HDL Optimization Using Timed Decision Tables

Jian Li and Rajesh K. Gupta
Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, Illinois 61801.

Abstract

System-level *presynthesis* refers to the optimization of an input HDL description that produces an optimized HDL description suitable for subsequent synthesis tasks. In this paper, we present optimization of control flow in behavioral HDL descriptions using external Don't Care conditions. The optimizations are carried out using a tabular model of system functionality, called Timed Decision Tables or TDTs. TDT based optimization presented here have been implemented in a program called PUMPKIN. Optimization results from several examples show a reduction of 3-88% in the size of synthesized hardware circuits depending upon the external Don't Care information supplied by the user.

1 Introduction

Due to the maturity of optimization and synthesis tools at logic and register transfer level, system specification is increasingly being done at behavioral level using a Hardware Description Language (HDL), such as VHDL and Verilog. Though optimization can be done at all levels of system specification, behavioral-level HDL optimization is more efficient and saves time in lower-level transformation and verification tasks. Further more, very often the final circuit size depends upon the programming style of the HDL description. Though running the optimization targeting at minimum area may recover some fraction of the area, in general the optimized circuit obtained from the "bad" HDL description will not be as small as even the initial circuit corresponding to the "good" description [1].

Behavioral optimization algorithms in this work are implemented on a model of system functionality called timed decision tables. Decision tables originated as a simple and visual way of documenting nested conditional branches in programming languages [2]. Compared with programming languages, decision tables are more concise and canonical. Timed decision tables (TDT) are an extension of the decision tables to model hardware-related aspects of operation timing and concurrency [3].

The suitability of tabular formalisms for hardware specification has been proposed in several independent works. In [4] the authors introduced Behavior Tables to represent a finite state machine where each row corresponds to a state transition in the machine described by the table. In [5] the authors used relational algebra to show the simplicity and power of tabular formalisms. Both these tabular models are RTL-level system descriptions. In contrast, a TDT represents a behavioral model and may be translated into different RTL-level tables after synthesis.

2 Timed Decision Tables

The Timed Decision Table is based on the notions of *condition* and *action*. A condition may be the presence of an input, or an input value, or the outcome of a test condition. A conjunction of several conditions defines a *rule*. A decision table is a collection of rules that map condition conjunctions into sets of actions. Actions include logic, arithmetic, input-output(IO), and message-passing operations.

Actions are grouped in action sets. With each action set, we associate a concurrency type of serial, parallel, or data-parallel. An action set of type serial is equivalent to a compound statement in an ordinary sequential program. A parallel type indicates that no ordering among individual actions in the action set is assumed. A data-parallel type means parallel action subject to data-dependencies between actions. Each action is associated with an execution delay which may be fixed, variable or even unbounded. We assume that condition testing takes zero time. The action delay is typically used to model the data-path delay associated with an operation. Because of the timing semantics associated with actions and action sets, we name the action sets *timed action sets.*

The structure of a TDT is shown in Figure 1. It consists of four quadrants. Condition stub is the set of conditions used in building the TDT. Condition entries indicate possible conjunctions of conditions as rules. Ac-

33rd Design Automation Conference ®
Permission to make digital/hard copy of all or part of this work for personal or class-room use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permssion and/or a fee.
DAC 96 - 06/96 Las Vegas, NV, USA ©1996 ACM, Inc. 0-89791-833-9/96/0006..$3.50

Condition Stub	Condition Entries
Action Stub	Action Entries

Figure 1: Structure of timed decision tables.

tion stub is the list of action sets that may apply to a certain rule. Action entries indicate the mapping from rules to actions. A rule is a column in the entry part of the table, which consists of two halves, one in the condition entry quadrant, called decision part of the rule, one in the action entry quadrant, called action part of the rule.

There are two ways to arrange the condition stub (or the action stub) and the condition entries (or action entries). In the *limited-entry form* the stub enumerates all possible conditions (or actions) and the entry section is a Boolean matrix that selects appropriate conditions (or actions). In contrast, in an *extended-entry form*, the entries may assume a range of values.

Example 2.1. A timed decision table. Consider the following behavior description in HardwareC:

```
if  C1  {
    if C2
         a1;
    else
         a2;
}
else
         a3;
```

Assuming that action a2 takes 2 cycles, while the rest take 1 cycle, the above behavior is described by the following timed decision table:

C1	Y	Y	N
C2	Y	N	X
A	a1	a2	a3
delay	1	2	1

Condition entries in this TDT are in the limited-entry form. Condition literals 'C1' and 'C2' form the condition stub. The action entries are in extended-entry form, containing three possible values 'a1', 'a2', and 'a3' of action 'A'. A rule { 'Y', 'N', 'a2'} in column 2 represents the condition that when 'C1' evaluates to true and 'C2' evaluates to false, the action set represented by literal 'a2' is executed. The 'X' in the condition entry indicates that a condition assumes a Don't Care value, for a particular rule. □

Execution Semantics. The execution of a TDT consists of two steps: (a) select the set of rules to apply, and (b) execute the actions that the selected rules map to. Execution of actions may generate events which are actions with future time stamps. These time stamps are determined by the execution delays and scheduling of operations in an action.

Iterations, as in loop structures and processes, are modeled using *process TDTs*. A process TDT is executed repeatedly until a deadlock occurs or an explicit exit operation is executed. A TDT that is executed only once, each time it is enabled as a part of an action, is called a *procedure TDT*.

As shown in Example 2.1 above, not all conditions may be applicable to a rule. Conditions that are not applicable to a rule take a Don't Care value, indicated by an 'X' in the corresponding column. We call these Don't Cares *condition entry Don't Cares*. Similarly, not all actions may apply to a rule. Only applicable actions are indicated in the action entries of a TDT. In presence of assertions [6], a weak form of external Don't Cares, a column may never be selected for execution.

A TDT is considered *complete* when each possible combination of condition variable values is covered by a column in the table. A TDT is *consistent* when a given combination of condition variable values is covered by only one column in the table. The order of rule application is immaterial in consistent TDTs. In general, A TDT generated from HDL descriptions need not be complete or consistent. Missing columns and overlapped portion of columns can be treated as Don't Care columns, or columns that will never be enabled for execution. Indeed in our work, incompleteness and inconsistency are used to derive or specify behavioral Don't Cares.

3 TDT Optimizations

The concept of Don't Cares (DCs) has been extensively used in logic synthesis for gate-level optimizations [1, 7, 8]. For synthesis tasks on behavioral descriptions, the notion of Don't Cares is relatively new [9]. However, it is believed that a proper definition and use of DCs at higher levels of abstraction will provide a large scope for HDL optimizations. We distinguish behavioral Don't Cares from structural Don't Cares where the latter applies only to control functions for hardware implementation [10]. Here we focus on TDT optimization using behavioral Don't Cares, in particular, condition dependencies in HDL descriptions. Condition dependencies, also referred to as assertions, are weaker and safer form of Don't Cares as discussed in [6].

For a given HDL description, its TDT model consists of hierarchically related Timed Decision Tables and Action Sets. Condition dependencies are normally specified at the interface of control modules. We propagate this information to relevant timed decision tables in the system model via data-flow analysis [3]. TDT optimization is used to reduce the size of the tables using condition dependencies. Table sizes can be reduce in two ways: (a) by reducing the number of columns, and (b) by reducing the number of conditions in the tables.

TDT optimization is carried out via a series of row or column operations targeting at reducing both the number of rows and the number of columns in a TDT [3, 10]. Performing a series of row or column operations with the goal of reducing the number of columns can be reduced to a two-level logic optimization problem of finding the minimum cover of a Boolean function. For this purpose, we formulated TDT in an algebraic form [3]. For example, the TDT in Example 2.1 can be written as

$$TDT_{Ex2.1} = C_1C_2a_1 + C_1\overline{C}_2a_2 + \overline{C}_1a_3 \qquad (1)$$

where C_1, C_2 are condition literals associated with a limited-entry condition stub. Now column reduction of a TDT can be done via finding the minimum cover for the algebraic form of the TDT. We have also implemented the column reduction optimizations using a two-level logic optimizer Espresso [11]. The following shows an example.

Example 3.1. TDT Optimization.
Given Don't Care: $C1 \wedge \overline{C}2 = 0$ the TDT in Example 2.1 is transformed as shown below.

C1	Y	Y	N
C2	Y	N	X
A	a1	a2	a3

(*a*)

$\Longrightarrow$

C1	Y	Y	N
C2	Y	N	X
A	a1	X	a3

(*b*)

$\Longrightarrow$

```
.i 2
.o 2
11 01
10 --
0- 11
```

(*c*)

$\Longrightarrow$

```
.i 2
.o 2
.p 2
1- 01
0- 11
.e
```

(*d*)

$\Longrightarrow$

C1	Y	N
C2	X	X
A	a1	a3

(*e*)

$\Longrightarrow$

C1	Y	N
A	a1	a3

(*f*)

Column 2 in TDT (a) is identified as a Don't Care column. Since action entry in the Don't Care column can assume any value, we write explicitly an 'X' in that action entry in (b). With an arbitrary encoding on the action literals, TDT in (b) can be expressed in a PLA form as input for espresso as shown in (c). The result in (d) is produced by running espresso on the input in (c). TDT (e) is obtained from the PLA form in (d). Finally, since row 2 in TDT (e) contains only Don't Care values, we can drop this row and obtain TDT (f). □

The number of condition literals in a TDT can be reduced in two ways: (a) by eliminating a row with all Don't Care entries, or (b) by re-encoding conditions in the table as described in below. Additional Don't Care entries are often produced by two-level logic optimizations since prime implicants are used in the minimum cover. Row elimination can then be more effectively applied after column reduction. Condition re-encoding is used after row elimination. For a TDT with N_c columns, a minimum of $\lceil log_2 N_c \rceil$ conditions are needed. If more than $\lceil log_2 N_c \rceil$ conditions are used, the conditions can be replaced by a new set of Boolean variables of minimum size where the new set of variables are functions of original variables. This process of finding a minimum number of condition variables is referred to as condition re-encoding.

TDT merging is applied to enhance the scope of TDT optimizations. For example, two procedure TDTs can be merged together if one TDT is the action value in a rule of another TDT. An algorithm can be found in [3].

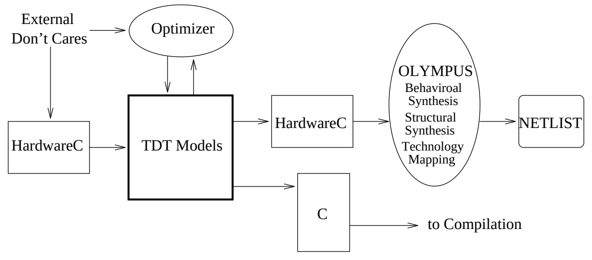

Figure 2: TDT optimizer for behavioral descriptions.

4 Implementation & Experimental Results

Merging algorithms and optimization techniques have been implemented in a program called PUMPKIN. An overview of PUMPKIN is shown in Figure 2. The PUMPKIN optimizer features a UNIX shell-like command interface to allow its interactive use by the system designer. We use HardwareC to describe system functionality. The system designer reads in a given HardwareC description and applies External Don't Care (EDC) conditions as additional input. A list of EDC conditions is maintained that can be updated as the modeling and synthesis of other blocks, with which the system interacts, proceeds.

To evaluate the effectiveness of TDT-based optimizations, we have experimented with several high-level synthesis benchmark designs. Our experimental methodology is as follows. The HDL description is compiled into TDT models, run through the optimizations, and finally output as a HardwareC description. This output is provided to the Olympus High-Level Synthesis System [12] for hardware synthesis under minimum area objectives. We use Olympus synthesis results to compare the effect of optimizations on hardware size on HDL descriptions. Hardware synthesis was performed for the target technology of LSI Logic 10K library of gates. Results are compared for final circuits sizes, in terms of numbers of cells used.

Table 2 describes results of TDT based optimization on example designs. Description 'gcd' models a hardware module that repeatedly samples the input on the rising edge of a control signal, then computes the greatest common divisor of two input values using Euclid's algorithm. Description 'trolley/motorcntrl' refers to the motor controlling module in a trolley controller used to transport assembly components on a shop-floor. Description 'ecc/decoder' decodes parity encoded data transmitted through a serial line and correct transmission errors. All the designs are from the high-level synthesis benchmark suite [12].

External condition dependencies can be classified in two categories: (a) assertions that are not part of HDL specifications, and (b) conditions which are inherent in

design	circuit size (cells)		Don't Care condition	$\Delta\%$
	before	after		
gcd	272	230	positive inputs	15
trolley/ motorcntrl	2952	366	no position variation	88
trolley/ motorcntrl	2952	2774	there is always variation	7
comm/ enqueue	234	209	`Bwr` $\wedge$ `Brd = 0` (write or read)	11
cruiser/ State	361	350	`in OFF_STATE` $\rightarrow$ `Control=1|3|9`	3
ecc/ decoder	141	119	correct only single error	16
parker86	384	270	in2 + in3 = 0	29
div8	174	85	positive input	51
traffic	35	34	no external dc specified	3

Table 1: Synthesis results: cell counts before and after language level optimization.

the behavior specification but are either hard to take into account while writing descriptions or make the descriptions harder to read if directly incorporated. For example, two different external conditions are considered for the design 'motorcntrl'. The first one, no position variation, is based on the assumption that the trolley is to be used in an application which provides guiding trails. A significant reduction in circuit size is obtained because the original description contains a large number of operations to handle the deviation of the trolley. The second condition, that there is always variation, assumes the same environment and produces a slightly different model of the system which results in a final circuit with the same functionality and less area. The error correction modeled in 'ecc' considered only single errors. However, this information can not be easily used for optimization unless explicitly stated as an external Don't Care.

As shown in Table 2, the use of external conditions can lead to reductions in the size of the hardware circuits. Further, HDL level presynthesis using PUMPKIN makes it possible to build a portable library of HDL models that can be instantiated into specification application domains. It should be noted, however, that the reduction in size ranging from 3-88% is neither typical nor representative. The actual reduction clearly depends upon the external Don't Care information used in presynthesis. The chief contribution of this work is to build a framework for source-level HDL optimizations. This enables the system designer to specify additional information about system environment and use it for system optimizations. As a consequence, it makes the most general purpose modules easily reusable in a similar but different environment. It also makes the results of the synthesis process relatively insensitive to the HDL coding-style of the system designer, hence reduces the time designers spend in behavioral specifications. Moreover, a description with external Don't Cares explicitly specified can be more readable than a detailed description incorporating these Don't Care conditions as part of the system behavior.

5 Conclusion

We have presented the TDT model for behavioral specification and optimization techniques. This model is used to incorporate the external Don't Care information in system optimization. The Don't Care information can be supplied by the user as a part of the input HDL description or entered interactively as assertions of control variables used in the behavioral descriptions.

6 Acknowledgments

This research is sponsored by a grant from NSF Career Award No. 95-01615 and a grant from the NSF Engineering Research Center Grant ECD 89-43166.

References

[1] S. Devadas, A. Ghosh, and K. Keutzer, *Logic Synthesis*. McGraw-Hill, 1994.

[2] P. J. H. King, "Decision tables," *The Computer Journal*, vol. 10, no. 2, August 1967.

[3] J. Li and R. K. Gupta, "Timed decision tables: A model for system representation and optimization," technical report, University of Illinois, 1995.

[4] K. Rath, M. E. Tuna, and S. D. Johnson, "Behavior Tables: A basis for system representation and transformational system synthesis," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1993.

[5] A. J. W. M. ten Berg, C. Huijs, and T. Krol, "Relational algebra as formalism for hardware design," *Microprocessing and Microprogramming*, 1993.

[6] D. Brand, R. A. Bergamaschi, and L. Stok, "Be Careful with Don't Cares," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 83–86, 1995.

[7] S. Muroga, *Logic Design and Switching Theory*. John Wiley, 1979.

[8] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[9] R. A. Bergamaschi, "Control optimization in high-level synthesis using behavioral don't cares," in *Proceedings of the Design Automation Conference*, pp. 657–661, 1992.

[10] R. K. Gupta and J. Li, "Control optimization using behavioral don't cares," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1996.

[11] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algrotithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[12] G. D. Micheli, D. C. Ku, F. Mailhot, and T. Truong, "The Olympus Synthesis System for Digital Design," *IEEE Design and Test Magazine*, pp. 37–53, Oct. 1990.