External Don't Cares in Logic Synthesis

Siang-Yun Lee[1], Heinz Riener[2], and Giovanni De Micheli[1]

[1] École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
[2] Cadence Design Systems, Munich, Germany

Abstract. Don't-care conditions are flexibilities in logic networks that can be used to optimize the networks by re-expressing incompletely-specified Boolean functions. Don't cares may arise from the internal structure of the network in the presence of reconvergent paths, or be given externally from the environment, such as the cascaded next-stage or previous-stage circuits. Theories on don't-care computation have been extensively studied in the 90s, and utilization of internal don't cares has become a common practice in modern logic synthesis tools. However, representation and consideration of external don't cares have rarely been discussed. There is currently no open-source logic synthesis tool capable of accepting external don't cares and utilizing them to optimize the circuit more than what can be achieved without them. In this paper, we first discuss different possible ways to define and represent external don't cares. Identifying the link between logic synthesis and Boolean relations, we use Boolean relations to describe and unify different types of don't cares. As the first step in this line of research, we propose to adopt the simulation-guided paradigm to consider external don't cares in logic optimization. Experimental results show that the presence of external don't cares indeed enables more optimization opportunities. The paper concludes by illustrating future directions towards better utilization of external don't cares.

Keywords: Logic synthesis · External don't cares · Boolean relation

1 Introduction

Logic synthesis, or more specifically, technology-independent logic optimization, is a step in the VLSI design flow after RTL synthesis and before technology mapping, attempting to optimize combinational circuits on technology-independent representations, such as *AND-Inverter Graphs* (AIGs). As a bottom line, the produced result of a logic synthesis algorithm must respect the given functionality of the circuit. To date, this means that the output circuit should be functionally equivalent to the original one, and is usually verified by performing *combinational equivalence checking* (CEC) [5] on the two circuits. However, this requirement might be too strong in some cases. Further high-effort optimization can be enabled by relaxing the requirement of exact functional equivalence and allowing flexibilities external to the combinational circuit under optimization.

Don't cares are flexibilities in logic functions or logic networks where output values of some (local) functions can be changed without violating the (global)

specification [3]. Don't-care conditions may be derived on various scales, from interconnections of logic gates within a combinational network [4] to interactions between submodules in a system [12]. Computation and utilization of don't-care conditions in combinational logic synthesis have often been formulated using incompletely-specified functions [2], also known as permissible functions [11]. Don't cares play a central role in logic synthesis. However, due to the intrinsically high computational complexity of don't-care computation, methods to (under-) approximate them were developed [9, 14, 15]. Nowadays, more powerful and scalable computation of don't cares enabled by *satisfiability* (SAT) solving and simulation is commonly used, but consideration of don't cares is still limited to those within a combinational network [10].

In contrast to internal don't cares computed within a network, external don't cares are flexibilities arising from outside of the combinational network under optimization, derived from a higher-level perspective of the system. For example, cascaded finite state machines may produce don't-care input sequences for each other [12]. As another example, sometimes the system is partitioned into submodules and optimized separately. While their boundaries are intended to be kept, flexibilities on the input-output relations of individual submodules due to their interactions are allowed. Considering external don't cares essentially changes the problem from optimizing a (completely-specified) Boolean function into optimizing a Boolean relation. The solution space is enlarged and the problem complexity is much higher, thus there is currently no open-source logic synthesis tool that supports taking and utilizing external don't cares. Nevertheless, with the increased computation power affordable nowadays, solving such optimization problems should be possible on smaller benchmarks. Moreover, in some applications, users of logic synthesis tools crave to optimize their circuit as much as possible and are willing to afford higher runtime.

This paper serves as a pioneer towards support of external don't cares in logic synthesis. During this journey, we will lay the foundation with mathematical definitions of don't-care conditions in general, explore different flavors of external don't cares, view the general problem of logic synthesis from a Boolean relation perspective, and finally take the first step of considering external don't cares in logic optimization. We will show with experimental demonstrations that external don't cares indeed open up more optimization opportunities that would have been impossible without them. In the end, we will also point out possible directions for future research.

2 Background and Terminologies

2.1 Boolean Functions and Boolean Relations

A *Boolean variable* is a variable taking values in the *Boolean domain* $\mathbb{B} = \{0, 1\}$. The (n-dimensional) *Boolean space* $\mathbb{B}^n$ is an n-ary Cartesian power of the Boolean domain. An (n-input, single-output, completely-specified) *Boolean function* is a function $f : \mathbb{B}^n \to \mathbb{B}$ of n Boolean variables. Multi-output Boolean functions can be seen as an ordered set of single-output functions.

A *Boolean relation* $\mathcal{R}$ is a binary relation over two Boolean spaces $\mathcal{R} \subseteq \mathbb{B}^n \times \mathbb{B}^m$, a *domain* ($\mathbb{B}^n$) and a *codomain* ($\mathbb{B}^m$). Boolean functions are special cases of Boolean relations. More specifically, they can be classified into two types:

- *Completely-specified* Boolean functions are special cases of Boolean relations where the relations are *functional* (i.e., an element in the domain maps into one unique element in the codomain) and *total* (i.e., every element in the domain maps into an element in the codomain). When describing Boolean functions as Boolean relations, an element in the domain, which is a value assignment to all the function's input variables, is also called a *minterm*.
- *Incompletely-specified* Boolean functions are Boolean functions for which the output values under some minterms are not specified. In other words, for some minterm $\vec{b} \in \mathbb{B}^n$, the output value can be either 0 or 1. In terms of Boolean relations, we have both $(\vec{b}, 0) \in \mathcal{R}$ and $(\vec{b}, 1) \in \mathcal{R}$. Given a non-functional Boolean relation $\mathcal{R} \subseteq \mathbb{B}^n \times \mathbb{B}^m$, a completely-specified function $f : \mathbb{B}^n \to \mathbb{B}^m$ is *compatible* with $\mathcal{R}$ if

$$\forall \vec{b} \in \mathbb{B}^n, (\vec{b}, f(\vec{b})) \in \mathcal{R}. \tag{1}$$

When not explicitly noted, *functions* in the remaining of this paper refer to single-output, completely-specified Boolean functions.

2.2 Logic Networks

Logic networks (or simply *networks*) are technology-independent representations of digital circuits. A logic network N is defined by a 4-tuple $N = (I, V, E, O)$, where the two sets (V, E) define a directed acyclic graph. The first set I is the set of *primary inputs* (PIs) to the network. Each element in the vertex set V, referred to as a *node* n, models either a logic gate or a PI. Thus, $I \subseteq V$. Each element (n_i, n_o, c) in the edge set $E \subseteq V \times V \times \mathbb{B}$ models a wire from node n_i to node n_o with a complementation tag $c \in \{0 = \text{regular}, 1 = \text{complemented}\}$ recording the existence of an inverter on the wire. n_i is said to be a *fanin* of n_o and n_o is said to be a *fanout* of n_i. Finally, each *primary output* (PO) in O is a tagged node (n, c) modeling an outgoing wire from a gate or a PI, with or without an inverter.

Cuts. A *cut* in a network, defined over a given set $R \subseteq V$ of *root* nodes, is a set C of nodes such that any path from a PI to a root includes a node in C. Let $\textsc{Cuts}(R)$ denote the set of all cuts for the set R,

$$C \in \textsc{Cuts}(R) \text{ if } \forall i \in I, r \in R, \forall p : i \overset{p}{\rightsquigarrow} r, \exists n \in C : n \in p. \tag{2}$$

When R contains only one node n, $\textsc{Cuts}(R)$ may be abbreviated as $\textsc{Cuts}(n)$ and is also referred to as a cut of n:

$$\textsc{Cuts}(n) \triangleq \textsc{Cuts}(\{n\}). \tag{3}$$

Conversely, given a set C of nodes, a node n is said to be *supported* by C if C is a cut of n. A *logic cone* between a cut $C \in \textsc{Cuts}(n)$ and a node n is the set of all nodes on any path from a node in C to n. All nodes in the logic cone are supported by C.

A cut of a network N is a cut where R is the set of nodes referenced by POs.

$$\textsc{Cuts}(N) \triangleq \textsc{Cuts}(\{n : \exists c, (n, c) \in O\}). \tag{4}$$

Given any set R of roots, the identical set $C = R$ is always a cut by definition, thus such cut is said to be a *trivial cut*. Also, the set I of PIs is always a cut in a network for any possible R.

Global function of nodes. Each node n in a network computes a Boolean function $f_n : \mathbb{B}^{|I|} \to \mathbb{B}$ in terms of the PIs, called the node's *global function*. To express the global functions, a Boolean variable x_i is associated with each PI $i \in I$. Let $\vec{x} = (x_1, \ldots, x_{|I|})$ be the set of all PI variables. By definition, the function of a PI node $i \in I$ is $f_i(\vec{x}) = x_i$. Then, in a topological order, the functions of all nodes in the network can be computed by composing the functions of a node's fanins with the function of the corresponding logic gate. Finally, the PO functions are computed by taking the function of a PO node and inverting if the PO is complemented.

Node function in terms of a cut. The function of a node may also be expressed in terms of a cut supporting it. Given a node n and a cut $C \in \textsc{Cuts}(n)$, the *local function* $f_n^C : \mathbb{B}^{|C|} \to \mathbb{B}$ is the Boolean function derived by associating a Boolean variable with each node in C and computing the local functions of each node in the logic cone between C and n in a topological order. The global functions are a special case of local functions using the PI set I as the cut:

$$f_n \triangleq f_n^I. \tag{5}$$

2.3 Don't-Care Conditions

A *don't care* for an incompletely-specified function is a minterm for which the output value is not specified. In a logic network, although all node functions (in terms of any cut) are completely specified, for some nodes, there may be some minterms where the output values of their functions are *flexible*. In other words, the function f_n^C of a node n in terms of cut C may be modified by changing its output value under some minterms without affecting the global functions of any PO. As a consequence, an incompletely-specified function where these minterms are don't cares and the output values under the other minterms are the same as f_n^C can be used to re-synthesize the logic cone between C and n. Two types of internal don't cares, arising from different reasons, may appear in logic networks:

Satisfiability don't cares. Given a cut $C \in \textsc{Cuts}(R)$ supporting a set R of nodes[3] and let $\vec{x} = (x_1, \ldots, x_{|C|})$ be Boolean variables associated with each node in C, a value assignment $\vec{b}_C \in \mathbb{B}^{|C|}$ to $\vec{x}$ (i.e., a minterm of the local functions f_n^C of any node $n \in R$) is a *satisfiability don't care* (SDC) if this value combination never appears under any PI value assignment:

$$\nexists \vec{b}_I \in \mathbb{B}^{|I|}, (f_n(\vec{b}_I) : n \in C) = \vec{b}_C. \tag{6}$$

Observability don't cares. Given a node n and a cut $C \in \textsc{Cuts}(n)$ and let $\vec{x} = (x_1, \ldots, x_{|C|})$ be Boolean variables associated with each node in C, a value assignment $\vec{b}_C \in \mathbb{B}^{|C|}$ to $\vec{x}$ (i.e., a minterm of the local function f_n^C) is an *observability don't care* (ODC) with respect to n if none of the PO functions are affected by flipping the output value of f_n^C under $\vec{b}_C$:

$$\forall \vec{b}_I \in \mathbb{B}^{|I|}, (f_n(\vec{b}_I) : n \in C) = \vec{b}_C \implies \forall o \in O, f_o^*(\vec{b}_I) = f_o(\vec{b}_I), \tag{7}$$

where f_o^* is the PO function derived by replacing any regular outgoing edge of n with a complemented one and replacing any complemented outgoing edge of n with a regular one.

3 Computation of Internal Don't Cares

Appearance of "don't care" as a technical term in the literature dates back to as early as the 80s [3]. Pioneering research attempted to derive don't cares in multi-level networks and use them in two-level minimization to resynthesize part of the network [2]. Theories on don't-care computation were formulated based on symbolic computations propagated through the network [4, 11]. Until the late 90s, computation of don't cares had been implemented using *Binary Decision Diagrams* (BDDs). Due to scalability concerns, approximated computation was adopted [9], and the compatibility of ODCs was studied to avoid re-computation of ODCs in the network once an ODC is used to change the function of a node [14]. Since the early 00s, computation tools of don't cares have moved from BDDs to SAT, enabling using complete, instead of approximate, don't cares while maintaining scalability [10].

In many modern logic synthesis tools, internal don't cares are derived locally (under-approximated) using bit-parallel circuit simulation:

- To compute the SDCs for a given set C of nodes, we first find another cut $C_0 \in \textsc{Cuts}(C)$ supporting C. Then, we perform circuit simulation by assigning projection functions to nodes in C_0 and obtain the local functions of nodes in C in terms of C_0, represented as truth tables. Finally, by analyzing

[3] The supported set R does not involve in the definition of SDCs, so it can, in theory, be empty and C is not necessarily a cut. Although one may define and compute SDCs for any set C of nodes, in practice, SDCs are only meaningful when C is indeed a cut, as SDCs are used to optimize nodes in R.

each bit in the truth tables, we identify the value combinations at C that do not happen, which are the SDCs at C.
- To compute the ODCs with respect to a node n, we first mark the transitive fanout cone of n for a pre-defined number of levels and collect the set R of nodes having fanouts outside of this transitive fanout cone. Then, we find a cut $C \in \textsc{Cuts}(R)$ supporting R and perform circuit simulation to obtain the local functions f_R of nodes in R in terms of C. After adding a temporary inverter at the output of n, we perform another simulation to obtain f_R^*. Finally, we compare the two simulation results to identify the minterms where f_R and f_R^* have identical values, which are the ODCs with respect to n.

4 Definition and Representation of External Don't cares

The general problem of technology-independent combinational logic synthesis asks for generating a logic network that implements the desired output functions and is optimized according to some predefined cost objective. Often, the desired functionalities are given as an un-optimized network. Besides improving the cost objective, a logic synthesis algorithm must preserve the functionalities of the given network. More precisely, the global PO functions must not change after optimization.

However, the desired functionalities may not be completely specified and there may be don't-care conditions external to the network under synthesis. For example, due to the interplay between the network and its environment (other cascaded circuits, previous- and next-stage sequential circuits, or user inputs), some input value combinations may never appear, or some output values are not used ("observed") under certain conditions. These *external don't cares* (EXDCs) can be leveraged to further optimize the network. As it is impossible to derive external don't cares from the network alone, they have to be given to a combinational optimization algorithm from a higher-level algorithm.

4.1 External Controllability Don't cares (External SDCs)

Extending the definition of SDC to the input boundary, a value assignment to the PIs that will never appear is called an *external controllability don't care* (EXCDC). These don't cares are controlled by the environment external to the network.

Mathematically, EXCDCs are essentially a special case of SDCs where the cut C is the set of PIs. The set of EXCDCs of a network N may be given as a function $f^{\mathrm{CDC}} : \mathbb{B}^{|I|} \to \mathbb{B}$:

$$f^{\mathrm{CDC}}(\vec{b}_I) = 1 \iff \vec{b}_I \text{ is an EXCDC.} \tag{8}$$

4.2 External Observability Don't cares

Extending the definition of ODCs to the output boundary, external ODCs are conditions under which some PO values are not of interest. Depending on the

reasons of such situations, there are several ways one may wish to define external ODCs.

As a function of PIs. For each PO $o \in O$, the condition under which the value of o is not observed may be specified as a function of PI values. For example, when the network describes the transition and output logic of a Mealy finite state machine, it may appear that for some previous states (PIs of the network), an output is not used. In this case, the external ODCs are described as a multi-output function $f^{\mathrm{ODCI}} : \mathbb{B}^{|I|} \to \mathbb{B}^{|O|}$:

$$\text{For each } o \in O, f_o^{\mathrm{ODCI}}(\vec{b}_I) = 1 \iff \vec{b}_I \text{ is an EXODC for } o. \tag{9}$$

As a function of other POs. For each PO $o \in O$, the condition under which the value of o is not observed may be specified as a function of other PO values. For example, when the outputs of the network are used in the next stage as a series of cascaded conditional statements such that if a PO of higher priority evaluates to 1, then the lower-priority POs do not matter. In this case, the external ODCs are described as a multi-output function $f^{\mathrm{ODCO}} : \mathbb{B}^{|O|} \to \mathbb{B}^{|O|}$:

$$\text{For each } o \in O, f_o^{\mathrm{ODCO}}(\vec{b}_O) = 1 \iff \vec{b}_O \text{ is an EXODC for } o. \tag{10}$$

The i-th output of f^{ODCO} should not depend on its i-th input. Note that in this case, the don't-care conditions depend on the actual implementation of the network. Using one ODC to optimize and change the function of a PO may invalidate opportunities of using another ODC to optimize some other POs.

As equivalence classes. Instead of specifying external ODCs separately for each PO, the flexible conditions might be some value combinations of a subset of POs. Figure 1 gives an example. Because of the cascaded next-stage logic at the output of N, the value combinations $o_1 = 0, o_2 = 1$ and $o_1 = 1, o_2 = 0$ have the same effect as seen from the system output (both map into $y_1 = 1, y_2 = 1$). Thus, these two PO value combinations may be classified into the same *external observability equivalence class* (EXOEC) and PI minterms that map to one of them are flexible to be re-mapped to either one. More generally, two PO value combinations are *observably equivalent* (in the same EXOEC) if their difference may not be observed when the network is immersed in a larger system. By definition, this is an equivalence relation and is *reflexive* (i.e., if a is observably equivalent to b then b is observably equivalent to a [a and b are indistinguishable]), *symmetric* (i.e., any PO value combination is observably equivalent to itself [trivial]), and *transitive* (i.e., if a is observably equivalent to b and b is observably equivalent to c, then a is observably equivalent to c [a, b and c are indistinguishable]).

EXOECs can be given as a function $f^{\mathrm{OEC}} : \mathbb{B}^{2\cdot|O|} \to \mathbb{B}$:

$$f^{\mathrm{OEC}}(\vec{a}_O, \vec{b}_O) = 1 \iff \vec{a}_O \text{ and } \vec{b}_O \text{ are observably equivalent.} \tag{11}$$

Because f^{OEC} describes an equivalence relation, it must fulfill the reflexivity, symmetry and transitivity properties as described above.

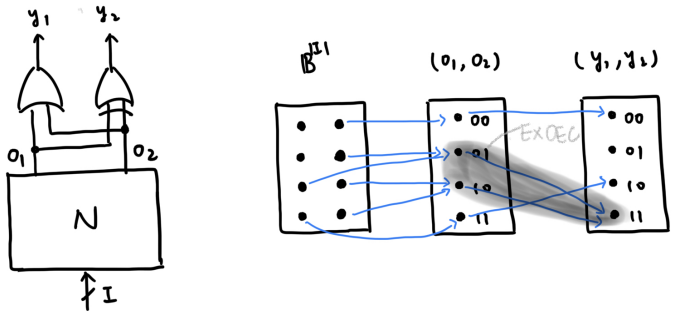

Fig. 1: Example of external observability equivalence classes.

4.3 Logic Synthesis from A Boolean Relation Perspective

A logic network computes a multi-output Boolean function at its primary outputs (i.e., the collection of PO global functions). Hence, it can be described as a Boolean relation. The task of logic synthesis is thus finding an (optimized) network whose output function is compatible with a given Boolean relation $\mathcal{R}$. The presence of external don't cares adds more elements into $\mathcal{R}$.

More generally, given a set C_1 of nodes and a cut $C_0 \in \textsc{Cuts}(C_1)$ supporting it, a Boolean relation $\mathcal{R}_{01}$ can be derived to describe the network functionality between C_0 and C_1. Moreover, if C_1 is also a cut supporting another set C_2, another Boolean relation $\mathcal{R}_{12}$ can be derived and cascaded with $\mathcal{R}_{01}$.

Example 1. Let $C_1 \in \textsc{Cuts}(N)$ be a cut of the network. Let $C_0 = I$ and let $C_2 = \{n : \exists c, (n, c) \in O\}$. We may derive two Boolean relations

$$\mathcal{R}_{01} = \{(\vec{b}_0, f_{C_1}^{C_0}(\vec{b}_0)) : \vec{b}_0 \in \mathbb{B}^{|C_0|}\} \tag{12}$$

$$\mathcal{R}_{12} = \{(\vec{b}_1, f_{C_2}^{C_1}(\vec{b}_1)) : \vec{b}_1 \in \mathbb{B}^{|C_1|}\}, \tag{13}$$

where $f_{C_1}^{C_0}$ is the function the nodes in C_1 compute in terms of C_0, and similarly for $f_{C_2}^{C_1}$.

Figure 2 illustrates the example. According to the definitions in Section 2.3, an (internal) SDC is an element $\vec{b}_1 \in \mathbb{B}^{|C_1|}$ such that

$$\nexists \vec{b}_0 \in \mathbb{B}^{|C_0|}, (\vec{b}_0, \vec{b}_1) \in \mathcal{R}_{01}. \tag{14}$$

Whereas an (internal) ODC for a node $n \in C_1$ is an element $\vec{b}_0 \in \mathbb{B}^{|C_0|}$ such that, let $\vec{b}_1 = f_{C_1 - \{n\}}(\vec{b}_0)$ be the values at $C_1 - \{n\}$ under $\vec{b}_0$,

$$\text{if } ((\vec{b}_1, 0), \vec{b}_2) \in \mathcal{R}_{12}, \text{ then also } ((\vec{b}_1, 1), \vec{b}_2) \in \mathcal{R}_{12}. \tag{15}$$

Generalizing internal and external don't cares, SDCs are elements in a Boolean space (which corresponds to any cut in the network) that are not mapped to

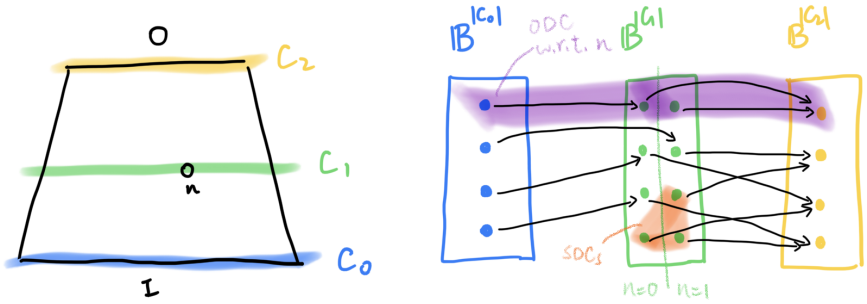

Fig. 2: Illustration of Example 1.

by any element in a previous-stage Boolean space. In contrast, ODCs arise from two elements in a Boolean space that map to the same element in a next-stage Boolean space.

4.4 Boolean Relation as Unified Representation of External Don't Cares.

We observe that none of $f^{\text{ODCI}}, f^{\text{ODCO}}, f^{\text{OEC}}$ is general enough to express the other two. More concretely,

- f^{ODCI} cannot be represented using f^{ODCO} or f^{OEC} because the latter ones lack conditioning on the PI values. There can be multiple PI value combinations leading to the same PO value but only some of them are don't cares.
- The example in Figure 1 cannot be represented using f^{ODCI} or f^{ODCO} because the condition is not simply ignoring the value of a single PO, but flipping the values of both POs.

It is worth noting that f^{ODCO} can be converted into f^{OEC}, but the conversion is not straightforward. Starting from $f^{\text{OEC}}(\vec{a}_O, \vec{b}_O) = \vec{a}_O \leftrightarrow \vec{b}_O$, for each $\vec{b}_O \in \mathbb{B}^{|O|}$ such that $f_o^{\text{ODCO}}(\vec{b}_O) = 1$, we make $f^{\text{OEC}}(\vec{b}_O, \vec{b}_O^*) = 1$, where $\vec{b}_O^*$ is derived by flipping the value corresponding to o in $\vec{b}_O$. The complication comes from propagating the equivalence and keeping the transitivity property of the equivalence relation during the process.

As discussed in Section 4.3, the specification of a logic synthesis problem can be seen as a Boolean relation. In the presence of external don't-care conditions, representation using Boolean relations is inevitable. Figure 3 shows how all of the EXDC functions (represented as networks) introduced in this section may be combined with an initial network to form the relaxed functional specification. This *specification function* $f^{\text{spec}} : \mathbb{B}^{|I|+|O|} \to \mathbb{B}$ is the characteristic function of the implicitly-represented Boolean relation $\mathcal{R}_{\text{spec}}$, asking if a certain pair of PI

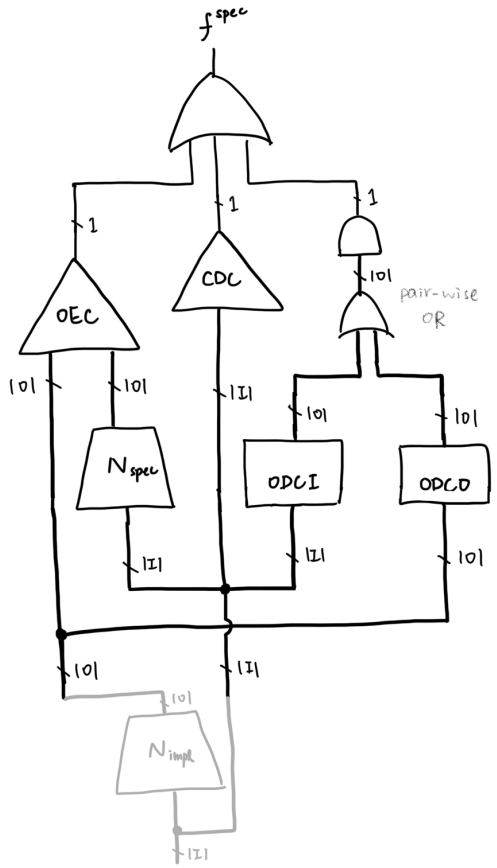

Fig. 3: Circuit representation of a Boolean relation combining an initial network N_{spec} and possible external CDCs and ODCs.

and PO minterms is in $\mathcal{R}_{\text{spec}}$:

$$f^{\text{spec}}(\vec{b}_I, \vec{b}_O) = 1 \iff \text{Under } \vec{b}_I, \vec{b}_O \text{ is acceptable at POs.} \tag{16}$$

$$\iff (\vec{b}_I, \vec{b}_O) \in \mathcal{R}_{\text{spec}} \tag{17}$$

Given f^{spec}, a network is *compatible* if its global PO function f^{impl} fulfills:

$$\forall \vec{b} \in \mathbb{B}^{|I|}, f^{\text{spec}}(\vec{b}, f^{\text{impl}}(\vec{b})) = 1. \tag{18}$$

After logic optimization, a verification step is usually done to ensure the functional correctness of the optimized circuit. Classical CEC verifies if the optimized circuit computes exactly the same global PO function as the original circuit. However, when optimization is performed with external don't cares, such

exact equivalence requirement is too strong. Verification must be modified to use the f^{spec} network instead of a miter network.

5 Optimization with External Don't Cares

To utilize both internal and external don't-care conditions, a *Boolean method*, which considers Boolean functions of the nodes instead of analyzing the network as algebraic expressions (i.e., an *algebraic method*), must be used. As it is computationally too hard to synthesize (or resynthesize) the entire network from a Boolean function or Boolean relation, modern Boolean methods often perform resynthesis and substitution locally within a smaller region, called a *window*.

However, in order to leverage the flexibilities provided by external don't cares, these conditions must be propagated from the boundaries of the network inwards to the windows being resynthesized. For this purpose, we propose to adopt the simulation-guided paradigm [7]. In this paradigm, node functions are approximated by their *simulation signatures*, obtained by performing global simulations using a non-exhaustive set of *simulation patterns* (value assignments to primary inputs). An optimization flow adopting the simulation-guided paradigm consists of the following key steps:

1. Generate a set of simulation patterns.
2. Simulate the network to obtain simulation signatures and use the signatures to compute optimization candidates. The resynthesis computation can be done in a window of any size. Optionally, ODCs may be computed by re-simulating the transitive fanout cone, similar to the method described in Section 3.
3. As the simulation is not exhaustive, a candidate needs to be formally verified before it can be substituted into the network. This is done by solving a SAT instance converted from the network. If a satisfiable assignment is derived by the SAT solver, it is a counter-example proving that the candidate produces unwanted output under a certain PI assignment. The counter-example is added into the simulation patterns. Otherwise, an unsatisfiable result proves that the candidate is valid and thus it is used to substitute the original sub-network.

Using global simulation, internal SDCs are accumulated and propagated within the network as missing bit-patterns in the simulation signatures. EXCDCs can be easily integrated by removing simulation patterns that are don't cares in Step 1. In contrast, EXODCs may only be used when ODC computation is enabled in Step 2 and is considered until primary outputs. In such case, ODC computation is modified as follows: To compute ODCs of a node n, two sets S and S^* of PO simulation signatures are obtained, one (S) by normal simulation and the other (S^*) by adding an inverter at the output of n. For each bit in the simulation signatures (corresponding to a PI simulation pattern), instead of checking if all POs have the same value in S and in S^*, we check if the PO value combination in S^* is in the Boolean relation $\mathcal{R}_{\text{spec}}$.

The SAT instance in Step 3 also needs to be relaxed to take external don't cares into account. Mathematically speaking, the desired SAT instance encodes the complement of Equation (18). Practically, we construct a network N_{spec} as in Figure 3 and convert it into CNF, convert the optimized network N_{opt} also into CNF using the same PI variables I and the same PO variables O as N_{spec}, and assert the output of N_{spec} to be 0. When the instance is unsatisfiable, Equation (18) is true; when the instance is satisfiable, a counter-example violating the Boolean relation is found.

6 Experimental Demonstration

To demonstrate the effectiveness of considering external don't cares in logic synthesis, we present some experimental results in this section. As external don't cares are not provided along with commonly-used benchmarks, we have to generate them by ourselves. The algorithm presented in Section 5 is implemented in the open-source C++ logic synthesis library *mockturtle*[4] [16].

6.1 Applying Randomly-Generated External Don't Cares on Highly-Optimized Circuits

We select 10 medium-sized (comparing to other benchmarks in the same suite) benchmarks from the IWLS'22 programming contest[5]. These benchmarks are originally provided as truth tables of PO functions in terms of PIs (i.e., completely-specified functions). In this experiment, we use the best (smallest in terms of number of gates) synthesized AIGs we have obtained in participation of the contest as the starting point. Without external don't cares, they cannot be optimized any further using the highest-effort (using the entire network as windows, considering internal ODCs until POs, and no limitation on the size of dependency circuits) simulation-guided resubstitution [7].

Table 1 summarizes the optimization results using randomly-generated external don't cares. All of the 10 benchmarks have 12 PIs and 3 POs. Column #Gates lists the number of gates before optimization using EXDCs, columns Δ list the reduction on the number of gates after optimization, columns % list the reduction percentage, and columns Time list the runtime in seconds. All benchmarks use the same external don't care conditions. Column EXCDC is optimized providing only a randomly-generated f^{CDC} having 248 minterms evaluating to 1, column EXODC is optimized providing only $f^{\text{ODCO}} = (f_{y_1}^{\text{ODCO}} = 0, f_{y_2}^{\text{ODCO}} = \neg y_1, f_{y_3}^{\text{ODCO}} = 0)$, and column Both is optimized with both f^{CDC} and f^{ODCO}.

This experiment shows that providing external don't cares indeed enables further optimization opportunities, and that the presented optimization technique works in practice.

[4] Available: `https://github.com/lsils/mockturtle`
[5] `https://www.iwls.org/iwls2022/`

Table 1: Optimization results of using randomly-generated external don't cares on highly-optimized benchmarks.

Benchmark				EXCDC			EXODC			Both		
Name	#PIs	#POs	#Gates	Δ	%	Time	Δ	%	Time	Δ	%	Time
ex70	12	3	263	15	5.70	0.24	0	0.00	0.27	15	5.70	0.35
ex71	12	3	369	2	0.54	0.70	13	3.52	0.75	13	3.52	0.70
ex72	12	3	456	83	18.20	2.03	38	8.33	1.80	35	7.68	2.13
ex73	12	3	208	1	0.48	0.36	1	0.48	0.28	1	0.48	0.24
ex74	12	3	468	40	8.55	3.78	0	0.00	3.78	37	7.91	3.78
ex75	12	3	489	78	15.95	1.43	114	23.31	1.20	132	26.99	1.03
ex76	12	3	246	2	0.81	0.22	1	0.41	0.24	4	1.63	0.27
ex77	12	3	319	89	27.90	0.37	25	7.84	0.32	98	30.72	0.29
ex78	12	3	369	42	11.38	0.36	56	15.18	0.35	52	14.09	0.35
ex79	12	3	365	0	0.00	0.92	20	5.48	0.70	17	4.66	0.78

7 Conclusion and Future Work

This paper aims primarily at raising and defining the problem of logic synthesis with external don't cares. It provides a review on the theoretical definition of don't-care conditions in general, and identifies different ways of representing external don't cares. An emphasis is made on the relation of don't cares and Boolean relations. Finally, using partial simulation and SAT-based verification, we present how external don't cares may be considered in logic optimization. In conclusion, this paper is the first step towards involving external don't cares in logic synthesis. While the theoretical formulations serve as a foundation for future research, the optimization technique is still limited in achievable optimization quality and scalability. In the following, we discuss some future research directions.

7.1 Multi-target Resynthesis

From the Boolean relation point of view, the classical definition of internal ODCs (Equation (7)) is additionally restricted to pairs of elements that only differ in one bit (corresponding to the node under consideration) instead of any pair that map to the same next-stage minterm. The advantage of this approach is that the don't care conditions are used to optimize one node at a time without the need to modify the other nodes. However, it is possible to generalize this class of don't cares by grouping all elements that map to the same element in the next-stage Boolean space together as an OEC and drop the dependency of the definition on a certain node. In this case, multiple nodes need to be optimized together and change their output values.

It is shown in [8] that considering the resynthesis problem of multiple nodes at the same time is necessary for some optimization opportunities to emerge, and the work provides algorithms to describe internal DCs as Boolean relations and

to resynthesize windows from Boolean relations. The problem of multi-target resynthesis specified by a Boolean relation is intrinsically more complex than the well-researched single-target resynthesis [6, 13]. While [1] discusses Boolean relation solving based on divide-and-conquer, further investigation still has potential. With such Boolean relation solver available, logic optimization with external don't cares can be further enhanced.

7.2 Propagation and Management of Observability Equivalence Classes

The biggest problem encountered in the utilization of external don't cares is to properly and efficiently propagate these conditions into the network. Propagation of EXCDCs by partial simulation is relatively straightforward without scalability concern. In contrast, propagation of external ODCs as presented in Section 5 is not scalable. On the one hand, computation of ODCs involves re-simulating the entire transitive fanout cone of the node and verification with EXODCs requires duplicating at least the transitive fanout cone, if not the entire network, in the SAT instance. One possibility to address this issue is to develop methods to propagate external OECs into a cut in the network. On the other hand, management of the OECs is not scalable with respect to the number of POs if PO minterms are explicitly represented. Thus, symbolic representations of OECs and their management methods (especially, merging equivalence classes according to the transitivity rule) need to be developed.

References

1. Bañeres, D., Cortadella, J., Kishinevsky, M.: A recursive paradigm to solve Boolean relations. IEEE Trans. Computers **58**(4), 512–527 (2009)
2. Bartlett, K.A., Brayton, R.K., Hachtel, G.D., Jacoby, R.M., Morrison, C.R., Rudell, R.L., Sangiovanni-Vincentelli, A.L., Wang, A.R.: Multi-level logic minimization using implicit don't cares. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **7**(6), 723–740 (1988)
3. Brand, D.: Redundancy and don't cares in logic synthesis. IEEE Trans. Computers **32**(10), 947–952 (1983)
4. Damiani, M., Micheli, G.D.: Don't care set specifications in combinational and synchronous logic circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **12**(3), 365–388 (1993)
5. Goldberg, E.I., Prasad, M.R., Brayton, R.K.: Using SAT for combinational equivalence checking. In: Nebel, W., Jerraya, A. (eds.) Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001, Munich, Germany, March 12-16, 2001. pp. 114–121. IEEE Computer Society (2001)
6. Lee, S., Riener, H., Micheli, G.D.: Logic resynthesis of majority-based circuits by top-down decomposition. In: Shafique, M., Steininger, A., Sekanina, L., Krstic, M., Stojanovic, G., Mrazek, V. (eds.) 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2021, Vienna, Austria, April 7-9, 2021. pp. 105–110. IEEE (2021)

7. Lee, S., Riener, H., Mishchenko, A., Brayton, R.K., De Micheli, G.: A simulation-guided paradigm for logic synthesis and verification. IEEE Trans. on CAD of Integrated Circuits and Systems (2021). https://doi.org/10.1109/TCAD.2021.3108704
8. Lee, T., Wu, C., Lin, C., Chen, Y., Wang, C.: Logic optimization with considering Boolean relations. In: Madsen, J., Coskun, A.K. (eds.) 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018. pp. 761–766. IEEE (2018)
9. McGeer, P.C., Brayton, R.K.: The observability don't-care set and its approximations. In: Proceedings of the 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD 1990, Cambridge, MA, USA, 17-19 September, 1990. pp. 45–48. IEEE Computer Society (1990)
10. Mishchenko, A., Brayton, R.K.: SAT-based complete don't-care computation for network optimization. In: 2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany. pp. 412–417. IEEE Computer Society (2005)
11. Muroga, S., Kambayashi, Y., Lai, H.C., Culliney, J.N.: The transduction method-design of logic networks based on permissible functions. IEEE Trans. Computers **38**(10), 1404–1424 (1989)
12. Rho, J., Somenzi, F.: Don't care sequences and the optimization of interacting finite state machines. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **13**(7), 865–874 (1994)
13. Riener, H., Lee, S., Mishchenko, A., Micheli, G.D.: Boolean rewriting strikes back: Reconvergence-driven windowing meets resynthesis. In: 27th Asia and South Pacific Design Automation Conference, ASP-DAC 2022, Taipei, Taiwan, January 17-20, 2022. pp. 395–402. IEEE (2022)
14. Saluja, N., Khatri, S.P.: A robust algorithm for approximate compatible observability don't care (CODC) computation. In: Malik, S., Fix, L., Kahng, A.B. (eds.) Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004. pp. 422–427. ACM (2004)
15. Savoj, H., Brayton, R.K.: The use of observability and external don't cares for the simplification of multi-level networks. In: Smith, R.C. (ed.) Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990. pp. 297–301. IEEE Computer Society Press (1990)
16. Soeken, M., Riener, H., Haaswijk, W., Testa, E., Schmitt, B., Meuli, G., Mozafari, F., De Micheli, G.: The EPFL logic synthesis libraries. arXiv preprint arXiv:1805.05121 (2018)