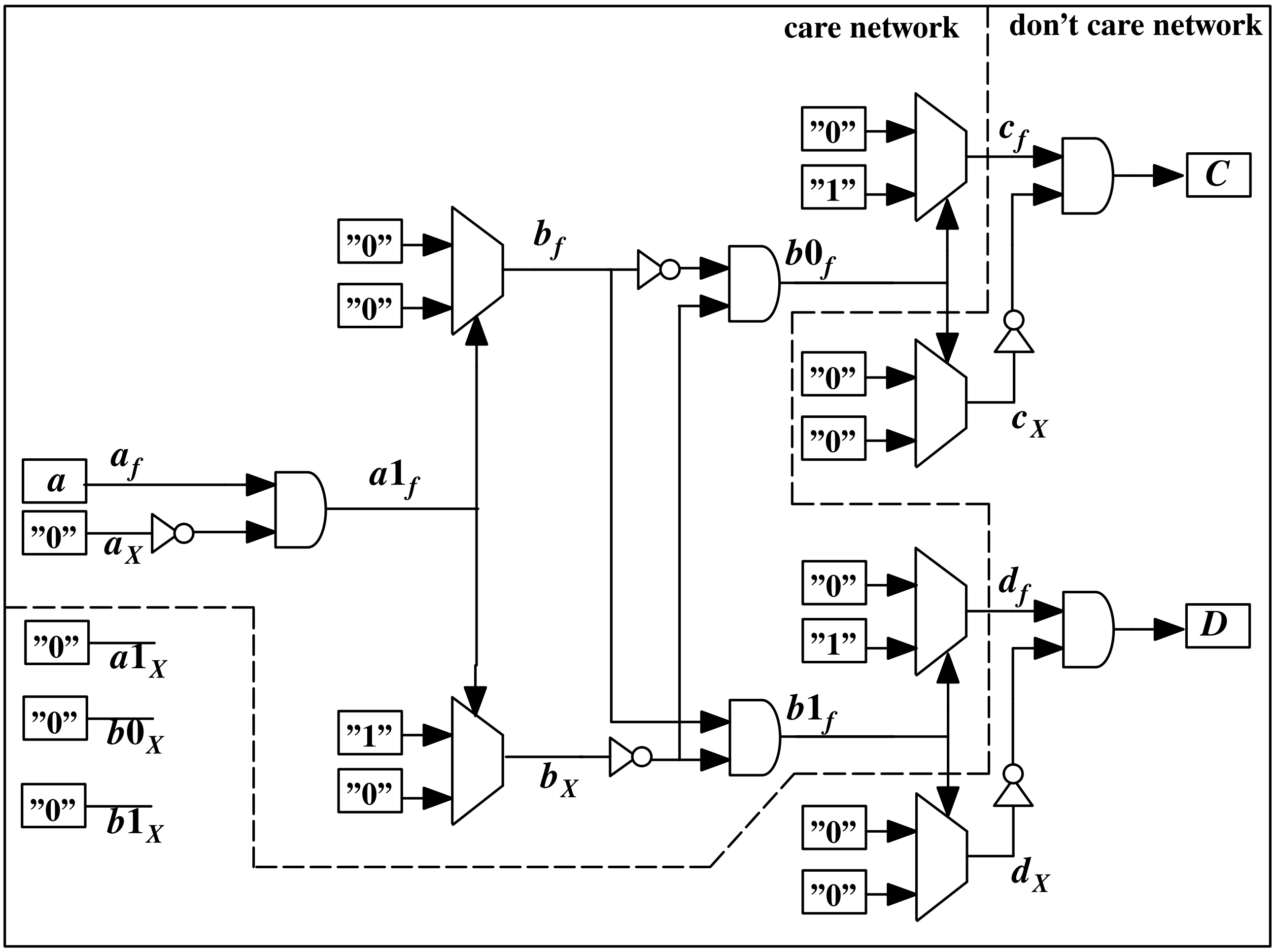

Figure 9: Implementation of *Partition1* and *Partition2* together with the don't care network.

An edge V in *Spec* may have more than two values. These values need to be encoded in the merged network. In the case of *Spec* there are three values {0,1,X} which are encoded as explained below (other encodings are also possible).

For each edge V in *Spec* two edges V_f and V_X are generated in the merged network (Figure 9). The logic feeding them will be built so that
$V_X = 1$ whenever $V = X$,
$V_X = 0$ and $V_f = 0$ whenever $V = 0$,
$V_X = 0$ and $V_f = 1$ whenever $V = 1$.

The construction proceeds according to the topological order of *Spec*.

If V is a primary input then V_X is implemented as the constant 0 because only boolean values can propagate between implemented partitions. The edge V_f is the output edge of the corresponding primary input in the merged network being built.
If V is the constant 0 then $V_X = 0$, $V_f = 0$.
If V is the constant 1 then $V_X = 0$, $V_f = 1$.

If V is the constant X then $V_X = 1$, $V_f = 0$ or 1. (This nondeterminism in the implementation of the constant X will be discussed later.)

If V is not a primary input or a constant then V_X and V_f are implemented based on the function of V's source node using the already existing implementations of the source node's inputs. That will be illustrated later using the example of *Partition1* and *Partition2*.

If V is a primary output then the corresponding primary output of the care network and the primary output of the merged network are defined as follows. The primary output of the care network must evaluate to 1 whenever V evaluates to a synthesis care 1 value, and must evaluate to 0 whenever V evaluates to a synthesis care 0 value. In the case of $S = \{0,1,\mathrm{X}\}$ the output of the care network is identical to V_f. The primary output of the merged network is the primary output of the care network (V_f) ANDed with the condition that V has a synthesis care value. That is, the primary output of the merged network is equal to AND(V_f, NOT(V_X)).

Having defined the merged network, the care network consists of all the nodes needed for the computation of any of the care network's outputs, and the don't care network consists of the rest. Please note that Figure 9 shows $b0_f$ and $b1_f$ as outputs of the care network; that is a simplification designed to avoid crowding of the figure. In reality the logic computing $b0_f$ and $b1_f$ must be replicated in the don't care network because only c_f and d_f are outputs of the care network.

Next the example of Figure 8 and Figure 9 will be considered in greater detail. VHDL semantics for all the nodes is assumed. In the two figures there are several kinds of nodes:
– input node for a,
– output nodes for c and d,
– constant nodes for "0", "1", and "X",
– AND gates,
– inverters,
– comparators (Comparators take two inputs and generate 0 or 1 depending on whether the two inputs are equal),
– multiplexors (Multiplexors have one control input on the side plus two data inputs, and they output one of the data inputs, namely, if the control input is 0 then the output is the data input closer to the control input).
The output of a comparator and the control input of a multiplexor will always have a Boolean value.

The edges are labeled with variable names; there are three variable names appearing in Figure 8, which do not appear in the textual definition above – $a1, b1, b0$; they are needed because "`if (a = 1)`" is implemented as a comparator and a multiplexor with $a1$ connecting them. The merged network in Figure 9 has also primary outputs C and D; these two outputs must preserve their functionality throughout optimization. The primary outputs of the care network are c_f and d_f and their functionality may change during optimization.

To illustrate the construction consider the comparator output $a1$. It is expected to have the value 1 if a=1 and 0 otherwise. Since $a1$ is always Boolean, $a1_X$ is implemented as the constant 0 in Figure 9. (For purposes of illustration Figure 9 actually shows $a1_X$ although it is not needed and can be removed.) The implementation $a1_f$ is 1 when a is 1 (which implies that a is a Boolean, a_X is 0, and a_f is 1). Similar rules apply to the other two comparators.

As mentioned above, some arbitrary choices were made in constructing the merged network. There are two ways of encoding the constant V=X, namely either by $V_X = 1, V_f = 0$, or by

$V_X = 1, V_f = 1$. Depending on the choice, Figure 9 would change; the second input into the multiplexor defining b_f would be either "0" or "1". (Please note that for both choices the network defines the same function, namely $c_f = \text{not}(a)$, $d_f = 1$.) Some arbitrary choices were also made when defining an output of the care network. An ideal synthesis system should find an optimal implementation regardless how the above choices are made. A heuristic synthesis system may give different implementations depending on those choices, but it may be difficult to predict which choice will result in the best implementation.

5.3 Don't Cares Representing Impossible Conditions

This section discusses a type of don't care which represents conditions that should never happen in the design. These conditions can involve primary inputs (e.g., two inputs are never on at the same time) or latch outputs (e.g., a state is unreachable); both are conditions on inputs into the combinational part. The validity of such don't cares (that is, checking that such conditions never happen) is established during design verification (e.g., simulation). This section presents two types of such don't cares, namely, explicitly specified don't cares using *language assertions* and don't cares extracted by high-level and RT-level synthesis.

In order to define correctness, the specification is represented by a partial function, undefined for inputs causing the don't care condition to be true. **Correctness** is then defined by Definition 8; that is, an implementation is correct if it gives the same outputs as the specification on the care set.

The don't cares of this section are a special case of non-boolean values of Section 5.2. Impossible conditions can be expressed by assigning X to all primary outputs when the don't care condition is true. Therefore Theorem 1 applies here, but becomes simpler because monotonicity is assured by restricting the possible values to boolean values.

Theorem 3 :
Let $f : \mathcal{B}^N \rightarrow \mathcal{B}^M$ and $f_U : \mathcal{B}^N \rightarrow \mathcal{B}^M$ be two partial functions. Then f is a universal replacement for f_U iff
1) f is correct with respect to f_U.
2) f has no more dependencies that f_U.

The don't cares of this section are weaker than those of Section 5.2; for instance, the original example of the instruction decoder (Figure 1) cannot be expressed in terms of impossible conditions. On the other hand, universal replaceability is easy to guarantee.

The care and don't care networks can be built as described in the example of Section 4. Namely, the source description of the design becomes the care network, and the don't care network is built so as to gate the outputs of the care network with the given care conditions.

In practice, this representation is replaced by another one, more efficient for manipulations using a test generator. The don't care network has the same inputs as the care network, but has no outputs. Inside the don't care network there are special don't care boxes whose inputs are the impossible conditions. The construction will be described in greater detail in the following subsections, and Section 6 will describe how the don't-care network is used for logic optimization.

Since the don't cares of this section are the most practical ones from the point of view of an overall methodology, they have been implemented in the HIS [16] and BooleDozer [17] systems, and this implementation will be described in greater detail.

5.3.1 Explicitly Specified Don't Cares Using Language Assertions

The approach proposed in this paper for specifying don't cares explicitly is based on the use of the *ASSERT* statement in VHDL (see Figure 10). The *assert* statement allows the designer to specify Boolean expressions within the VHDL source, which can be checked during simulation. These *assert* expressions, which used to be ignored by synthesis, are actually mapped onto a network and used for the purposes of optimization and verification. The expression given by an *assert* statement should always be true (otherwise the report message is printed during simulation), thus its inverse is a don't-care condition which can be used for optimization.

Two types of *assert* statements were adopted in this work, namely: *don't-care* and *verify*. A *don't-care assert* describes a condition whose cause is external to the partition being synthesized. In contrast, *Verify asserts* describe internal conditions which (in a correct implementation) should form a tautology when combined with the logic in the partition plus associated *don't-care asserts*. Verify conditions can be checked for tautology and used as a verification mechanism, but provide no extra information that synthesis can take advantage of. To be consistent with the way that *don't-care asserts* are represented, the *verify* logic is also inverted and BooleDozer checks it for 0 (which is equivalent to checking the assertion for tautology).

A typical use of these *assert* statements is illustrated in Figure 10. Suppose that the two partitions are large and need to be synthesized separately. In this case, partition 2 will not know that the condition $A =$ "11" never happens unless a designer specifies that as a don't care. This is done by using a *don't-care assert* statement in partition 2.

At the same time, it is important that the designer implementing partition 1 make sure that the condition $A =$ "11" indeed never happens. By using a *verify assert* statement in partition 1, the designer can check it during simulation and BooleDozer can formally check that the assert logic simplifies to 0.

HIS synthesizes the expressions in *assert* statements and connects them to *don't-care* or *verify* boxes, according to the user string in the report message as shown in Figure 10. BooleDozer then uses the *assert* logic to either simplify the network using don't cares (see Section 6) or check the verify conditions.

Another benefit of using *assert* statements in this way is to make the synthesized result more independent from the style of the input description. Consider, for example, the following VHDL fragments, in which it is known by the designer that all bits of input A are orthogonal (that is, only one bit is '1' at a time).

Example 2

```
-- Input Port  A : bit_vector (2 downto 0);
--
Case (A) is                    | If A(0)='1'    then  | If A(0)='1' then
  when  "001" => out1 <= d1;   |      out1 <= d1;     |    out1 <= d1; end if;
  when  "010" => out1 <= d2;   | elsif A(1)='1' then  | If A(1)='1' then
  when  "100" => out1 <= d3;   |      out1 <= d2;     |    out1 <= d2; end if;
```

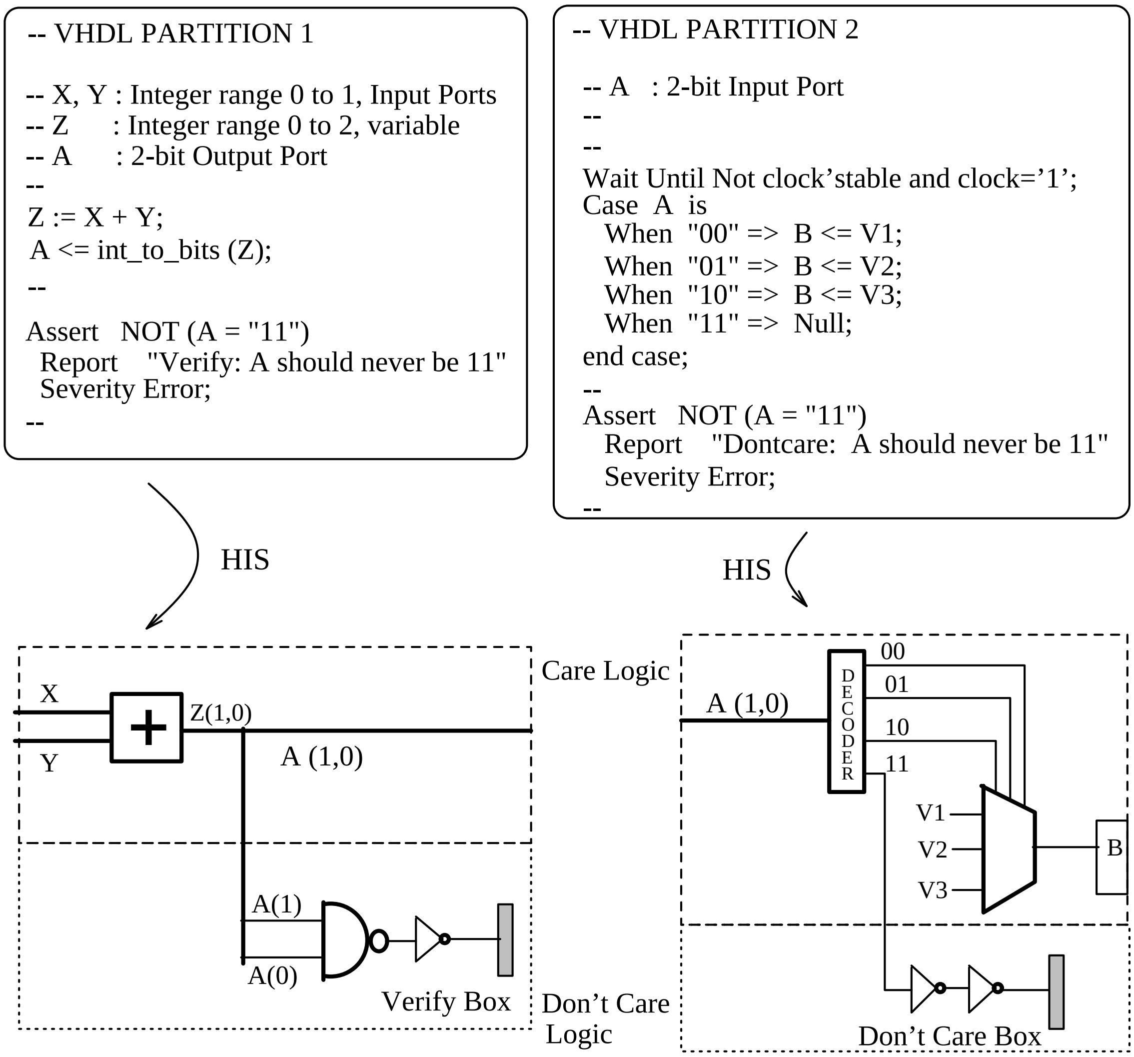

Figure 10: Example of Don't-Care and Verify Assert statements and networks.

```
  when others => out1 <= 'X'; |  elsif A(2)='1' then  |  If A(2)='1' then
end case;                     |       out1 <= d3;     |     out1 <= d3; end if;
                              |  end if;              |
--           (1)                        (2)                      (3)

Assert orthogonal_bits(A)
 Report "Dontcare: A does not have orthogonal bits"
 Severity Error;
```

These three fragments, when synthesized, will normally produce different and unoptimal implementations. However, if an *assert* statement (as shown above) is used, then all three fragments will be synthesized to the same, optimal implementation.

5.3.2 Don't Cares Extracted by High-Level and RTL Synthesis

This section deals with two types of implicit don't cares: those created by the use of multi-bit variables and those derived from unreachable states in finite-state machines.

5.3.2.1 Multi-bit Variables

Variables representing integers and enumeration types are widely used in modern hardware-description languages. For synthesis purposes, such a variable is mapped onto a bit-vector which may unnecessarily increase the design space represented by the variable. Consider, for example, an integer variable defined with a range from 0 to 16. It needs 5 bits, however almost half the design space (from 17 to 31) represented by 5 bits, by definition, will not be part of the care set of the design. The design may be optimized further if the condition $A > 16$ is used as a don't-care condition.

The don't cares associated with unused values of multi-bit variables are extracted by HIS by analyzing the uses of multi-bit variables in conditional operations in the input description. For example, if a variable used in a *Case* statement (in VHDL) is compared with five values, then it is implied that these values are the only ones assumed by the variable, and all other unused values can be used as don't-care conditions. This implication follows from the syntax of VHDL which requires all sequential *Case* statements and *Selected signal assignment* statements to be complete, that is, all values must be specified[1].

In principle one could derive such don't-care conditions in a more general way, by looking at the type of the variable and all its assignments and uses. In descriptions with only simple assignments, it is possible to derive the set of values that a variable can assume by statically propagating values through assignments and uses (which can be done using data-flow analysis techniques). However, in more realistic descriptions containing general operations this is not possible. The approach in this paper is conservative because it only looks at the constructs whose implied semantics unambiguously define the care set of values.

In order to guarantee correctness, the algorithm for extracting don't-care conditions must take into account all care conditions in the path leading to the use of the multi-bit variable. Consider, for example, the following fragment of a VHDL Process:

Example 3

```
-- Input Port        in1  : integer range 0 to 8
-- Internal variable mbv1 : integer range 5 to 8;
mbv1 := in1;
Case (mbv1) is
  when  5 | 6 => out1 <= d1;
  when  7     => out1 <= d2;
  when  8     => out1 <= d3;
end case;
```

If this fragment is the whole description, then the expression representing the unused values of $mbv1$ is:

$$DC = not\ ((in1 = 5) \mid (in1 = 6) \mid (in1 = 7) \mid (in1 = 8)) \tag{5}$$

[1] if *when others* is used, it comprises all valid values not used by other *when* clauses.

This expression can be used as a don't-care condition for the optimization of the decoding logic.

In the case of more complex descriptions with different conditions leading to the use of the multi-bit variables (that is, to the Case statement), the don't-care expression must be restricted to the conditions under which the Case statement is executed. Consider the following extension to the previous example:

Example 4

```
-- Input Port        in1  : integer range 0 to 8
-- Internal variable mbv1 : integer range 5 to 8;
-- Internal variable mbv2 : integer range 0 to 4;

Wait Until Not clock'Stable And clock='1';   -- start of state 0
Case (in1) is
  when  0 | 1 | 2 => out0 <= d1;
  when  3 | 4 | 5 => out0 <= d2;
  when  6 | 7 | 8 => out0 <= d3;
end case;

Wait Until Not clock'Stable And clock='1';   -- start of state 1
mbv1 := in1;
Case (mbv1) is
  when  5 | 6 => out1 <= d1;
  when  7     => out1 <= d2;
  when  8     => out1 <= d3;
end case;

Wait Until Not clock'Stable And clock='1';   -- start of state 2
mbv2 := in1;
Case (mbv2) is
  when  0 | 1 => out2 <= d1;
  when  2 | 3 => out2 <= d2;
  when  4     => out2 <= d3;
end case;
```

Note that although the second case statement in the Example 4 is exactly the same as in Example 3, the don't-care expression derived in Example 3 cannot be used directly in Example 4. The difference lies in the semantics implied in the two examples. In Example 3, the case statement was always executed, therefore $in1$ and $mbv1$ were always the same and the don't-care conditions derived from $mbv1$ could always be expressed in terms of $in1$.

In example 4, however, input $in1$ is compared with three sets of values (the three case statements) in three different states of the machine. The functionality implied in this description is such that: 1) $in1$ can assume values from 0 to 8 only if the machine is in State 0; 2) $in1$ can assume values from 5 to 8 only if the machine is in State 1; and 3) $in1$ can assume values from 0 to 4 only if the machine is in State 2. These qualifying conditions must be part of the don't-care expressions.

Hence, three don't-care expressions can be derived from Example 4:

$$\begin{aligned} DC1 &= (State\_0)\ \&\ (not\ ((in1 = 0) \mid (in1 = 1) \mid (in1 = 2) \mid \\ &\qquad (in1 = 3) \mid (in1 = 4) \mid (in1 = 5) \mid \\ &\qquad (in1 = 6) \mid (in1 = 7) \mid (in1 = 8))) && (6) \\ DC2 &= (State\_1)\ \&\ (not\ ((in1 = 5) \mid (in1 = 6) \mid (in1 = 7) \mid (in1 = 8))) && (7) \\ DC3 &= (State\_2)\ \&\ (not\ ((in1 = 0) \mid (in1 = 1) \mid (in1 = 2) \mid \\ &\qquad (in1 = 3) \mid (in1 = 4))) && (8) \end{aligned}$$

The algorithm for computing these don't-care expressions consists of three steps:

1. Compute a sub-expression representing the OR of all unused values of the multi-bit variable used in a *Case* statement, which is equivalent to the NOR of all valid values.

2. Compute the condition under which the *Case* statement is executed. This is done as part of synthesis and thus has no computational overhead.

3. The don't-care condition for the multi-bit variable used in the *Case* statement is given by the AND of the expressions computed in steps 1 and 2.

The HIS system maps these don't-care expressions into gates, representing the don't-care logic, as shown in Figure 11.

5.3.2.2 Unreachable States

In a finite-state machine with N states, encoded using E bits, there will be $2^E - N$ unreachable states. The expression representing all unreachable states can be used as a don't-care condition for the optimization of the design.

Using unreachable states as don't-care conditions for logic optimization is not new and is commonly used by state encoding programs, such as Mustang [6]. In most cases, the unreachable states are added to the state transition table as output don't-care conditions (that is, under those conditions the outputs and next states are don't cares), in a two-level representation. Then Espresso-like techniques are used to optimize the table.

In this paper the principle of using unreachable states is similar, however they are represented directly as a multi-level network, thus avoiding possible complexity problems of two-level representations.

As a result of high-level and RTL synthesis, HIS produces a controller and a datapath. The controller contains a finite-state machine which uses a state decoder to produce signals representing all reachable states. The don't-care expression for unreachable states is given by the complement of all reachable states, which is implemented simply by a NOR gate with inputs coming from the state decoder. The don't-care expression for unreachable states in Example 4, assuming a sequential encoding in the minimum number of bits, is given by:

$$DC4 = not\ ((State\_reg = "00") \mid (State\_reg = "01") \mid (State\_reg = "10")) \qquad (9)$$

The don't-care expression representing all unreachable states is mapped into gates in the don't-care logic, as shown in Figure 11.

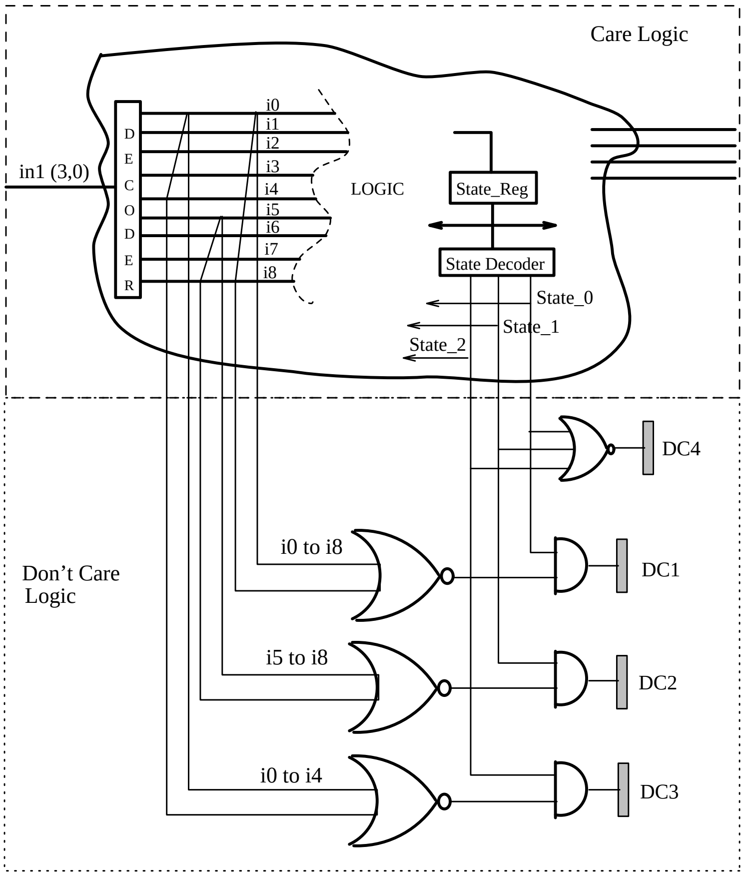

Figure 11: Example of network generated by HIS, consisting of *care* logic and *don't-care* logic representing multi-bit variables and unreachable states.

6 Logic Simplification Using Don't Cares

Given a care and a don't care network, this section describes how to optimize the care network, while taking advantage of the don't care network. During optimization the care network may change its function, but the whole merged network must retain its functionality – this is the condition that guarantees correctness of the care network.

In BooleDozer a test generator [18, 19] is used during optimizations to perform Boolean reasoning. A test generator's main advantage is that it does not need any special representation of the logic (such as cubes, BDDs), which may grow exponentially with the size of an equivalent network. This is especially important when dealing with don't cares, because of their potential volume. While the size of the don't care representation is an issue in some systems [20], it is not a problem in BooleDozer because the don't care information is concisely represented by a network.

Furthermore, there is no loss of reasoning power in using a test generator. As it has been shown in [21], any network (interpreted using primitive boolean functions) can be converted to any equivalent network using transformations based on a test generator. The test generator needs to be asked only one kind of question – "Is a given pin redundant for stuck-at-0 or stuck-at-1?". This

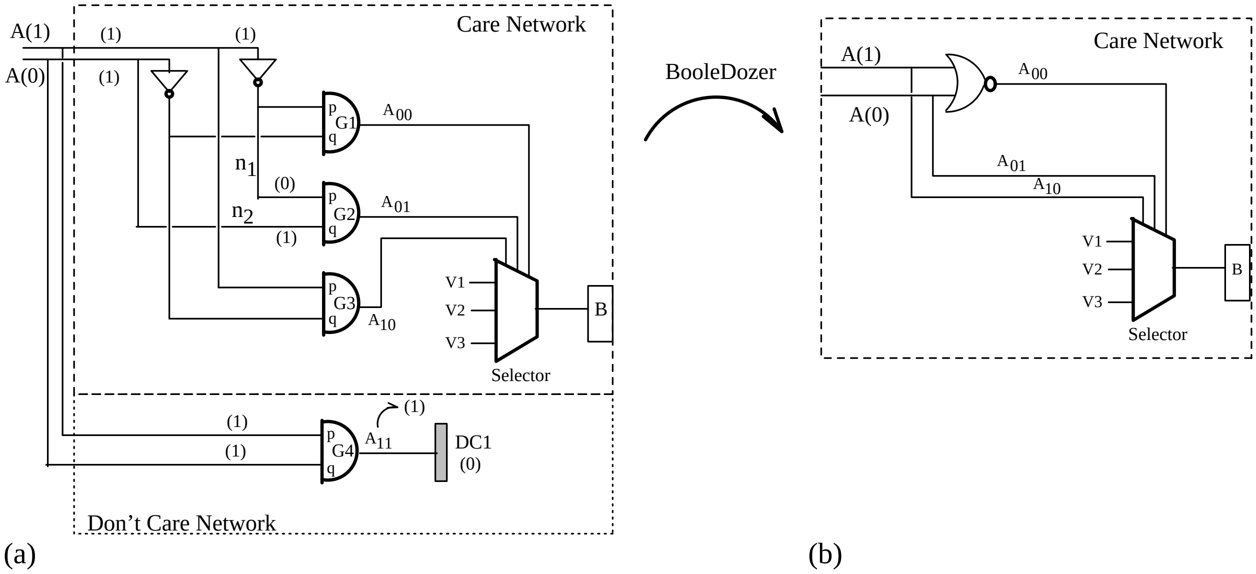

Figure 12: Example of initial merged network and optimized care network.

question is asked by synthesis transformations operating on a network and they use the answer in deciding whether a particular transformation is legal.

Test generators have been previously used for optimization with respect to don't cares in [9]. In contrast to that method, the approach used in this paper is more efficient and does not need any modifications to existing test generators.

Synthesis transformations operate only on the care network, and are not even aware of the existence of the don't-care network. It is important that the don't-care network be invisible to synthesis transformations and timing analysis, so that they get a correct picture of the logic, including number of connections, fanout, etc. They perform optimizations subject to the don't cares, by letting the test generator determine the validity of any logic change. The test generator sees the whole merged network and reports a particular fault as testable if it is testable in the whole merged network, not just the care network. In BooleDozer the test generator is used in this manner by redundancy removal, transduction [22], selector generation [23], false path elimination [24], and incremental synthesis [25].

In order to understand how the test generator is used for logic optimization, consider the network given in Figure 12. Figure 12a shows the merged (Care + Don't-Care) network as generated by high-level and RTL synthesis. When optimizing the *Care* network, a redundancy removal algorithm *asks* the test generator, for example, whether input p of gate G_2 is testable for stuck-at-1. To answer this question the test generator first sets all don't-care nets (i.e., A_{11}) to 0. This forces the test generator to consider only those test patterns that make all don't-care nets 0. Then the test generator proceeds as usual when looking for a test pattern. In doing so, it must set net n_1 to 0 and net n_2 to 1 and propagate the values towards the primary inputs. This propagation reveals that both $A(1)$ and $A(0)$ have to be 1. This, however, forces the don't-care net A_{11} to 1, which violates the initial assumption that don't-care nets should be 0. The result of this violation is that input p of gate G_2 is untestable for stuck-at-1 and can be connected to a constant 1. By similar analysis, it can be shown that input q of gate G_3 is also untestable for stuck-at-1. Consequently, both gates G_2 and G_3 can be eliminated, eventually resulting in the optimized logic shown in Figure 12b.

7 Results

In practice the use of don't cares in synthesis (in HIS and BooleDozer) is restricted to the types of don't cares presented in sections 5.1 and 5.3. The results presented in this section use the types of don't cares shown in section 5.3.

Three experiments were devised in order to validate the approach in this paper. The first two consist of running HIS and BooleDozer using the don't-care optimizations described, in two sets of examples.

Table 1 shows the results of running HIS+BooleDozer with and without don't cares, on small internal IBM design examples. All don't cares were explicitly specified by the designers using *Assert* statements. The same synthesis script was used whether don't cares were considered or not. Logic optimization and technology mapping were done with average area and delay efforts and simple timing correction was applied at the end. Table 1 presents the values of area in cells[2] and *Worst-Case* delay in time units (normalized to the shortest delay in each row).

Design	Area (cells)			Worst Delay (tu)		
	DC	no DC	%Δ	DC	no DC	%Δ
Example1	168	177	5.1%	1.00	4.40	77.3%
Example2	445	410	-7.9%	1.00	4.02	75.1%
Example3	489	541	9.6%	1.00	2.86	65.1%
Example4	522	606	13.9%	1.00	4.18	76.1%
Example5	635	757	16.1%	1.00	1.58	36.6%
%Δ Average			7.4%			66.0%

Table 1: Synthesis results in area and delay, using (DC) and not using don't cares (no DC). All examples contain don't cares specified using *assert* statements. %Δ is the difference expressed as a percentage of the larger value.

The second set of examples has high-level synthesis benchmarks as well as design partitions containing don't cares coming from multi-bit variables and/or unreachable states. Table 2 shows synthesis results in area and delay, with and without don't cares for these examples.

The third experiment (Table 3) consists of comparing the results of HIS+BooleDozer using don't cares with the results of HIS + Mustang + Espresso + BooleDozer where don't cares were solely used by Mustang and Espresso. Two small finite-state machines containing unreachable states were used and both results used the same state encoding (as given by Mustang and back-annotated onto the VHDL for the HIS+BooleDozer path). Table 3 gives the values of area and delay for the two machines.

From Table 1 it can be seen that don't cares specified by designers can have a significant effect on area and delay. Table 2 shows that don't cares due to multi-bit variables and unreachable states, extracted by high-level synthesis, have a smaller average impact on area and delay, although significant in some cases. Most of the examples in Table 2 contained small don't care sets. Table 3

[2] A cell is a basic unit of area

Design		Area (cells)			Worst Delay (tu)		
		DC	no DC	%Δ	DC	no DC	%Δ
His6	us	37	45	17.8%	1.00	1.28	21.9%
Mark2	us	1377	1378	0.1%	1.00	1.03	3.4%
Bb_Act	mb	69	76	9.2%	1.00	1.03	3.3%
Bus_Cnt	mb	304	311	2.3%	1.00	1.06	5.2%
Lbsm	mb	851	874	2.6%	1.00	1.02	1.8%
Drink40	us	126	143	11.9%	1.00	1.19	15.9%
Kopfsm	us	282	281	-0.4%	1.00	1.02	1.4%
Pci	mb	69	88	21.6%	1.00	1.10	9.2%
Sevens	mb	61	71	14.1%	1.00	1.28	21.9%
Rtpt1a	mb	1255	1297	3.2%	1.00	1.01	0.5%
Memop	us	229	232	1.3%	1.00	1.24	18.9%
Addrc	mb/us	1305	1405	7.1%	1.08	1.00	-7.0%
M6502	mb/us	6409	6635	3.4%	1.03	1.00	-3.3%
%Δ Average				7.2%			7.2%

Table 2: Synthesis results in area and delay, using and not using don't cares. All examples contain don't cares from multi-bit variables (mb) and/or unreachable states (us) extracted by high-level synthesis.

Design	Area (cells)			Worst Delay (tu)		
	DC	Mustg/Espr	%Δ	DC	Mustg/Espr	%Δ
Drink40	138	148	6.7%	1.00	1.29	22.5%
Kopfsm	279	268	-3.9%	1.00	1.26	20.5%

Table 3: Synthesis results in area and delay given by HIS+BooleDozer (using don't cares – column labeled DC) and HIS+Mustang+Espresso+BooleDozer (where don't cares were used only by Mustang and Espresso – column labeled Mustg/Espr). Only don't cares for unreachable states were used.

shows that the approach presented in this paper is comparable in area and delay with the PLA optimization approach (represented here by Mustang and Espresso). However, the approach in this paper has the potential to use much larger don't-care conditions than the PLA approach. Furthermore, the approach in this paper allows don't cares to be taken advantage of at any stage of synthesis (e.g., during timing optimization of mapped logic), not only during initial optimization.

The extra CPU time used by HIS for the generation of the don't-care logic is negligible. The extra CPU time used in BooleDozer for the don't-care optimizations depends on the characteristics of the design. While extra time has to be spent taking into account the don't care logic, the reduction in the size and delay of the network causes the other algorithms to run faster, in some cases even reducing the total CPU time. In all the examples presented, the use of don't cares did not change the overall CPU time by more than 10%.

7.1 Don't Cares and False Paths

During logic design it happens very often that a critical path reported by a static timing analyzer is "false". That means that no input pattern can allow a signal to propagate along that path, and hence that path does not constitute a timing problem. Therefore, a lot of work has been done on analyzing false timing paths and reporting a more optimistic delay of a circuit [24, 26, 27]. The general experience from published work is that it is rather rare to find all the critical paths reported by a static timing analyzer to be false. This is in contrast to designers' experience where such a situation is very common. The purpose of this section is to consider whether the discrepancy might be due to don't cares; namely the designer knows some constraints of his environment, which he does not tell the timing analyzer.

To test this hypothesis several examples containing don't cares (of the type described in section 5.3) were analyzed for false paths. However, most of the designs where a designer specified some assertions did not generate enough false paths. To increase the number of false paths, extra artificial don't cares were inserted. How the artificial nature of the don't cares impacts the results is explained later.

For false path analysis the algorithm of [24] was used, where false paths are eliminated, at the expense of possibly replicating some nodes. The reported delay is then the delay calculated by a static timing analyzer. This approach is less accurate than, say, exhaustive simulation, because the replication of nodes changes loading of nets. This again has an impact on the conclusions.

Table 4 shows the results on several partitions. Each partition was synthesized by technology independent optimization, technology mapping, followed by simple fanout and power level adjustment. No timing correction was performed because the results would depend too much on specific timing requirements. The synthesis was performed in two ways – without any don't cares and with don't cares. In the latter case it was asserted that random groups of primary inputs and latch outputs were hot-1 encoded. Table 4 reports delays under six different conditions, where all delays have been normalized to the shortest delay in each row.

OPTIMIZED WITHOUT DC			OPTIMIZED WITH DC		
static	dynamic without DC	dynamic with DC	static	dynamic without DC	dynamic with DC
1.16	1.16	1.07	1.00	1.00	1.00
1.83	1.83	1.17	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00
1.16	1.16	1.06	1.00	1.00	1.00
1.09	1.09	1.09	1.00	1.00	1.00
1.98	1.91	1.43	1.00	1.00	1.00
1.17	1.13	1.10	1.06	1.06	1.00
1.31	1.28	1.11	1.00	1.00	1.00

Table 4: Delays depending on whether don't cares are considered for optimization and for timing analysis

The six columns of Table 4 will be explained one at a time. The first column ("static") is the result of synthesizing each partition and them evaluating its delay using a static timing analyzer. The second column ("dynamic without DC") is based on the same synthesized logic, but delay is calculated only after eliminating false paths. That results in reduced delay for the last three partitions.

In the third column ("dynamic with DC") false paths are also eliminated, but (in contrast to the previous column) don't cares are taken into account in identifying the false paths. That results in identifying more paths as false and in obtaining smaller delay yet.

In the second half of Table 4 ("OPTIMIZED WITH DC") don't cares are used throughout the synthesis process, and the resulting delays are reported. The fourth column ("static") reports the delay without considering any false paths. For the the fifth and sixth columns false paths are eliminated, and as above, only the last column ("dynamic with DC") uses don't cares in identifying the false paths.

Consider the first half of Table 4 where optimization is done without don't cares. It can be seen that false path analysis reduces delay more if it considers don't cares (third column) than if it does not (second column). Thus a timing analysis tool without access to don't care information (second column) will report false path problems on only the last three of the nine examples, while a designer aware of don't care information (third column) would see a false path problem in six examples.

This is consistent with the hypothesis that many false path problems are caused by don't cares. That implies two things; first it is important to collect don't care information before doing false path analysis; secondly designers may be more willing to provide the don't care information if that will lead to fewer false paths in future synthesis runs.

If don't care information is available it can be used not just for false path analysis but also for other types of optimizations (second half of Table 4). Such optimizations may result in smaller static delay (column 4 is better than any of the first three columns), and in fewer false path problems (columns 5 and 6).

The numbers reported in Table 4 are inaccurate in several ways. First, the method [24] of calculating dynamic delay is inaccurate because it changes loading of nets. Secondly, the don't care assertions are generated artificially and they are much more massive than can be expected in practice. This makes the effect of don't cares exaggerated, which was necessary so that an effect could be observed. That means that the results can be used to indicate a trend, but not the magnitude of the trend. Thirdly, these results cannot claim that don't cares alone are sufficient to explain the gap between false path problems reported by designers and reported in the literature, but don't cares are certainly a contributing factor.

8 Conclusions

Effective use of don't cares requires solving several theoretical and practical problems. The theoretical problems are caused by the need to have all tools in a methodology treat don't cares in a consistent manner. The theoretical problems addressed in this paper included a formalism shared by all the tools, a definition of correct implementation, and a guarantee of replaceability. The practical problems considered in this paper involved derivation of don't-care conditions from a design language, mapping them into representations suitable for synthesis, and use of the don't-care conditions for optimization.

This paper considered three common kinds of don't cares and presented solutions for the theoretical and practical problems in each kind. The don't cares considered in Section 5.1 are theoretically the most powerful (more powerful than partial functions or Boolean relations), and are very convenient for designers as well as synthesis. Their main disadvantage lies in the absence of universal replaceability; they allow safe replaceability only. The don't cares considered in Section 5.2 are less powerful, but they offer a possibility of universal replaceability. However, this possibility can be realized only under conditions too stringent for existing design languages to satisfy. The don't cares considered in Section 5.3 are the least powerful, but they guarantee universal replaceability, which makes them the most practical.

This paper presented techniques and algorithms for practical use of the don't care conditions of Section 5.3, which encompass both high-level and logic synthesis domains. On the high-level domain, it was shown that don't care conditions can be explicitly specified by the designer by using *assert* statements, and then used by logic synthesis for optimization and verification. Furthermore, don't care conditions for unreachable states and multi-bit variable can be efficiently derived during high-level synthesis. It was shown that these conditions can be concisely mapped to a general network, thus avoiding the need for a two-level representation, and allowing large don't cares to be used. In the logic synthesis domain, it was shown that the don't care network can be used for the optimization of the care network by using test generation techniques (instead of PLA-type optimizations).

Experimental results were used to demonstrate three things. First, for designs where don't cares were sufficiently small to be handled by two-level methods (e.g., Espresso, Mustang), it was shown that the method in this paper has comparable effectiveness. Secondly, it was shown that this method can also handle much larger don't cares, resulting in significant improvements in logic quality. Thirdly, experimental results showed that don't cares can cause false timing paths. That means that getting don't-care conditions from designers and using them for logic optimization can help with the false path problem by reducing the number of false paths and by shortening the delay of the remaining true paths.

9 Acknowledgements

This work would not have happened without Ken Shepard providing both a motivation and examples. M. Berkelaar and S. Prakash made an important contribution to the implementation.

References

[1] R. K. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* The Netherlands: Kluwer Academic Publishers, 1985.

[2] R. Wei, S. G. Rothweiler, and J. Jou, "BECOME: Behavior level circuit synthesis based on structure mapping," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, (Anaheim, CA), pp. 409–414, ACM/IEEE, July 1988.

[3] R. A. Bergamaschi, D. Lobo, and A. Kuehlmann, "Control optimization in high-level synthesis using behavioral don't cares," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*, (Anaheim), pp. 657–661, ACM/IEEE, June 1992.

[4] S. C.-Y. Huang and W. H. Wolf, "Performance-driven synthesis in controller-datapath systems," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 68–80, March 1994.

[5] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: State assignment of finite state machines for optimal two-level logic implementations," *IEEE Transactions on Computer-Aided Design*, vol. CAD-9, pp. 905–924, September 1990.

[6] S. Devadas, H.-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State assignment of finite state machines targeting multi-level logic implementation," *IEEE Transactions on Computer-Aided Design*, vol. CAD-7, December 1988.

[7] R. E. Bryant, "Graph based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, August 1986.

[8] K. Bartlett, R. K. Brayton, G. Hachtel, R. M. Jacoby, C. R. Morisson, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "Multilevel logic optimization using implicit don't cares," *IEEE Transactions on Computer-Aided Design*, vol. CAD-7, pp. 723–740, June 1988.

[9] D. Brand, "Redundancy and don't cares in logic synthesis," *IEEE Transactions on Computers*, vol. C-32, pp. 947–952, October 1983.

[10] H. Savoj and R. K. Brayton, "The use of observability and external don't cares for the simplification of multi-level networks," in *Proceedings of the 27rd ACM/IEEE Design Automation Conference*, ACM/IEEE, June 1990.

[11] R. K. Brayton and F. Somenzi, "An exact minimizer for boolean relations," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 316–319, IEEE, November 1989.

[12] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, November 1987.

[13] S. Devadas, H.-K. T. Ma, and A. R. Newton, "Redundancies and don't cares in sequential logic synthesis," *Journal of Electronic Testing: Theory and Applications*, January 1990.

[14] IEEE, *IEEE Standard VHDL Language Reference Manual*, March 1988.

[15] C. Pixley, V. Singhai, A. Aziz, and R. K. Brayton, "Multi-level synthesis for safe replaceability," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 442–449, IEEE, November 1994.

[16] R. Bergamaschi, R. O'Connor, L. Stok, M. Moricz, S. Prakash, A. Kuehlmann, and D. S. Rao, "High-level synthesis in an industrial environment," *IBM Journal of Research and Development*, vol. 39, pp. 131–148, January/March 1995.

[17] D. Brand, R. Damiano, L. van Ginneken, and A. Drumm, "In the driver's seat of BooleDozer," in *Proceedings of the IEEE International Conference on Computer Design*, pp. 518–521, IEEE, October 1994.

[18] S. Kundu, L. H. Huisman, I. Nair, V. S. Iyengar, and L. N. Reddy, "A small test generator for large designs," in *Proceedings of the International Test Conference*, pp. 30–40, September 1992.

[19] R. P. Kunda, P. Narain, J. A. Abraham, and B. D. Rathi, "Speed up of test generation using high-level primitives," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, (Orlando), pp. 594–599, ACM/IEEE, June 1990.

[20] A. Saldanha, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Multi-level logic simplification using don't cares and filters," in *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 227–282, ACM/IEEE, June 1989.

[21] W. Kunz and P. R. Menon, "Multi-level logic optimization by implication analysis," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 6–13, IEEE, November 1994.

[22] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The transduction method – design of logic networks based on permissible functions," *IEEE Transactions on Computers*, vol. 38, pp. 1404–1424, October 1989.

[23] M. Berkelaar and L. P. P. P. van Ginneken, "Efficient orthonormality testing for synthesis with pass-transistor selectors," in *Proceedings of the International Workshop on Logic Synthesis*, May 1995.

[24] H. C. Chen and D. H. C. Du, "Circuit enhancement by eliminating long false paths," in *Proceedings of the 29rd ACM/IEEE Design Automation Conference*, pp. 249–252, ACM/IEEE, June 1992.

[25] D. Brand, A. Drumm, S. Kundu, and P. Narain, "Incremental synthesis," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 14–18, IEEE, November 1994.

[26] D. Brand and V. S. Iyengar, "Timing analysis using functional analysis," *IEEE Transactions on Computers*, vol. C-37, pp. 1309–1314, October 1988.

[27] P. C. McGeer and R. K. Brayton, *Integrating Functional and Temporal Domains in Logic Synthesis*. The Netherlands: Kluwer Academic Publishers, 1991.