Journal of Universal Computer Science, vol. 11, no. 11 (2005), 1849-1862
submitted: 1/9/05, accepted: 1/10/05, appeared: 28/11/05 © J.UCS

Visualization and Manipulation of Incomplete and Uncertain Dependencies by Decision Diagrams

Denis V Popel
(Department of Computer Science, Baker University, KS 66006, USA
Also with Neotropy LLC, Overland Park, KS 66223
popel@ieee.org)

Abstract: The data mining community is focused on a variety of methods and algorithms to manipulate incompletely specified or uncertain data and their dependencies. The major obstacle in the representation and visualization of incompletely specified data is the size explosion problem through defining undefined or uncertain values, which commonly raises questions about suggested heuristics and their practical applicability. Recently, there is a renewed interest in resolving the size explosion problem for incompletely specified and uncertain data based on symbolic techniques. One of such techniques, decision diagram, has been successfully applied to many knowledge visualization and data manipulation problems.
Key Words: data mining, decision diagrams, incompletely specified functions, minimization
Category: I.6.8, F.4.3

1 Introduction

Incompletely specified functions are common representations of problems with uncertain or incomplete specifications, where unspecified values are called *don't cares*. In incompletely specified functions, unlike completely specified ones, values are assigned to a function for only a subset of combinations of values of variables, called states. Two types of uncertain conditions are distinguished:

- Some input states may never occur and the output states are irrelevant;
- For some input states, the corresponding output states need not to be specified.

Different aspects of decision diagram representation for incompletely specified data have been extensively studied in the following areas: (i) decomposion of functions using decision diagrams [10]; (ii) minimization of logic networks [7]; and (iii) synthesis of *Finite State Machines* (FSM) [11]. Some problems can be classified as *weakly specified* considering initial specification of functions on a restricted number of combinations. Thus, many practical problems in data mining require up to 200 input variables with an enormous number of don't cares [13].

Though the decision diagrams have proved to be a practical tool for symbolic verification and logic function manipulation, they are not always efficient to deal with incompletely specified functions, especially weakly specified. Moreover, the problem of designing an optimal decision diagram for incompletely specified data is NP-complete and only heuristic approaches are of practical use [16]. As an alternative, new graph structures and novel principles, such as "don't care about don't cares", might be exploited and validated for the visualization and manipulation of incompletely specified functions. This paper compares two decision diagram techniques for visualization and manipulation of incompletely specified functions, i.e. the technique based on redefinition of values and the "don't care about don't cares" technique, considering different types of applicable graph structures.

The paper is structured as follows. A brief overview of incompletely specified functions and their representations is given in Section 2. Section 3 describes the basic types of decision diagrams for incompletely specified functions. Some optimization problems associated with decision diagrams are introduced in Section 4. Section 5 concludes the paper and outlines different application of decision diagrams for incompletely specified functions.

2 Representation of incompletely specified functions

An *incompletely specified function* f, also known as a function with don't cares, is the relation where certain combinations of its variables cannot occur. Thus, the truth table of the function f does not generate output values for every possible combination of input values. In the following, we consider the incompletely specified m-valued function f: $\mathbf{A}^n \rightarrow \mathbf{B}$ over the variable set $X = \{x_1, \cdots, x_n\}$, where $\mathbf{A}=\{0,\ldots,r-1\}$ and $\mathbf{B}=\{0,\ldots,m-1\}$, and n is the number of r-valued variables. More formally, the function f can be represented by the sets:

$$\mathcal{C}_i = \{\epsilon \in \mathbf{A}^n | f(\epsilon) = i,\ i = 0, \ldots, m-1\};\ \ \mathcal{DC} = \{\epsilon \in \mathbf{A}^n | f(\epsilon) = d\},$$

where d is an undefined value, and ϵ is a cube. $k = \sum_{i=0}^{m-1} |\mathcal{C}_i|$ is the number of cubes ϵ of the function f. A cube ϵ is labeled with a decimal value $j = 0, \ldots, r^n - 1$. Given a completely specified function f, the set $\mathcal{DC} = \oslash$.

Example 1 *An incompletely specified function $f(x_1, x_2, x_3, x_4)$ is given by the following sets: $\mathcal{C}_0 = \{0, 1, 6, 7, 12, 13, 14\}$, $\mathcal{C}_1 = \{2, 4, 5, 8, 9\}$, and $\mathcal{DC} = \{3, 10, 11, 15\}$. Figure 1(a) shows the corresponding Karnaugh map.*

There are two basic principles of dealing with don't cares of an incompletely specified function. The first one is based on redefining unspecified values of the function, that is assigning concrete values to the don't care values in order to optimize the representation form. This principle has been extensively exploited for several years [4, 16]. Another principle utilizes the idea "don't care about don't cares" [19] which stands for excluding unspecified values from further analysis. Although the latter principle is more attractive, there are no diagram related techniques to manipulate incompletely specified functions that follow the idea “don’t care about don’t cares.”

For incompletely specified functions, different representation forms can be used to describe the function; each associated with a different assignment of binary values to don’t cares. Finding the assignment that leads to the smallest diagram is known to be NP-complete [16] and exact techniques are typically too computationally expensive. Therefore, heuristic algorithms have been developed to address this minimization problem [18].

Example 2 *(Continuation of Example 1) Consider three different assignments of values to the don’t cares: (i) assigning the value 0 to all don’t cares gives* $f = x_1'x_2x_3' + x_1x_2'x_3' + x_1'x_2'x_3x_4'$*; (ii) assigning 1 to all don’t cares results in* $f = x_1'x_2x_3' + x_1x_3x_4 + x_1x_2' + x_2'x_3$*; and (iii) assigning 1 to don’t cares* $\mathcal{DC} = \{3, 10, 11\}$*, and 0 to* $\mathcal{DC} = \{15\}$ *gives* $f = x_1'x_2x_3' + x_1x_2' + x_2'x_3$*. The last choice of values leads to the simplest solution (see the Karnaugh map in Figure 1(b)).*

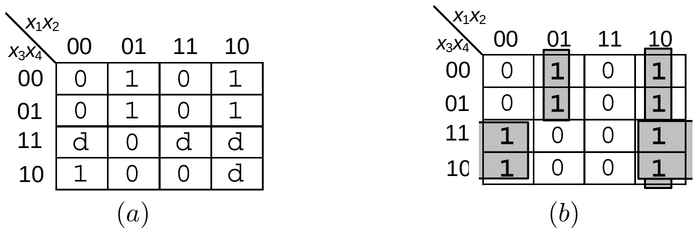

Figure 1: Karnaugh maps for Examples 1 and 2

Following the idea “don’t care about don’t cares,” only the sets $\mathcal{C}$ are considered to specify the function f and manipulate with cubes. This approach is more computationally attractive than the redefinition of don’t cares. Note that the approach is not limited to single-output functions, it can be applied to functions with several outputs. Different methods are suggested to take advantage of specified values only, some examples include finding optimal Reed-Muller representations [19] and minimal decision diagram structures [12].

Example 3 *(Continuation of Example 1) The “don’t care about don’t cares” approach leaves the following sets for future analysis:* $\mathcal{C}_0 = \{0, 1, 6, 7, 12, 13, 14\}$ *and* $\mathcal{C}_1 = \{2, 4, 5, 8, 9\}$*. Further manipulation techniques depend on the problem and the final representation of the function.*

3 Types of decision diagrams

While there are other visualization forms such as cubes, lattices, relations, etc., graph-based structures have become the advanced tools for representing and manipulating discrete data because of their simplicity, canonical nature, and effective algorithms [15].

A Boolean function f: $\{0,1\}^n \rightarrow \{0,1\}$ can be represented by a *Binary Decision Diagram* (BDD) [2], i.e. a directed acyclic graph with node (vertex) set V where: (i) each *non-terminal* node u is labeled by a variable x and assigned as a decision variable with two successors (children) $u.left$ and $u.right$; (ii) a *terminal* node is labeled with the leaf value and has no successors.

A decision diagram is called *ordered* if the variables X appear in the same order $x_{j_1} \prec x_{j_2} \prec \ldots \prec x_{j_n}$ in each path from the root to a terminal node. Otherwise, it is called *free*. In the following, only ordered decision diagrams are considered. It is known that the order of variables can be changed to reduce the diagram size as the number of nodes. This fact and the integrated dynamic variable ordering techniques are used in diagram minimization algorithms for incompletely specified multiple-valued functions. A decision diagram can be compacted by applying multiple rules to eliminate redundant nodes.

The basic problem of a decision diagram representation for an incompletely specified function explored by many researchers [4, 18] is how to assign the set $\mathcal{DC}$ so that the size of the diagram representing the corresponding completely specified function is minimized. This problem, known to be NP-complete [16], has been addressed through a variety of methods both heuristic and exact. In addition, the variable reordering technique utilizing don't cares has been presented in [17].

3.1 Binary and multiple-valued decision diagrams

Among many types of BDDs, reduced ordered BDDs (ROBDDs) are most widely used ones in practice. For a given variable ordering, the ROBDD representation of a completely specified function is unique. For an incompletely specified function, however, many ROBDDs can be used to represent the function, each associated with a different assignment of don't cares to binary values. The problem of redefinition is reformulated to finding an assignment of don't cares that yields a small ROBDD representation. Those heuristic methods try to maximize the instances of node sharing or sibling-substitution during the minimization process. BDD nodes become shared if the reassignment of don't cares makes their associated functions identical. Sibling-substitution is a special case of node sharing where a child of a BDD node is replaced by the other child. Sibling-substitution leads to fewer nodes because a parent and its two children are replaced by the child when the two children are made identical.

There are multiple heuristics suggested to minimize BDDs/ROBDDs. Thus, a framework of sibling-substitution-based heuristics was proposed in [18]. These heuristics, specifically *restrict* and *constrain*, outperform others in terms of both run-time and resulting BDD size. Another method of assigning binary values to don't cares by traversing the BDD structure from top to bottom was outlined in [3]. Being computationally complex, it makes the sub-BDDs shared which results in overall diagram reduction. More recently, restrict and constrain heuristics were adjusted to minimize BDDs safely [5]. The idea of the *safe* BDD minimization is to perform sibling-substitution only on nodes that will not cause increase in BDD size.

3.2 Ternary decision diagrams

Ternary Decision Diagrams (TDDs) are similar to BDDs, except that each non-terminal node has three successors [14]. Having three outgoing edges, is is possible to represent Boolean functions with unspecified (third) values. This ternary structures implement Kleene functions. A *Kleene function* K is $\mathbf{T}^n \rightarrow \mathbf{T}$ over the variable set $X = \{x_1, \cdots, x_n\}$, where $\mathbf{T}$=$\{0,1,d\}$, and n is the number of variables. d denotes unknown input or output values. The Kleene function represents the behavior of logic function in the presence of unknown values. For a given two-valued logic function, the Kleene function is unique.

The Kleenean strong logic introduced in [8] is used to represent incompletely specified functions in the form of TDDs. Many TDDs manipulation techniques were adopted from BDDs. Generally, a TDD needs less space than a pair of BDDs. Moreover, BDD pairs cannot handle the unknown input or don't care output directly while a TDD can. Comparing to BDD, TDD has one more terminal node, and one more edge for non-terminal nodes. TDDs can be reduced the same way as BDDs, and a TDD usually refers to the reduced one. To obtain a TDD from a BDD, the following transformations are needed

Expansion Expand the BDD into decision tree.

Alignment Apply alignment operation recursively:

$$alignment(x, y) = \begin{cases} x \; if \; x = y \\ d \; if \; x \neq y. \end{cases}$$

Reduction Reduce the ternary decision tree to a directed acyclic graph.

TDD is canonical as well, so all the properties for canonical form still apply [9]. However, with the size explosion problem, TDDs are often too large to build.

Example 4 *The identity function for a variable x in Kleenean logic is given in Figure 2(a). Figure 2(b) depicts an example of the full TDD for an incompletely specified function $f(x_1, x_2, x_3)$. Correspondingly, the abbreviated TDD is given in Figure 2(c). The resulting expression is $f = x_1x_2 + x_3$.*

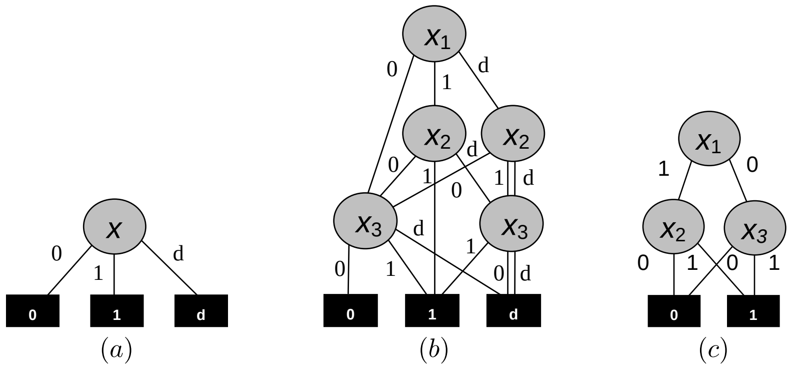

Figure 2: TDD minimization: (a) identity function; (b) full diagram; and (c) abbreviated diagram

3.3 Incompletely specified decision diagrams

The deficiencies of existing heuristic algorithms, which redefine don't cares and manipulate variable order, can be avoided using different principles and an extension of the traditional graph structure. We start by defining an incompletely specified DD using the idea "don't care about don't cares."

An *Incompletely Specified DD* Ξ is a directed acyclic graph with a node set in which each non-terminal node u has at least one successor.

A *chain* ξ is a linear incompletely specified decision diagram composed of (i) non-terminal nodes associated with each variable x of the function f, and (ii) terminal nodes with the value $f(\epsilon)$. It represents a cube ϵ as a conjunction of a set of variable values: $\epsilon \Rightarrow \xi$. Note that don't cares are excluded from consideration.

Example 5 *An example of an incompletely specified DD is shown in Figure 3(a). Thus, the corresponding function is specified on three combinations:* $\{x_1^1 x_2^0 x_3^2\}$, $\{x_1^2 x_2^0 x_3^0\}$ *and* $\{x_1^2 x_2^0 x_3^1\}$. *The order of variables in the diagram is* $\prec x_2\ x_1\ x_3 \succ$. *A chain for the cube* $\{x_2^0 x_1^2 x_3^1\}$ *from the set* $\mathcal{C}_0$ *is depicted in Figure 3(b).*

The properties of incompletely specified DDs are defined similarly to those for traditional DDs. The compactness of incompletely specified DDs is guaranteed by two rules: (i) merging that shares equal functions, and (ii) deletion that deletes a node where all r children are equal. These rules are formulated as for completely specified functions except the case of nodes with fewer successors than the radix of variables.

The variable ordering in the chain ξ is adjusted to the variable ordering in the existing diagram according to the output of dynamic variable reordering. To add the chain ξ into the current incompletely specified DD, we merge graph structures through a *fusion* operation defined below.

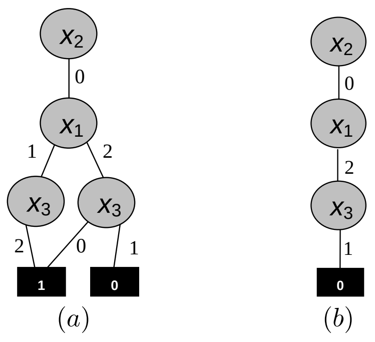

Figure 3: Incompletely specified decision diagram (a) and its chain (b)

A *Fusion Operation* $\coprod$ unites two incompletely specified DDs Ξ_1 and Ξ_2 with an identical variable ordering: $\Xi = \Xi_1 \coprod \Xi_2$. In the following, we consider the design of the incompletely specified DD Ξ for the given incompletely specified multiple-valued function f as a sequence of fusion operations on k chains ξ: $\Xi = \coprod_k \xi_k$. This process can be described iteratively as a sequence of snapshots: $\Xi_{t+1} = \Xi_t \coprod \xi_{t+1}$, where $t = 1, \ldots, k$ and $\Xi_1 = \xi_1$.

Example 6 *Let us apply the fusion operation* $\coprod$ *to the incompletely specified decision diagrams* Ξ_1 *and* Ξ_2 *shown in Figure 4(a) and (b) respectively. The top-down iterative strategy results in a diagram* Ξ *depicted in Figure 4(c).*

4 Optimization problems

Each type of decision diagrams has a set of specific optimization techniques for obtaining feasible (applicable in heuristic algorithms) or optimal (applicable in exact algorithms) solutions. These techniques are outlined for three types: BDDs and MDDs, TTDs, and Incompletely Specified DDs.

4.1 Safe BDD minimization

Let us consider the safe BDD minimization in details. An incompletely specified function is given by a pair of completely specified functions $[f, c]$, where f is a cover of the incompletely specified function and c denotes the care-function. A set of cubes f is a cover of the original function g if $C_0(g) \subseteq C_0(f)$ and $C_1(g) \subseteq C_1(f)$. Two BDDs $[F, C]$ are built for the functions $[f, c]$. The safe minimization algorithm consists of two phases:

– The *mark-edge* phase handles the preprocessing of the original BDDs $[F, C]$ identifying nodes for which applying sibling-substitution does not increase overall BDD size.

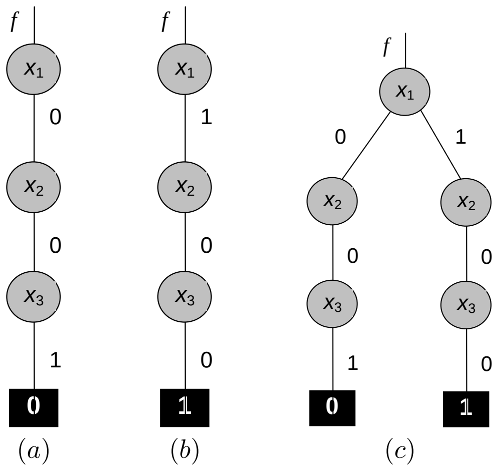

Figure 4: Fusion operation: (a) the first diagram, (b) the second diagram, and (c) the result of fusion

– In the second phase, called *build-result*, sibling-substitution is selectively applied to the nodes identified in the first phase.

Example 7 *Let an incompletely specified function $g = g(x_1, x_2, x_3)$ represented by the sets: $\mathcal{C}_0 = \{2\}, \mathcal{C}_1 = \{5, 7\}, \mathcal{DC} = \{0, 1, 3, 4, 6\}$, be given by two decision diagrams F and C (Figure 5(a) and (b) respectively). By traversing the diagrams, the sets of care values are: $\mathcal{C}_0(F) = \{0, 1, 2, 4, 6\}, \mathcal{C}_1(F) = \{3, 5, 7\}, \mathcal{C}_0(C) = \{2, 3\}, \mathcal{C}_1(C) = \{0, 1, 4, 5, 6, 7\}$. The result of the first edge-marking phase is shown in Figure 5(c). The second phase, build-result, produces the diagram in Figure 5(d).*

Two basic phases of the safe BDD minimization are outlined below.

Mark-edges recursive algorithm:

```
Step 1. Compare nodes from $F$ and $C$ for being a leaf and with 0
    respectively. If it is not a leaf-0, continue with Steps 2-4.
Step 2. Get the top variable $x$ from $F$ and $C$ which will be used as a
    substitution.
Step 3. For non-terminal nodes and the left sub-BDD, mark the edge and
    call the recursive function passing $F_{x'}$ and $C_{x'}$.
Step 4. For non-terminal nodes and the right sub-BDD, mark the edge and
    call the recursive function passing $F_x$ and $C_x$.
```

Build-result recursive algorithm:

```
Step 1. Compare nodes from $F$ for being a leaf. If it is not a leaf,
    continue with Steps 2-4.
Step 2. Get the top variable $x$ from $F$.
Step 3. If the left sub-BDD is marked and the right sub-BDD is not, call
    the recursive function passing $F_x$.
```

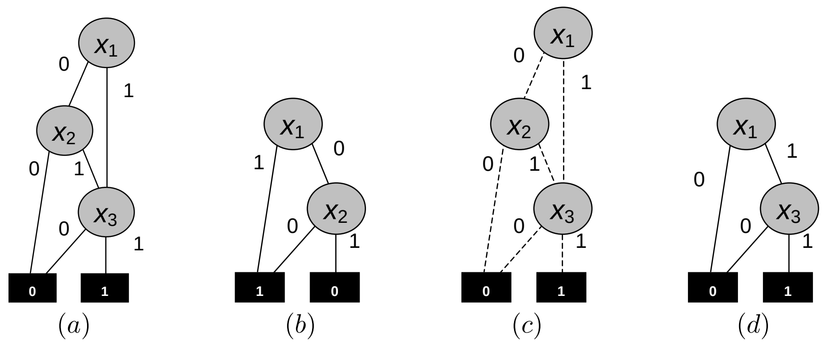

Figure 5: Safe BDD minimization: (a) F diagram; (b) C diagram; (c) edge-marked F; and (d) minimized diagram

Step 4. `If the left sub-BDD is not marked and the right sub-BDD is, call the recursive function passing` $F_{x'}$.

Recalling that a BDD minimization using don't cares is safe if the minimized BDD is guaranteed to be larger than the original BDD. The two-phase compaction algorithm is considered to be safe [5]. The result of minimization is produced by replacing some nodes with one of their descendents. It is safe because it ensures that no node will be split. This property can be deducted from the structure of build-result. It creates one node for each node it visits and visits each node at most once. Specifically, nodes that are not reachable from the root by a path of marked edges are not visited by build-result and, therefore, not included in the minimized BDD.

4.2 Some operations on TDDs

The following changes are made to the original BDD algorithms to enable manipulations with unspecified values:

Reduce The algorithm reduce is used to generate reduced TDDs from TDDs according to the reducing rules. All the nodes in a TDD are labelled with integers, then the nodes with the same label are combined. We use a bottom-up method to label the nodes one by one. A non-terminal node can not be labelled until all its branches have been completely labelled. $id(n)$ denotes the label of the node n, $0(n)$ and $1(n)$ represent the corresponding successor of the node n. The labelling method has the following sequence of steps:

Step 1. `If` $id(0(n)) = id(1(n))$, `then` $id(n) = id(0(n))$.
Step 2. `If there exists a node` m `with` $id(0(m)) = id(0(n))$ `and` $id(1(m)) = id(0(n))$, `then` $id(n) = id(m)$. `Otherwise we assign the next unused label to` $id(n)$.

Restrict The restrict operation computes a new TDD with the same variable ordering, but restrict certain variable to a given value. We use $f(x = t)$ to denote a formula obtained by replacing all the occurrences of x in f by t. The TDD for $f(x = t)$ is constructed by forcing all the edges pointed to the node associated with x to point the root of its proper edge instead. The reduce operation has to be executed afterwards.

Example 8 *An incompletely specified function is given by its TDD as shown in Figure 6(a). The second step of the reduce operation eliminates one node with common successors from the second level of the diagram minimizing the number of nodes and their interconnections (Figure 6(b)). Using the obtained diagram and assuming that $x_2 = d$, the restrict operation forces all outgoing edges from the first level to point to terminal nodes as shown in Figure 6(c).*

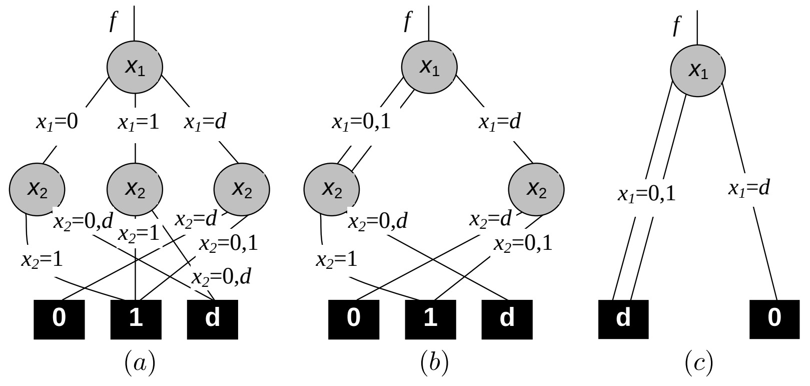

Figure 6: Ternary decision diagram operations: (a) original TDD, (b) result of reduce, and (c) result of restrict for $x_2 = d$

4.3 Incompletely specified decision diagrams

There are two basic operations on incompletely specified DDs: variable reordering and minimization. These operations support the dynamic essence of incompletely specified DDs: if the specification of incompletely specified function is updated, the structure of the diagram will be modified instantaneously.

Variable reordering Many heuristics have been proposed for finding a good variable ordering, it is evident that none of them guarantees that the solution is optimal. In this approach, we have chosen to use sifting, because of its dynamic nature. The basic idea of the sifting algorithm is to select the best position for one variable assuming that the relative order of all others remains the same.

This process is repeated for all variables, starting with variables situated in the level with the largest number of nodes. Dynamic variable ordering is integrated into the algorithm outlined below.

The method consists of the following steps:

```
Step 1. The levels are sorted according to their size. The largest level
    is considered first.
Step 2. For each variable:
   2.1 The variable is exchanged with its successor variable until it is
       the last variable in the ordering.
   2.2 The variable is exchanged with its predecessor until it is the
       topmost variable.
   2.3 The variable is moved back to the closest position which has led
       to the minimal size of the BDD or MDD, respectively.
```

Example 9 *Let us consider the incompletely specified decision diagram given in Figure 7(a) with lexicographical order of variables. The dynamic reordering exchanges variables at the first and second levels assigning x_2 to the first level and x_1 to the second level. The reordering results in a diagram shown in Figure 7(b). This diagram has fewer nodes and their interconnections, and will be more preferable as the result of optimization.*

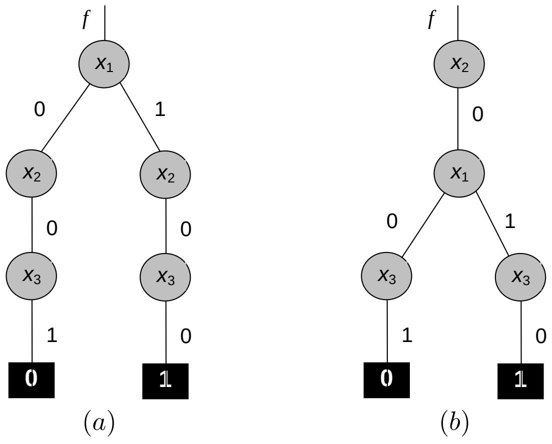

Figure 7: Incompletely specified decision diagrams: (a) before the dynamic reordering, (b) after the reordering is completed

Don't care minimization A sketch of the minimization algorithm for the function f given on the sets $\mathcal{C}$ is shown below:

```
Step 1. Initialize the variable ordering lexicographically: $x_1 \prec x_2 \prec \ldots \prec x_n$ ($x_1$ is the topmost variable), and the incompletely specified
    DD $\Xi = \oslash$.
Step 2. Consider a cube $\epsilon$ from the given set $\mathcal{C}$: $\epsilon \in \mathcal{C}$. Build a
    corresponding chain $\xi$ applying current variable ordering: $\epsilon \Rightarrow \xi$.
Step 3. Merge the chain $\xi$ with the existing incompletely specified DD $\Xi$
    applying the fusion operation: $\Xi = \Xi \coprod \xi$.
```

Step 4. Reorder the obtained incompletely specified DD Ξ applying sifting.
Step 5. Perform the compaction of the incompletely specified DD Ξ to eliminate nodes which shares equal functions, and parent nodes with equal children.
Step 6. If there are other cubes $\mathcal{C} \neq \oslash$, go to Step 2. Otherwise, terminate the algorithm and do the minimization of the incompletely specified DD Ξ.
Step 7. (Postprocessing) Perform the minimization of the incompletely specified DD Ξ to eliminate nodes with don't cares.

Example 10 *Let us consider the following incompletely specified 3-valued function $f = f(x_1, x_2, x_3)$ given on three combinations of variable values $\mathcal{C}_0 = \{x_1^2 x_2^0 x_3^1\}$, and $\mathcal{C}_1 = \{x_1^1 x_2^0 x_3^2, x_1^2 x_2^0 x_3^0\}$. The iterative process of constructing an incompletely specified MDD and its minimization is illustrated in Figure 8. The first step combines two chains $x_1^1 x_2^0 x_3^2$ and $x_1^2 x_2^0 x_3^0$ by applying the fusion operation. The post-processing reordering (sifting) swaps variables assigned to the first and second levels. The resulting diagram is combined next with the remaining chain $x_1^2 x_2^0 x_3^1$. The final reordering operation does not change the order of variables in the diagram.*

5 Concluding remarks

Recent progress in soft computing is accelerated by advances in artificial neural networks, fuzzy logic, genetic algorithms, genetic and evolutionary programming, and data mining. In different ways, these approaches try to solve complex and poorly defined problems that previously developed analytic models could not efficiently tackle. All of these approaches offer a method of automatic learning. Machine learning has become a general paradigm for software system design, unifying all these previously disconnected areas. Data, which is most commonly incompletely specified, should be visualized by efficient structures ready for knowledge interpretation. This paper gives an outline of decision diagrams for representing incomplete and uncertain dependencies.

Numerous applications can take advantage of decision diagram representations for incompletely specified functions.

Planning As planning agents grow more sophisticated, plan representation issues arise. Planners work over increasingly large and difficult problems and output is often complex. Further, where planners must interact with human users either for plan verification and analysis, or in mixed-initiative settings plans must be represented so that the intended course of action is readily available. Some techniques are proposed for simplification of, and conversion between, plan representations [1].

Embedded software Embedded systems have extremely tight realtime and code/data size constraints, that make expensive optimizations desirable. Some BDD minimization techniques are proposed in [6] in the presence of a don't care set to synthesize code for extended Finite State Machines from a BDD-based

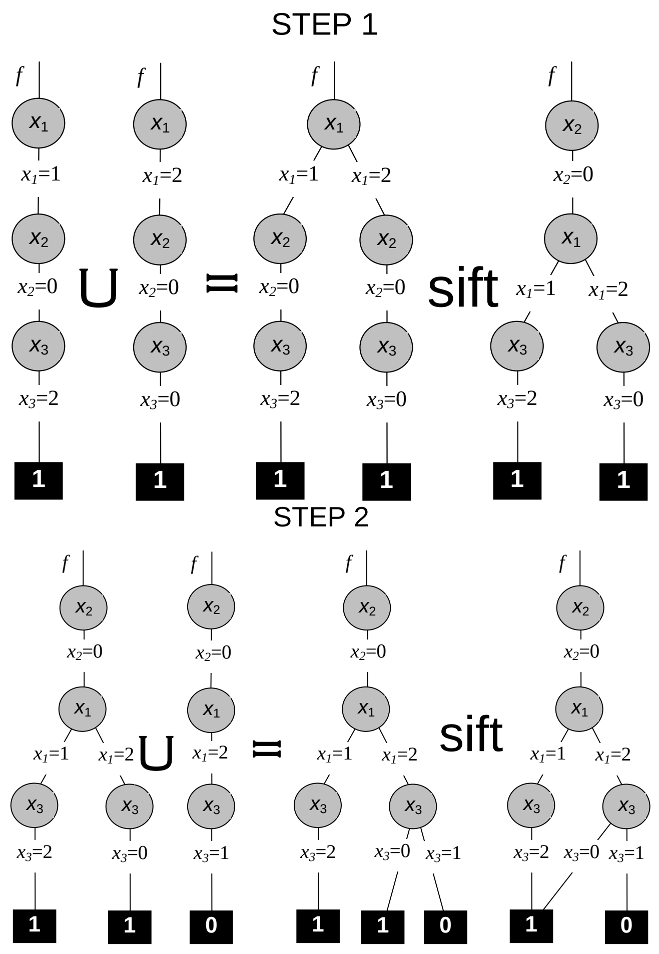

Figure 8: Multiple-valued decision diagram minimization for the function f (Example 10): Step 1 is forming the diagram from two chains $x_1^1x_2^0x_3^2$ and $x_1^2x_2^0x_3^0$, and Step 2 is adding the chain $x_1^2x_2^0x_3^1$.

representation of the FSM transition function. The don't care set can be derived from local analysis (such as unused state codes or don't care inputs) as well as from external information (such as impossible input patterns).

Compiler optimization Note that some compiler optimizations, such as variable lifetime analysis, constant value propagation can be considered a form of don't care exploitation. For example, avoiding assigning a variable that is not read before being assigned again, is exploiting a form of "observability don't cares", just as the elimination of an "if" statement with a constant condition is exploiting a form of "controllability don't cares". The software synthesis technique is based on the use of BDDs to optimally synthesize software (in particular C code) from a specification in the form of FSMs extended with integer arithmetic capabilities [6]. That technique uses a direct mapping between BDD nodes and low-level C statements in order to derive a highly optimized implementation of the FSM transition relation.

References

1. M. Allen and S. Zilberstein. Automated conversion and simplification of plan representations. In *Proc. Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 1272–1273, 2004.
2. R. Bryant. Graph - based algorithm for Boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):667–691, 1986.
3. S. Chang, M. Marek-Sadowska, and T. Hwang. Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(10):1226–1248, 1996.
4. Y. Hong, P. Beerel, J. Burch, and K. McMillan. Safe BDD minimization using don't cares. In *Proc. IEEE/ACM Int. Design Automation Conference*, pages 208–213, 1997.
5. Y. Hong, P. Beerel, J. Burch, and K. McMillan. Sibling-substitution-based BDD minimization using don't cares. *In IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(1):44–55, 2000.
6. Y. Hong, P. Beerel, L. Lavagno, and E. Sentovich. Don't care-based BDD minimization for embedded software. In *Proc. Design Automation Conference*, pages 506–509, 1998.
7. Y. Jiang and R. Brayton. Don't cares and multi-valued logic network minimization. In *Proc. IEEE/ACM Int. Conference on CAD*, pages 520–525, 2000.
8. S. Kleene. *Introduction to Metamathematics.* Van Nostrand, Princeton, NJ, U.S.A., 1964.
9. P. Lindgren. Improved computational methods and lazy evaluation of the ordered ternary decision diagrams. In *Proc. Asia and South Pacific Design Automation Conference*, pages 379–384, 1995.
10. A. Mishchenko, C. Files, M. Perkowski, B. Steinbach, and C. Dorotska. Implicit algorithms for multi-valued input support minimization. In *Proc. Int. Workshop on Boolean Problems*, pages 9–20, 2000.
11. A. Oliveira and S. Edwards. Limits of exact algorithms for inference of minimum size finite state machines. In *Algorithmic Learning Theory Workshop*, volume 1160, pages 59–66, 1996.
12. D. Popel and R. Drechsler. Efficient minimization of multiple-valued decision diagrams for incompletely specified functions. In *Proc. IEEE International Symposium on Multiple-Valued Logic*, pages 241–246, 2003.
13. D. Popel and N. Hakeem. Multiple-valued logic in decision making and knowledge discovery. Technical report, Baker University, KS, U.S.A., 2002.
14. T. Sasao. Ternary decision diagrams: Survey. In *Proc. Int. Symposium on Multiple-valued Logic*, pages 241–250, 1997.
15. T. Sasao. *Switching Theory for Logic Synthesis.* Kluwer Academic Publishers, Norwell, MA, U.S.A., 1999.
16. M. Sauerhoff and I. Wegener. On the complexity of minimizing the OBDD size for incompletely specified functions. *In IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(11):1435–1437, 1996.
17. C. Scholl, D. Möller, P. Molitor, and R. Drechsler. BDD minimization using symmetries. *In IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(2):81–100, 1999.
18. T. Shiple, R. Hojati, A. Sangiovanni-Vincentelli, and R. Brayton. Heuristic minimization of BDDs using don't cares. In *Proc. IEEE/ACM Int. Design Automation Conference*, pages 225–231, 1994.
19. A. Zakrevskij. Optimizing polynomial implementation of incompletely specified Boolean functions. In *Proc. Workshop on Application of the Reed-Muller Expansions in Circuit Design*, pages 250–256, 1995.