System Modeling and Presynthesis Using Timed Decision Tables

†Jian Li and ‡Rajesh K. Gupta

†Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

‡Information & Computer Science
University of California, Irvine
Irvine, CA 92697

Abstract

In this paper, we present a tabular model of system behavior called Timed Decision Table (TDT). The TDT model is useful for identifying control-data interaction and in performing control-oriented optimizations. TDTs provide an ideal vehicle to implement source-level optimizations on a given behavioral description in a procedural hardware description language (HDL). These optimizations are used to produce improved synthesis results by simplifying the HDL models using Don't Cares or assertions in particular.

TDT also provides a convenient data structure for extracting information that can be used in further synthesis subtasks to obtain improved synthesis results. One example of this is the information on mutual exclusiveness between a pair of operations, which can be used to optimize operation scheduling.

Source-level control-flow optimization and analysis which extracts useful information from input HDL source for optimization in synthesis process are collectively referred to as **presynthesis.** *We have implemented TDT-based presynthesis techniques in a program called* PUMPKIN. *Our experiments running* PUMPKIN *on named benchmarks shows improved synthesis results after presynthesis has been carried out on the input HDL descriptions and information extracted in presynthesis has been incorporated in the synthesis optimizations.*

1 Introduction

Due to the maturity of optimization and synthesis tools at logic and register transfer levels, system specification is increasingly being done at the behavioral level using Hardware Description Languages (HDL). Behavioral descriptions in HDLs are similar to software (textual) programs in high-level programming languages. A program-like description of hardware is compiled into a formal representation on which various synthesis tasks are carried out.

Traditionally graph models such as variants of data-flow graphs (DFGs) and control-data flow graphs (CDFGs) [1, 2, 3, 4, 5, 6] have been used for various software compilation or hardware synthesis tasks. Most DFG-based representations are efficient in implementing data-oriented transformations, but often inadequate in control-flow modeling. In most CDFG models, the control-flow is embedded in the modeling hierarchy that limits the scope of control-oriented transformations [7]. To address this problem, algebraic models such as Control-flow Expression (CFE) [8] have been proposed for control-oriented optimizations. The CFE is based on process algebra and regular expressions. It is useful in identifying synchronization relationships between concurrent processes and minimizing process interactions. However, CFEs have little information about actions and thus make restrictive assumptions on control and data interactions. This limits the quality and scope of possible optimizations on the HDL input.

Tabular models have been used for hardware modeling at different abstraction levels [9, 10, 11]. In [9] the authors model hardware as a set of functions each of which is implemented as a table. Relational algebra is used to manipulate hardware as a set of tables. In [10] the authors have proposed "behavior tables" as a register transfer language using finite state machine model. These tables have been used to evaluate control versus data tradeoff while preserving cycle-accurate (RTL) behavior. This is accomplished by using an "indirection transformation". For instance, a state indirection puts the state control variable into a register thus reducing many state-related control operations into a single register operation [10]. Behavior tables have also been used to describe interface behavior [12].

In this paper, we present a model called Timed Decision Tables (TDT). Our original idea on TDT came from decision tables [13, 14]. We have added features such as timing semantics for the purpose of hardware modeling and followed a modeling methodology which is suitable for modeling general control-flows in behavioral HDL descriptions.

The TDT model provides an ideal vehicle for presynthesis. Presynthesis works on source-level HDL descriptions. It is a pre-processing process performed on input behavioral HDL source code before any synthesis tasks are carried out. Presynthesis does two things: (I) source-level optimization which removes control-flow redundancies in the behavioral code using assertions [11], and searches for similar operations that can be shared, (II) analysis which extracts useful information that helps in synthesis optimizations.

Source-level code optimization which removes specification redundancies and searches for sharable code segments is also referred to as *presynthesis optimization.* Presynthesis optimization is useful for many reasons. First, as we mentioned earlier, system specification is increasingly being done at the behavioral level using HDLs. Specific optimizations are needed for both efficient synthesis of HDL description into hardware and software compilation of HDL description into directly executable binary code that can be executed by a general purpose processor. Hardware specific optimizations concern the degree of simultaneity of operations as it affects scheduling and resource allocation steps, while control-flow optimizations are required for both hardware and software. Sec-

ond, a hardware module usually operates in a very well-defined external environment, on which Don't Cares or assertions in particular can be derived and used to get efficient designs. Third, in hardware synthesis optimization at the source level can save time in tasks at lower levels of the synthesis process since design details grow in size as the abstraction level goes lower.

One example of information extracted for use in synthesis optimization is the mutual exclusiveness of operation pairs. It is known that scheduling and binding are two interrelated problems in high-level synthesis and that the goals of reducing the total delay and reducing the resource usage are often conflicting [7]. However, operations can share resource without increasing the total delay if they are mutually exclusive (m.e.). Therefore the detection of m.e. operations becomes important in obtaining optimal scheduling/binding solution, and in particular becomes important in obtaining the optimal operation schedule when there are constraints on resource usage [15, 16, 17].

In this paper, we discuss the TDT model and demonstrate its utility in presynthesis. The rest of the paper is organized as follows. In Section 2, we present the TDT model, along with a discussion of Don't Cares in the context of TDT. We show how limited exceptions can be conveniently modeled in the TDT representation. In Section 3, we present basic behavior-preserving TDT transformations. Section 4 discusses TDT presynthesis techniques based on those transformations. Section 5 discusses the PUMPKIN presynthesis tool and shows how presynthesis can also be applied to behavioral descriptions with exceptions. We discuss experimental results in Section 6 and conclude in Section 7.

2 The TDT Representation of System Behaviors

This section presents the TDT model. We first introduce the basic definitions along with the execution semantics of TDTs. We then discuss Don't Cares in the context of TDT, followed by a two-level representation of TDTs. Finally, we show how limited exceptions can be conveniently modeled in TDT.

2.1 The Basics of TDT

The TDT model 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 into action sets, or compound actions. 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 is assumed in the action set. This models the concurrency inherent in hardware models. 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 operations.

Condition Stub	Condition Entries
Action Stub	Action Entries

Figure 1: Basic structure of timed decision tables.

The basic 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. Action stub is the list of actions 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 contains a list of possible conditions (or actions) and the entry section is a matrix with binary values that selects appropriate conditions (or actions). In contrast, in an *extended-entry form*, the entries may assume a range of values. In below, we give two examples. Example 2.1 shows a TDT representation with its action part arranged in an extended-entry form. Example 2.2 shows the same behavior as a TDT with its action part arranged in limited-entry form.

We prefer the limited-entry form over the extended-entry form for two reasons. First, in limited-entry form, action sets can be shared among rules. Second, the limited-entry form is useful to describe more general control structures as discussed later. In the rest of this paper, we will focus on the limited-entry form and only use the actions in extended-entry form as a short-hand for the cases where each action is selected by only one column.

Example 2.1. 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 condition. □

Example 2.2. The action part of the above TDT can be arranged in limited-entry form as shown below:

C1	Y	Y	N	
C2	Y	N	X	**delay**
a1	1	0	0	1
a2	0	1	0	2
a3	0	0	1	1

A '1' in the action entries indicates that the action set in the corresponding row will be executed when the rule in the corresponding column is selected. In contrast, a '0' in the action entries indicates that the action set in the corresponding row will not be executed when the rule in the corresponding column is selected. □

Execution Semantics of TDT. The execution of a TDT consists of two steps: (I) select the set of rules to apply, and (II) 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.

More than one action sets can be executed in Step 2 when one rule is selected for execution. The order of execution in such case depends on the concurrency type, data dependency, and serialization relation specified between action sets in the action stub.

The concurrency type between two action sets can be serial, parallel, or data-parallel. When a concurrency type of parallel is specified between two action sets, they are invoked simultaneously if both are selected for one rule. When a concurrency type of serial is specified between two action sets, a serialization order also needs to be specified between these two actions. Normally, we use the order in which these actions appear in the action stub unless the order is otherwise specified. Execution of the two action sets follows the serialization order if both action sets are selected for execution in one TDT rule. When a concurrency type of data-parallel is specified between two action sets, either a serialization order or a data-dependence relation may be specified to determine the order of execution when both action sets are selected in one TDT rule. If neither data dependency nor serialization order is specified, the two actions may be run in parallel. Concurrency type, data dependency, and serialization relation between any two operations can be specified in an auxiliary table as shown in Example 2.3.

Example 2.3. Consider the following TDT where more than one action sets are selected for each rule.

						C1	Y	Y	N	
a11	a12	a21	a22	a31	a32	C2	Y	N	X	delay
-	s+	m	m	m	m	a11	1	0	0	1
s-	-	m	m	m	m	a12	1	0	0	3
m	m	-	p	m	m	a21	0	1	0	4
m	m	p	-	m	m	a22	0	1	0	2
m	m	m	m	-	d-	a31	0	0	1	6
m	m	m	m	d+	-	a32	0	0	1	1

The meaning of each symbol is explained in Table 1. The colume 'delay' lists the number of cycles each action set takes to execute. Note that in TDTs in limited-entry form, each delay value is associated with each action set in the same row. This TDT can be re-written in HardwareC as shown in below.

```
if C1 {
    if C2
        [ a11; a12; ]
    else
        < a21; a22; >
    }
else
        { a31; a32; }    /* a32 consumes data produced in a31 */
```

Note that in HardwareC, a pair of squared brackets ('[' and ']') indicate a concurrency type of serial, a pair of arrowed brackets ('<' and '>') indicate a concurrency type of parallel, and that a pair of braces ('{' and '}') indicate a concurrency type of data-parallel. □

Table 1: Symbols used in the data-flow sub-table.

m	*ras* and *cas* appear in mutually exclusive path
p	*ras* and *cas* runs in parallel if both selected for one rule
s+	serial, *ras* should be executed before *cas* if both are selected for a rule
s-	serial, *ras* should be executed after *cas* if both are selected for a rule
d+	data-parallel *ras* feeds data to *cas*
d-	data-parallel *ras* consumes data from *cas*
d0	data-parallel no data dependence

ras(row action set) : the action set in this row of the auxiliary table

cas(column action set): the action set in this column of the auxiliary table

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*. Procedure TDTs are used to model function calls in the HDL descriptions. This calling semantics also distinguishes TDTs from behavior tables that represent only concurrent processes.

2.2 Don't Cares in TDT

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.* Not all column may ever be selected for execution. Columns never selected for execution are called *Don't Care columns.*

The ON-set, OFF-set, and DC-set of an operation can be described as follows.

Definition 2.1 *(**Condition Vector**): A column in the condition part of a TDT represents a* **condition vector** *as an assignment of values to the condition variables of the TDT.*

Definition 2.2 *(**ON-set**): The set of condition vectors in a TDT τ that selects an operation $\mathcal{O}$ for execution is called the* **ON-set** *of $\mathcal{O}$ in τ.*

Definition 2.3 *(**OFF-set**): The set of condition vectors in a TDT τ that does not select operation $\mathcal{O}$ for execution is called the* **OFF-set** *of $\mathcal{O}$ in τ.*

Definition 2.4 *(**Care Set**): The* **care set** *is the union of ON&OFF sets.*

Definition 2.5 *(**DC-set**): The* **DC-set** *is the complement of the care set.*

The DC-set of operation $\mathcal{O}$ depends on the assertions either explicitly specified on the environment of a system by the designer or extracted from the data-flow. An assertion is a special case of Don't Cares and it can be expressed as a function of conditions that always evaluates to true [18]. In the context of the TDT representation, an assertion is a function over condition variables. A column in the condition part of a TDT represents an assignment of variables. If such an assignment satisfies this assertion, we say that this column (or corresponding rule) satisfies this assertion. If a column does not satisfy an assertion, we say that each operation takes a Don't Care value in this column, and the corresponding condition vector belongs to the DC-set of each operation. A column that does not satisfy an assertion will never be selected for execution, and is therefore also a Don't Care column.

Example 2.4. Consider the following TDT:

c_1	Y	Y	N
c_2	Y	N	X
o_1	1	0	1
o_2	0	1	1

Then the ON-set of o_1 is $c_1c_2 + \overline{c_1}$. The OFF-set of o_1 is $c_1\overline{c_2}$. The ON-set of o_2 is $c_1\overline{c_2} + \overline{c_1}$. The OFF-set of o_2 is c_1c_2. The DC-sets of both operations are empty sets. □

Example 2.5. Now given the assertion, $f(c_1, c_2) = C_1$, the third column does not satisfy this assertion. Then it can be considered a Don't Care column indicating all action sets as Don't Cares.

c_1	Y	Y	N
c_2	Y	N	X
o_1	1	0	-
o_2	0	1	-

In this case, the ON-set of o_1 is c_1c_2. The OFF-set of o_1 is $c_1\overline{c_2}$. The ON-set of o_2 is $c_1\overline{c_2}$. The OFF-set of o_2 is c_1c_2. The DC-sets of both operations are $\overline{c_1}$. We also say that o_1 and o_2 take Don't Care values when c_1 is false. □

2.3 A Two-level Representation of TDT

The logical relationships expressed in limited entry TDTs can be expressed algebraically using Boolean variables. Before we present the two-level logic expression for TDTs, we introduce some definitions.

Definition 2.6 *(**Action Enabling Vector**): An **action enabling vector** is a vector where each element indicates the condition under which the corresponding operation is enabled for execution. The action part of each column in a TDT corresponds to an action enabling vector each element of which is either '0', '1', or a Don't Care value indicated by a '-'.*

Definition 2.7 *(**OR Operation between Two Action Enabling Vectors**): Given two action enabling vectors $\vec{A}_1 = (ae_{1,1}, ae_{1,2}, \cdots ae_{1,n})$, and $\vec{A}_2 = (ae_{2,1}, ae_{2,2}, \cdots ae_{2,n})$, where n is the number of operations in action enabling vectors, we have $\vec{A}_1 + \vec{A}_2 \equiv (ae_{1,1} + ae_{2,1}, ae_{1,2} + ae_{2,2}, \cdots ae_{1,n} + ae_{2,n})$ where '$\equiv$' indicates "is defined as", and we use a '+' to denote both the logic OR operation and the OR operation defined here.*

Definition 2.8 *(**AND Operation between a Boolean Variable and an Action Enabling Vector**): Given a condition variable c and an action enabling vector $\vec{A} = (ae_1, ae_2, \cdots ae_n)$, where n is the number of operations in $\vec{A}$, then $c \cdot \vec{A} = (c \cdot ae_1, c \cdot ae_2, \cdots c \cdot ae_n)$ where $\equiv$ indicate "is defined as", and we use a '$\cdot$' to denote both the logic AND operation and the AND operation defined here.*

Note that the OR operation as defined above is associative and commutative and that the AND operation as defined above is commutative. These properties of the newly defined operations can be easily proved by following basic definitions.

Now we present the two-level algebraic representation for the TDTs. As mentioned earlier, a TDT can be viewed as a collection of rules, each of which is determined by a conjunction of several conditions. The conjunction of conditions can be expressed as follows

$$s_j = b_{1,j} \cdot b_{2,j} \cdots b_{nc,j} \quad (1)$$

$$= \prod_{i=1}^{nc} b_{i,j} \quad (2)$$

$$b_{i,j} = \begin{cases} c_i, \text{when } ce_{i,j} = Y \\ \overline{c}_i, \text{when } ce_{i,j} = N \\ 1, \text{when } ce_{i,j} = X \end{cases} \quad (3)$$

where '$\cdot$' represents the Boolean AND operation, nc is the number of conditions or number of rows in the decision section, c_1 through c_{nc} are condition literals in the condition stub, $ce_{i,j}$ is the condition entry value at row i and column j which is either a true (Y) or false (N) value or a Don't Care value (X). With this representation of conjunction of conditions, each rule, or each column in a TDT, can be represented by

$$r_j = s_j \cdot \vec{A}_j \quad (4)$$

where '$\cdot$' is the AND operation defined in Definition 2.8, $(\vec{A}_j)_i = ae_{i,j}$ and $ae_{i,j}$ is the action entry value at row i and column j. Finally, the TDT can be written as

$$TDT = \sum_{j=1}^{nc} r_j \quad (5)$$

$$= \sum_{j=1}^{nc} s_j \cdot \vec{A}_j \quad (6)$$

$$= s_1 \cdot \vec{A}_1 + s_2 \cdot \vec{A}_2 + \cdots s_{nr} \cdot \vec{A}_{nr} \quad (7)$$

$$= b_{1,1} \cdot b_{2,1} \cdots b_{nc,1} \cdot \vec{A}_1 + b_{1,2} \cdot b_{2,2} \cdots b_{nc,2} \cdot \vec{A}_2 + \cdots b_{1,nr} \cdot b_{2,nr} \cdots b_{nc,nr} \cdot \vec{A}_{nr} \quad (8)$$

where '$+$' indicates the OR operation defined in Definition 2.7, nr is the number of rules or number of columns in the TDT.

Example 2.6. Consider the TDT in Example 2.1. It can be put into an algebraic form as

$$TDT_{Ex2.1} = \sum_{j=1}^{4} r_j = c_1 c_2 (1,0,0) + c_1 \overline{c}_2 (0,1,0) + \overline{c}_1 (0,0,1)$$

Note that for brevity, we have omitted the '$\cdot$'s in the above expression. Assuming $c_1 = 0$, $TDT_{Ex2.1}$ evaluates to $(0,0,1)$. This indicates that only a_3 will be selected for execution whenever c_1 is false. □

2.4 Modeling Exceptions in TDT

Behavioral descriptions in HDLs use control-flow constructs such as conditional branches and loops, which are also commonly used in programming languages. In addition to the normal control flow constructs such as `if` statement or `while` loop, some HDLs also support mechanisms for exception handling. There are two kinds of exceptions: (I) immediate transfer of control such as a `goto` statement inside a loop that transfer control out of this loop on exceptional conditions, (II) interruption, in which the interrupted control flow is resumed after exception processing is completed. The exception handling is frequently used for concurrent system simulation, although

its use in synthesis has been very limited. Using TDT, it is possible to model control transfers described above as limited exceptions that can be used in later synthesis tasks.

As an example of exception modeling mechanism, consider the Verilog `disable` statement. Verilog disable statements are defined on named blocks [19]. A Verilog block is defined by a pair of `begin` and `end` and groups together one or more behavioral statements. Verilog blocks can be assigned names. A Verilog block with a name is called a *named block*. A `disable` statement is defined within a named block. Semantically, a disable statement on a named block is equivalent to a `goto` to the end of this block. The `disable` statements are used to break out of a loop or nested branches or to continue executing with the next iteration of the loop. A `disable` statement in multiple processes is used to model interruption. In this paper, we focus on modeling control exceptions within a single process. We give one example in below. Since each column in a TDT represents a control path and all action sets are presented, modeling a control jump in a TDT simply requires a correct selection of action sets in each path.

Example 2.7. Consider the Verilog description in Example 2.7 showing a named block and a `disable` statement.

```
begin: blockA
    if (A)
      begin
        if(B)
          begin
              ABC;
              disable blockA;
          end

        XYZ;
      end

    FGH;
end;
```

Here the outermost block is named `blockA`. The statement "`disable blockA;`" breaks out of the nested branches. It can be modeled using a TDT with action stub and action entries in limited-entry form as below:

A	Y	Y	N
B	N	Y	X
ABC	0	1	0
XYZ	1	0	0
FGH	1	0	1

The conditions 'A' and 'B' are taken from the condition expressions in the original Verilog description. There are three action sets 'ABC', 'XYZ', and 'FGH'. □

3 Behavior-Preserving Transformation on TDTs

In this section, we introduce a set of behavior preserving TDT transformations. By "behavior preserving" we mean that, given any input sequence, the TDT models before and after transformations

produce identical output sequences. Since behavioral TDT models are un-scheduled, the exact cycle time of the output sequence depends on the scheduling decisions made by schedulers. Based on these behavior-preserving transformations, we have constructed several optimization schemes targeting at minimizing the size of TDTs and removing specification redundancies in TDT models, which will eventually lead to size reduction in the final synthesized circuits.

3.1 Transformation on a Single TDT

Now we list a set of behavior preserving transformations on a single TDT. These transformations can be classified in three categories: (I) column operations, (II) row operations in the condition part, and (III) row operations in the action part.

3.1.1 Column Operations

Column operations are performed on columns, or rules, of a TDT. Three column operations may be applied: *column merging*, *column elimination*, and *action entry modification*. The three operations refer to merging two columns in a TDT, removing a column from a TDT, and changing action entries in a column of a TDT respectively. These operations may be applied to a TDT without changing the specified behavior under certain conditions. In below, we list the conditions under which each operation can be safely applied.

(a) *column merging*: Two columns are merged under two conditions: (I) the condition entries of the two columns differ in only one row, and (II) the action entries of the two columns are identical.

(b) *column elimination*: A Don't Care column may be eliminated.

(c) *action entry modification*: A Don't Care column will never be selected for execution. It is, therefore, safe to modify the action entries of a Don't Care column without changing the behavior. This transformation is typically performed to make other transformations applicable.

3.1.2 Row Operations in the Condition Part

Each row in the condition part of a TDT, including the condition and its corresponding row of condition entries, is referred to as a condition row. Row operations in condition part apply to condition rows. The following operations may be applied: *row elimination*, *row insertion*, *row negation*, *row encoding*, and *row swapping*. In the following, we list each operation along with the condition under which the operation may be safely applied without changing the behavior of a TDT.

(a) *row elimination*: A condition row may be eliminated if each condition entry assumes a Don't Care value.

(b) *row insertion*: A condition row with all Don't Care values in its condition entries may be added without changing the behavior of a TDT.

(c) *row negation:* A condition may be negated if its entry values are merged accordingly at the same time.

(d) *row splitting:* A condition in the form of a logic expression may be split into two rows that preserves the logical relationship between conditions and actions. The procedure of this splitting is outlined in Algorithm 3.1 below.

Algorithm 3.1 Algorithm for Row Splitting.

---

rowSplit(*tdt*, *cond*)

1. Factor *cond* = **<expr1> <op> <expr2>**;
2. Replace *cond* row with two rows corresponding to **<exp1>** and **<exp2>**;
3. **foreach** condition entry **in** row *cond* **do**
4. Replace this condition entry with all possible value combinations of **<exp1>** and **<exp2>** s.t. **<expr1> <op> <expr2>** gives the value of this condition entry;
5. Duplicate condition entries and action entries in other rows accordingly if more than one columns result in Step 4;
6. **end foreach.**

Notice that we have dropped details of the data structure to focus on giving a big picture to the readers. In an actual implementation, Step 4 is realized via checking the truth table of operator op. The running time does however depends on the data structure. Here we give an estimation based on our implementation. Step 1 takes constant time. Step 2 also takes constant time if the conditions are implemented as a double linked list as in our implementation. In the worst case, all condition entries and action entries need to be duplicated. This could cost $O(R(A+C))$ in Step 3-6, where R is the number of columns, or rules, in *tdt*, A is the number of action sets, and C is the number of conditions.

(e) *row encoding*: Conditions can be encoded to reduce the number of rows in the condition part, and hence the amount of condition checking in behavioral descriptions. 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 where the new set of variables are functions of original variables.

(f) *row swapping:* Two condition rows can always be safely swapped.

3.1.3 Row Operations in the Action Part

Each row in the action part of a TDT, including the action set in the action stub, and its corresponding row of action entries, is referred to as an action row. Row operations in the action part apply to action rows. Three operations may be applied: *row elimination*, *row merging*, and *row swapping*. We give in below an explanation of each operation along with the condition under which each of the operations may be safely applied.

(a) *row merging:* Merge rows in the action part with identical action sets. If we find two identical action sets in the action stub, we merge the two corresponding rows in two steps: (I) merge action entries into one row, each column of which should be marked as selected ('1') in the row if the same column of any original row is marked as selected, (II) modify the data dependence relations so that the shared copy will inherit all the original relations specified on the action stub. Note that action row merging is valid only if every copy of the identical action set has the same concurrency types relations in relation to other action sets.

(b) *row elimination:* An action row which does not apply to any rules may be safely eliminated.

(c) *row swapping*: Two action rows may be swapped if a concurrency type of data-parallel is specified and there is no data dependency specified between the two action sets. Two action rows may also be swapped if a concurrency type of parallel is specified.

3.2 TDT Merging

Transformation can also be performed on more than one TDTs or on a TDT with preceding/following action set via TDT merging. Merging is used to create a more concise TDT representation. In addition, it also increases the scope for TDT optimizations which are based on transformations on single TDTs. Leaf TDTs are merged by recursively identifying and applying one of the following merging cases. Three basic cases are possible: (I) merging TDTs in a sequence, (II) merging TDTs in a hierarchy, and (III) merging a TDT with a following or preceding action set. In this paper, we focus our discussion on merging that involves only procedure TDTs.

3.2.1 Merging TDTs in a Sequence

Two procedure TDTs in a sequence can be merged if (I) they appear in an enclosing action set of concurrency type data-parallel, and (II) they share no columns except Don't Care columns or

columns that contain no action sets. Merging two TDTs as described here result in a TDT which contains the union of the columns in the two original TDTs if the two condition stubs are identical. If the condition stubs in the two TDTs are different, transformations are needed to first change the condition stubs into an identical form. Four transformations on the condition rows are typically applied for this purpose: row insertion, row splitting, row negation, and row swapping. We give one example in the following.

Example 3.1. Consider a TDT sequence $\{TDT_2;TDT_3\}$ where

TDT_2 =

T1	Y	N
x	X	N
A	$+_4$	$+_5$

TDT_3 =

!T1 && x	Y	N
A	$+_6$	

Before merging, we perform row splitting to convert TDT_3 to TDT_3' and then row negating to convertTDT_3' to TDT_3'' as shown in below.

TDT_3' =

!T1	Y	Y	N
x	Y	N	X
A	$+_6$		

TDT_3'' =

T1	N	N	Y
x	Y	N	X
A	$+_6$		

TDT_2 and TDT_3'' can then be merged into TDT_m where

TDT_m =

T1	Y	N	N
x	X	N	Y
A	$+_4$	$+_5$	$+_6$

□

3.2.2 Merging TDTs in a Hierarchy

Procedure TDTs in a hierarchy result from nested branches in behavioral HDL descriptions. In below, we present an algorithm for merging two procedure TDTs in a hierarchy. We refer to the outermost TDT as *calling TDT* and the inner TDT in the hierarchy as the *called TDT*. The called TDT is an action of an action set in a rule of the calling TDT.

Algorithm 3.2 Algorithm for Merging Two Procedure TDTs

---

merge(tdt1, tdt2, i, j)
1. `/* preconditions:`
2. `* tdt2 is listed as the ith action set of tdt1`
3. `* tdt1 calls tdt2 in its jth rule`
4. `*/`
5. Duplicate the ith action row into $A(tdt2)$ rows;
6. Duplicate the jth column into $R(tdt2)$ columns;
7. Append condition rows of $tdt2$ to the condition part of $tdt1$;
8. Copy condition entries of $tdt2$ over starting at position $(C(tdt2), j)$;
9. Copy action entries of $tdt2$ over starting at position (i, j);
10. **foreach** *cond*1 in $tdt1$ and *cond*2 in $tdt2$ **do**

```
11.     if cond2 = !cond1 then
12.         Negate the condition row of cond2;
13.     if cond2 = cond1 then
14.         Remove the column where condition entries of cond1 an cond2 do not agree;
15.         Replace condition rows cond1 and cond2 with their intersection;
16. end foreach.
```

Algorithm 3.2 presents a procedure which keeps modifying $tdt1$ to obtain the resulting procedure TDT. Line 1-4 state the preconditions of this algorithm, or conditions that must hold before this *merge* procedure is called. The functions C(tdt), A(tdt), and R(tdt) return the number of conditions, the number of action sets, and the number of rules in tdt respectively. Line 8 copies the condition entries of $tdt2$ over $\{CE(l,m)|C(tdt1) \leq l \leq C(tdt1) + C(tdt2) - 1, j \leq m \leq j + R(tdt2) - 1\}$, where $CE(l,m)$ is the condition entry at the lth row and the mth column. Line 9 copies action entries of $tdt2$ over $\{AE(l,m)|i \leq l \leq i + A(tdt2) - 1, j \leq m \leq j + R(tdt2) - 1\}$. Line 10-15 handles the case when the two TDTs contain an identical condition or a pair of conditions in the two TDTs are negation of each other. The intersection of two condition rules is obtained by treating each condition row as a cub set. For example, the intersection of row {C1, Y, X} and {C1, X, N} is a new row {C1, Y, N}.

In the worst case, the running time of this algorithm is $O(C_1C_2(R_1 + R_2) + A_1 + A_2)$, where C, R, and A stands for the number of conditions, the number of rules, and the number of action sets respectively.

3.2.3 Merging a TDT and an Action Set

A TDT and an action set can usually be merged unless the action set modifies a variable that is used to compute a condition in the TDT. There are two cases: (I) when the action set appears before the TDT, the two can always be merged; (II) when the action set appears after the TDT, merging is valid only if there is no def-use path which starts from within one action set ends in one condition in the TDT. A def-use path is a sequence of operations where each operation defines a data item used in the computation of the next one.

After merging a TDT with its following or preceding action set, the condition stub and the condition entries in the resulting TDT remains the same. The action set needs to be inserted appropriately in each column of the resulting table. We give one example in below.

Example 3.2. We assume a_3 follows TDT1 in an action set of concurrency type serial. We can then merge a_3 and TDT1 to produce a new table TDT2 with the same functionality and timing semantics.

TDT1:

	C	Y	N
a_1	1	0	
a_2	0	1	

a_3;

$\Longrightarrow$ TDT2:

	C	Y	N
a_1	1	0	
a_2	0	1	
a_3	1	1	

To preserve the behavior, we specify a concurrency type of serial between (a_1, a_3) and (a_2, a_3) in the action stub of TDT2. The serialization order follows the order in which these actions appear in the action stub. □

4 TDT-Based Presynthesis Techniques

Presynthesis works on behavioral descriptions. It removes control redundancies using assertions and extracts information useful in obtaining optimal synthesis result. The process of removing control redundancies and producing optimized behavioral descriptions is also referred to as presynthesis optimization.

Presynthesis optimization is implemented by first translating the input HDL description into the TDT representation, then performing TDT optimization techniques, and finally translating the optimized TDT model back into the HDL description. The TDT optimization techniques are based on the TDT transformations introduced in the previous section. Three optimization techniques are carried out: (I) *column reduction*, (II) *row reduction*, and (III) *action sharing*.

TDT transformations can also be used to extract additional information that is useful in synthesis to obtain optimal final results. As an example, we will show how m.e. operator pairs can be identified in TDT models and how this information can be used to obtain optimal scheduling solutions.

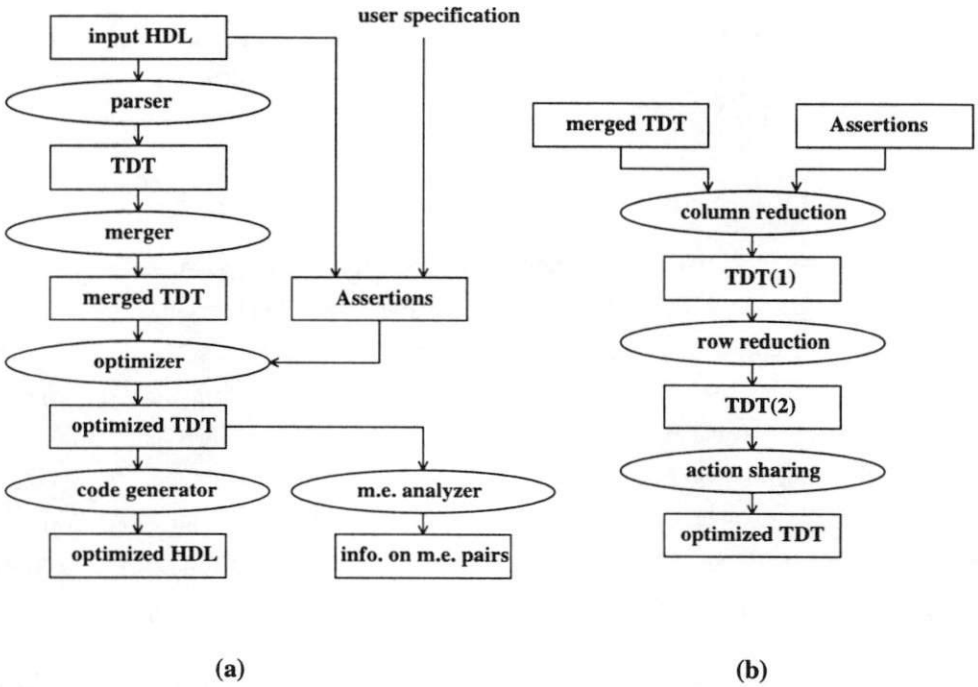


Figure 2: Flow diagram for presynthesis: (a) the whole picture, (b) details of the optimizer.

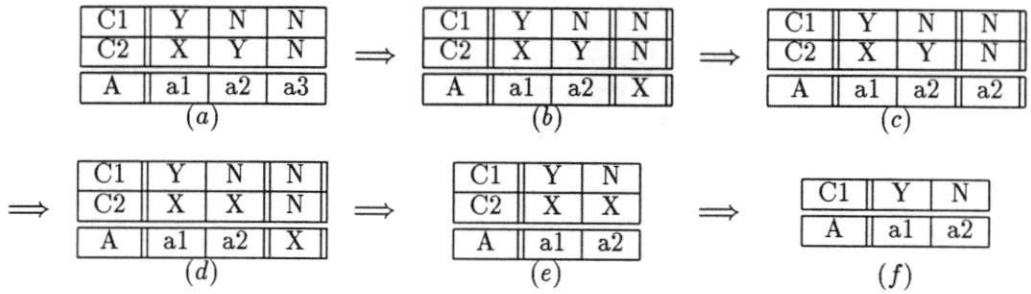
Figure 2 illustrates the presynthesis process. Note that the m.e. analysis is performed on the optimized TDT representation. The TDT merging transformations presented in the previous section are used in the merger. In this section, we focus on the details of the presynthesis optimizer and the m.e. analyzer. We introduce column reduction, row reduction, action sharing, and m.e. analysis

respectively in each subsection. Both the optimizer and the m.e. analyzer are based on the basic TDT transformations presented in the previous section.

4.1 Column Reduction

By reducing the number of rows and the number of columns, the TDT transformations presented in the previous section can be used for control oriented optimization of system behavior. For example, consider the sequence of transformations performed on a TDT shown in below.

Example 4.1. Given assertion: $\overline{C}1 \wedge \overline{C}2 = 0$, consider the following transformations on TDT(a).



Column 3 in TDT (a) is identified as a Don't Care column and marked explicitly in (b). Since action entry in the Don't Care column can assume any value, we change action entry in TDT (b) to 'a2' to produce TDT (c). Because column 2 and 3 in TDT (c) differ only in one condition entry, they are merged to produce TDT (d). TDT (e) is obtained by dropping the Don't Care column in TDT (d). Finally, since row 2 in TDT (e) contains only Don't Care values, we can drop this row and obtain TDT (f). □

As shown above, a sequence of behavior-preserving transformations can be applied to reduce the number of columns in a TDT. This process is very similar to that of two-level logic optimization. Equation 8 in Section 2 shows that a TDT can be expressed in a sum-of-product (SOP) form, where each term in the sum consists of several products of condition literals or negative condition literals. Recall that assertions are Boolean functions over conditions that are always true. Then minimizing the number of columns in a TDT is equivalent to minimizing the number of product terms (or the cardinality) of the SOP form of the TDT.

> **The Column Reduction Problem:** Given a TDT in its algebraic form as in Equation 8, and assertions specified in the form $\vec{A}(C_1, C_2, \cdots, C_n) = 1$, minimize the cardinality of Equation 8.

This problem can be solved using logic synthesis techniques. We can use a two-level logic minimizer such as ESPRESSO to minimize the number of columns in a TDT. This is accomplished by mapping as follows:

condition literal	$\longrightarrow$	input literal
action enabling vector	$\longrightarrow$	output literals
assertion	$\longrightarrow$	Don't Care

Since assertion are specified in logic equations, minterms that satisfy the assertion equations need to be generated to form the PLA format as input for ESPRESSO. We give one example in below to show how the representation is changed and how ESPRESSO is used to reduce the number of columns in a TDT.

Example 4.2. Consider TDT (a) with assertion: $\overline{C}1 \wedge \overline{C}2 = 0$.

C1	Y	N	N
C2	X	Y	N
a1	1	0	0
a2	0	1	0
a3	0	0	1

(a)

$\Longrightarrow$

```
.i 2
.ilb C1 C2
.o 3
.olb a1 a2 a3
.p 3
1- 100
01 010
00 001
.e
```

(b)

$\Longrightarrow$

```
.i 2
.ilb C1 C2
.o 3
.olb a1 a2 a3
.p 3
1- 100
01 010
00 ---
.e
```

(c)

$\Longrightarrow$

```
.i 2
.ilb C1 C2
.o 3
.olb a1 a2 a3
.p 2
0- 010
1- 100
.e
```

(d)

$\Longrightarrow$

C1	Y	N
C2	X	X
a1	1	0
a2	0	1
a3	0	0

(e)

$\Longrightarrow$

C1	Y	N
a1	1	0
a2	0	1

(f)

Column 3 in TDT (a) violates the specified assertion and therefore can be identified as a Don't Care column. Since a Don't Care column will never be selected for execution, we can change each action entry in TDT (a) into a Don't Care '-' to form TDT (b). We do a representation transformation to change TDT(b) into PLA form in (c). ESPRESSO works on the PLA form in (c) and produces an optimized PLA representation in (d). We transform the PLA form in (d) back into TDT (e). Finally, we perform row reduction to remove the condition row with all Don't Cares and the action row with all '0's in (e) to produce TDT (f). □

4.2 Row Reduction

Row reduction reduces the number of condition literals, or condition rows, in a TDT. The number of rows can be reduced by performing two of the behavior preserving TDT transformations: (I) row elimination, which eliminates condition rows with all Don't Cares, or (II) row encoding, which finds a form of encoding of condition rows to reduce the number of conditions. Row reduction is typically carried out after column reduction for two reasons. First, additional condition-entry Don't Cares are produced after column reduction is carried out. Second, row encoding has to be performed after column reduction since the result of row encoding depends on the number of columns and values of condition entries which are changed after column reduction.

4.3 Action Sharing

Action sharing refers to the identification of duplicate actions. Action sharing is performed in two major steps: (I) searching for identical action sets, (II) merging corresponding action rows when it's valid. Merging is valid only if the concurrency types among action sets and specified serialization relations can be preserved. For example, suppose we have two identical action sets, one produces data for action set as_1, one for as_2, then both as_1 and as_2 need to depend on the shared action set in order for this sharing to be valid.

The optimization process of action sharing is outlined in Algorithm 4.1. Before merging identical action sets in Step 3, we check to see if the merging violates any originally specified concurrency relations or serializations in Step 2. Step 3 also modifies the serialization relations so that each shared copy will basically inherit all the data dependencies of the original action sets.

Algorithm 4.1 Sharing Identical Action Sets

---

shareActionSets(tdt)

1. Search for identical action sets in the action sub of *tdt*;
2. Decide if it is valid to merge these action sets into a shared copy by checking into the concurrency types among action sets and serialization relations;
3. If Step 2 returns invalid go to Step 4, otherwise merge the corresponding action rows and modify the serializations when necessary;
4. Repeat Step 1-3 until no more action sets can be shared.

Notice that many details have been left out so we can focus on giving a big picture. The search process can be implemented as a repeat loop which will be executed $O(A^2)$ times, where A is the number of action sets. In the worst case, all serialization relations and concurrency types will be checked, which costs $O(A^2)$ in running time. Each merging takes $O(R)$ in time, where R is the number of rules in *tdt*. In the worst case, merging takes $O(AR)$ when all action sets are identical. However, this is unlikely to happen. In conclusion, the worst case running time of this algorithm is $O(A^2 + AR)$, while the typical running time should be $O(A^2 + R)$.

Example 4.3. Consider the description fragment shown in (a). It is first translated into a TDT model in

(b). Following Algorithm 4.1, we search for identical action sets in TDT (b) and merge them to form TDT (c). Then, we generate the optimized code in (d) from TDT (c).

```
if(sync_mode) {
   if(msgwait(ichannel))
     xdata = receive(c); /*1*/
   else
     another_action_set;
}
else
   xdata = receive(c); /*3*/
```

(a)

sync_mode	Y	Y	N
msgwait	Y	N	X
xdata = receive(c) /*1*/	1	0	0
another_action_set	0	1	0
xdata = receive(c) /*3*/	0	0	1

(b)

sync_mode	Y	Y	N
msgwait	Y	N	X
xdata = receive(c)	1	0	1
another_action_set	0	1	0

(c)

```
if(!sync_mode | msgwait(c))
   xdata = receive(c);
else
   another_action_set;
```

(d)

Here we have merged the two copies of the "`receive(c)`" operation. □

4.4 Identifying Mutually Exclusive Operations for Behavioral Synthesis

Information on mutually exclusive (m.e.) operator pairs can be used to improve solution in the *scheduling* and *binding* phase of high-level synthesis. Scheduling determines the start time of each operation while binding maps operations to hardware components. Binding and scheduling are interrelated problems. Decisions made in binding will affect the result of scheduling and vice versa. The quality of binding and scheduling can be determined by the resource usage and the total delay. The two goals of reducing total delay and reducing resource usage are often conflicting. Total delay can be be reduced by maximizing operations in each control step. This however often increases the number of required resources. On the other hand, resource sharing often results in additional serialization and hence a longer delay. However, operations can share resource without increasing the total delay only if they are "mutually exclusive".

Mutually exclusive operations in a behavioral description are operations that will never be executed in the same control step in any execution of the system behavior. In addition, as shown in one example in this paper, the execution of an operation may imply that the execution of another operation is a behavioral *Don't Care* [20]. These implications provide a rich source of mutually exclusive operations that can be exploited to improve the quality of high-level synthesis. Accordingly, we also consider operations as mutually exclusive if they never need to be executed together.

4.4.1 Classification of m.e. Operator Pairs in Behavioral Models

In a non-pipelined execution, two operations with a data dependency can not be scheduled in the same control step, and therefore are not mutually exclusive. Two operations with no data

dependency are mutually exclusive if they belong to mutually exclusive control paths such as conditional branches, or if the result of one operation is a Don't Care when the other operation is executed. According to the way how mutually exclusive (m.e.) operations are identified, we can divide them into three categories,

(i) *structural*,

(ii) *behavioral*, and

(iii) *data-flow*.

A pair of operations is considered as a structural m.e. pair if the two operations can be identified entirely based on the language structures in the input HDL description. A behavioral m.e. pair refers to two operations conditionally enabled under mutually exclusive conditions (that is, conditions that never evaluate to true simultaneously). A data-flow pair of m. e. operations refers to two operations that are never required to be executed in any execution of the system behavior based on the data values. Identification of data-flow m.e. pairs relies on the data-flow analysis and the knowledge of other m.e. pairs.

Example 4.4. Consider the following HDL description in HardwareC. It is modified from the example in [15].

```
process example(a, b, c, d, e, f, g, x, y, u, v)
in port a[8], b[8], c[8], d[8], e[8], f[8], g[8];
in port x, y;
out port u[8], v[8];
{
        static T1;
        static T2[8];
        static T3[8];
        static T4[8];
        static T5[8];

        T1 = ( a + b ) < c;              /* -- 1 -- */
        T2 = d + e;                      /* -- 2 -- */
        T3 = c + 1;                      /* -- 3 -- */

        if(y) {
            if(T1)
                u = T3 + d;              /* -- 4 -- */
            else if( !x )
                u = T2 + d;              /* -- 5 -- */
            if( !T1 && x )
                z = T2 + e;              /* -- 6 -- */
        }
        else {
            T4 = T3 + e;                 /* -- 7 -- */
            T5 = T4 + f;                 /* -- 8 -- */
             u = T5 + g;                 /* -- 9 -- */
        }

}
```

Operator pairs $\{+_4, +_5\}$, $\{+_4, +_7\}$, $\{+_4, +_8\}$, $\{+_4, +_9\}$, $\{+_5, +_7\}$, $\{+_5, +_8\}$, $\{+_5, +_9\}$, $\{+_6, +_7\}$, $\{+_6, +_8\}$, and $\{+_6, +_9\}$ are structural m.e. pairs. Operator pairs $\{+_4, +_6\}$ and $\{+_5, +_6\}$ are behavioral. Operator pairs $\{+_1, +_7\}$, $\{+_1, +_8\}$, $\{+_1, +_9\}$, $\{+_2, +_3\}$, $\{+_2, +_4\}$, $\{+_2, +_7\}$, $\{+_2, +_8\}$, and $\{+_2, +_9\}$ are data-flow m.e. pairs. □

4.4.2 TDT-Based Approach for Identifying M.E. Operator Pairs

Algorithm 4.2 Identify M.E. Operator Pairs Using TDT

meFind(inputHDL)
1. Convert *inputHDL* descriptions into connected leaf TDTs;
2. Merge the leaf TDTs;
3. Perform TDT-based optimization to produce *optimized_tdt*;
4. Mark any two operators in two different columns of a TDT as a m.e. pair;
5. Call $dataflow\_meFind(optimized\_tdt)$ to identify the remaining m.e. operator pairs.

Step 1-3 are performed for tasks other than identifying m.e. operator pairs. M.E. pairs identified in Step 4 are either structural m.e. pairs or behavioral pairs. In step 1, any one m.e. pair of operators will end up in one leaf TDT. It can be checked that merging transformations as described in Section 4 will keep two operators in a leaf TDT in separate columns of one TDT until they are marked as m.e. in Step 4. Merging and optimization process check condition dependencies. Operators enabled under mutually exclusive conditions will eventually be placed in different columns of a TDT. Therefore behavioral m.e. pairs will also be marked in Step 4. Data-flow m.e. pairs are identified in Step 5 with the help of a TDT-based def-use analysis. We leave the detailed discussion on data-flow pairs to the next sub-section.

Notice that Step 1-3 won't need to be performed once we have optimized TDT representation, which is also used in other presynthesis tasks. Step 4 takes $O(n^2)$ in time, where n is the total number of operators. Running time of Step 5 will be shown later with Algorithm 4.3.

4.4.3 Identifying Data-flow M.E. Operators in TDT Models

Data-flow m.e. pairs are identified with the help of def-use analysis. We perform a def-use analysis on the merged TDT representation. We give our definition of the use set of an operator in below.

Definition 4.1 *The* **use set** *of an operator o is the set of operators that consumes the data computed by o.*

Use sets of all operators in a behavioral description can be computed using standard data-flow techniques as discussed in [1]. We list the operator use sets of the description example in Table 2. An 'OUT' indicates that the result of the operator is written to an output port or sent to another process via a messaging channel.

Table 2: Use sets of operators in the example in Figure 1.

operator	operator use set	operator	operator use set
$+_1$	$\{ +_4, +_5, +_6 \}$	$+_6$	$\{ OUT \}$
$+_2$	$\{ +_5, +_6 \}$	$+_7$	$\{ +_8 \}$
$+_3$	$\{ +_4, +_7 \}$	$+_8$	$\{ +_9 \}$
$+_4$	$\{ OUT \}$	$+_9$	$\{ OUT \}$
$+_5$	$\{ OUT \}$		

Given the use sets of operators and information on whether or not some of the operator pairs are mutually exclusive, additional information on m.e. pairs can be obtained following Theorem 4.1 as shown in below. M.E. pairs thus detected are said to be data-flow m.e. pairs.

Theorem 4.1 *Given two operators o_1 and o_2 and their use sets $USE(o_1)$ and $USE(o_2)$,*

(a) o_1 and o_2 are mutually exclusive if $\forall \alpha \in USE(o_1), \forall \beta \in USE(o_2)$, α and β are mutually exclusive;

(b) o_1 and o_2 are mutually exclusive if $\forall \alpha \in USE(o_1)$, α and o_2 are mutually exclusive;

(c) o_1 and o_2 are not mutually exclusive if $\exists \alpha \in USE(o_1) \exists \beta \in USE(o_2)$ such that α and β are not mutually exclusive;

(d) o_1 and o_2 are not mutually exclusive if $\exists \alpha \in USE(o_1)$ such that α and o_2 are not mutually exclusive.

Proof: First we define *usage condition* operator o as the condition under which the result of o is ever used. We denote usage condition of o as $ucond(o)$. Then we observe that

(i) $ucond(o) = \bigcup_{\alpha \in USE(o)} ucond(\alpha)$

(ii) Two operators o' and o'' are mutually exclusive if and only if $ucond(o') \cap ucond(o'') = \phi$.

We start with proving (b). Suppose $\forall \alpha \in USE(o_1), \alpha$ and o_2 are mutually exclusive. Then we have $\forall \alpha \in USE(o_1), ucond(\alpha) \cap ucond(o_2) = \phi$. Then $\bigcup_{\alpha \in USE(o_1)} ucond(\alpha) \cap ucond(o_2) = \phi$. Since $ucond(o_1) = \bigcup_{\alpha \in USE(o_1)} ucond(\alpha)$, $ucond(o_1) \cap ucond(o_2) = \phi$, and therefore o_1 and o_2 are mutually exclusive.

We use (b) to prove (a). Suppose $\forall \alpha \in USE(o_1), \forall \beta \in USE(o_2)$, α and β are mutually exclusive. Pick arbitrarily $\alpha \in USE(o_1)$. Then α are o_2 are mutually exclusive according to (b). Since α is picked arbitrarily, o_1 and o_2 are now mutually exclusive.

It is straight forward to prove (c) and (d) by following basic definitions. □

After TDT merging, any pair of operators that appear in different columns of a TDT are determined as a m.e. pair. We can also determine that any pair of operators with a data-dependency between them is not a m.e. operator pair. With this information as a starting point, we can apply Theorem 4.1(b) recursively to determine all data-flow m.e. pairs. The order to apply Theorem 4.1 is presented in the Algorithm 4.3. Notice that we start with a TDT not only properly merged, but also optimized.

Algorithm 4.3 Algorithm to Identify Data-flow m.e. Operator Pairs

---

dataflow_meFind(optimized_tdt)

1. Create a def-use graph $G = \{V, E\}$ where
   $V = \{o | o \text{ is an operator}\} \cup \{OUT\}$,
   $E = \{ (o_1, o_2) \mid o_2 \in USE(o_1)\}$;
2. $Visited \leftarrow \{OUT\}$;
3. **foreach** edge $e = (o_1, o_2)$ **do**
4. $me(o_1, o_2) \leftarrow$ 'N';
5. **foreach** pair (o_1, o_2) **do**
6. **if** o_1 and o_2 are in different columns of a TDT **then**
7. $me(o_1, o_2) \leftarrow$ 'Y';
8. **repeat**
9. Pick $o \in (V - Visited)$ where all operator in $USE(o)$ have been visited;
10. **foreach** $\beta \in Visited$ **do**
11. Determine the value of $me(o, \beta)$ following Theorem 4.1(b);
12. $Visited \leftarrow Visited \cup \{o\}$;
13. **until** (all nodes in V have been visited).

The complexity of this algorithm is $O(n^3)$, where n is the number of operators. The creation of def-use graph takes $O(n^2)$. The first loop takes $O(E)$ where E is the number of edges in the def-use graph. The second loop takes $O(n^2)$. The repeat loop will be repeated n times. The first operation in this loop needs to be expanded before actual implementation, since we are showing only an outline. If we manage a list of unvisited nodes and for each un-visited node we also manage a list of use nodes, the total time spent on the first operation in n iterations will be $O(n^2)$. In each iteration of the repeat loop, the inner loop takes $O(n^2)$ since it takes $O(|USE(o)|)$ to check Theorem 4.1 (b).

In this section, we have looked into presynthesis techniques. We have followed Figure 2 to present the key algorithms used in the TDT-based presynthesis system, namely the algorithms for column reduction, row reduction, action sharing, and m.e. analyzer.

In the next section, we introduce PUMPKIN, our software implementation of the TDT-based presynthesis system which takes HardwareC descriptions as input and outputs optimized HardwareC code and information on m.e. operator pairs. Since the code generator as shown in Figure 2 is more implementation dependent, we also leave the discussion of the code generator to the next

section, where we will show our HardwareC code generation algorithm along with other details of PUMPKIN.

5 PUMPKIN Presynthesis System

The presynthesis system shown in Figure 2 has been implemented in about 20,000 lines of C++ code named PUMPKIN. The PUMPKIN presynthesizer features a UNIX shell-like command interface that is provided to allow interactive use by the system designer. The system designer reads in a given HardwareC description and applies External Don't Care (EDC) conditions as additional input for the optimizer. 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. The command interface to PUMPKIN is shown in Table 3. A typical command sequence in optimization include `readhc`, `mergedt`, `edc_input`, `opdt`, and `dt2hc`.

Table 3: Commands in PUMPKIN.

`readhc`	Read in a HardwareC description, and convert it into connected leaf TDTs
`mergedt`	Merge leaf TDTs obtained from `readhc`
`edc_input`	Invoke the `edc` sub system to specify external Don't Cares.
`pedc`	Print all the currently specified external Don't Cares.
`reset_edc`	Reset the list of current Don't Care conditions to NULL.
`opdt`	Perform various optimization techniques as described in Section 3.
`dt2hc`	Generate HardwareC code from (optimized) TDT model.
`pme`	Detect and print m.e. operator pairs in the (optimized) TDT.
`preloaddt`	Load TDT directly.
`pdt`	Print TDT models.

The structure of PUMPKIN follows the flow diagram in Figure 2. We have presented basic algorithms for the merger in Section 3. We have given details on the optimizer and m.e. analyzer in Section 4. In the rest of this section, we show how HardwareC code is generated in PUMPKIN and how PUMPKIN handles input HDL code with exceptions.

5.1 Generating HDL Descriptions from TDT

To use the existing tools, we translate TDT models back into HDL behavioral descriptions. Algorithm 5.1 shows how PUMPKIN generates HardwareC code from optimized TDT models. Note that when one action set is selected for execution in all paths in a sub-tdt, only one copy of the code corresponding to this action set is generated. However, if control jump structures are not allowed in a HDL, it is not always possible to rewrite a description without duplicating identical

code segments. In cases when several but not all columns shared an action set in a TDT, often we have to duplicate the shared action sets during code generation.

Algorithm 5.1 Generating HardwareC Code from TDTs Actions in Limited-entry Form

---

gencodeFromTDT(tdt)
T1. **if** (there is an action row with all '1's) **then**
T2. Split *tdt* into tdt_1, $actionSet_m$, and tdt_2;
T3. Call $gencodeFromTDT(tdt_1)$;
T4. Call $gencodeFromActionSet(actionSet_m)$;
T5. Call gencodeFromTDT(tdt_2);
T6. **elseif** (there is still a shared action row) **then**
T7. Separate the shared part from tdt;
T8. Call *gencodeFromActionSet*() on the separated action sets;
T9. Call *gencodeFromTDT*() on the reset of *tdt*;
T10. Put two pieces of HardwareC code generated above according to the way the action set is separated from tdt;
T11. **elseif** (*tdt* is a unit TDT with one condition) **then**
T12. Emit "if (condition of *tdt*) ";
T13. Call *gencodeFromActionSet*(yes-action-set of *tdt*);
T14. Emit "else";
T15. Call *gencodeFromActionSet*(no-action-set of *tdt*);
T16. **else**
T17. Pick in the condition entries a row with no Don't Cares;
T18. `/* Create two tables` tdt_A `and` tdt_B `as follows */`
T19. Copy all columns in *tdt* with 'Y' in row i to tdt_A;
T20. Copy all columns in *tdt* with 'N' in row i to tdt_B;
T21. Delete row i from tdt_A and tdt_B;
T22. `/* Generate HardwareC code as follows */`
T23. Emit "if (C_i)";
T24. Call $gencodeFromTDT(tdt_A)$;
T25. Emit "else";
T26. Call $gencodeFromTDT(tdt_B)$;
T27. **endif**.

gencodeFromActionSet(actionSet)
S1. **foreach** action **in** actionSet **do**
S2. Emit proper delimiter according to the concurrency type;
S3. Call gencodeFromAction(action);
S4. Emit proper delimiter according to the concurrency type;
S5. **end foreach**.

gencodeFromAction(action)
A1. **switch** (*action* →type)
A2. **case** TDT: call $gencodeFromTDT(action \rightarrow tdt)$;
A3. **case** ActionSet: call $gencodeFromActionSet(action \rightarrow subActionSet)$;
A4. **case** ALU: ...
A5. **case** IO: ...
A6. **case** MessagePassing: ...
A7. **endswitch**.

One approach to avoid duplicating shared action sets, as shown in the second branch in *gencodeFromTDT*, is to separate the shared action sets from the TDT while generating HardwareC code from TDT. This separation is not always possible. It is only valid in the following cases:

(a) A concurrency type of data-parallel is specified on the action stub and the shared action appears as the first action set.

(b) A concurrency type of data-parallel is specified on the action stub and the shared action appears as the last action set.

(c) Cases that can be transformed into the above cases via behavior preserving transformations. For example, the order of two action set can be swapped in a TDT if a concurrency type of data-parallel is specified and there is no data-dependency specified between the two action sets.

Though this approach has avoided having multiple copies of identical action sets, it introduces additional control circuits.

5.2 Presynthesis Optimization on Behavioral Descriptions with Exceptions

Translating HDL descriptions with `disable` statements involving modifying both the parser and merger. Algorithm 5.2 shows the major changes. Since each column in a TDT represents a control path and all action sets are presented, modeling a control jump in TDTs simply requires deletion of action sets between the control jump and the jump target point in each path that contains the control jump.

Algorithm 5.2 Translate HDL Description with Disable into TDT and Perform Merging Operation

---

1. `/* consider step 2-4 as a modified parser */`
2. Process **disable** in the same way as all other ALU statements;
3. Add a special *endblock* statement for each named block
4. Call the original parser;
5. `/* consider step 6-9 as the modified merger */`
6. Call other merging algorithms;
7. **foreach** column with a pair of `disable` and `endblock` statements **do**
8. Mark all action sets in between as '0';
9. Remove all **disable** statements;
10. Remove all **endblock** statements.

We have shown in this section some details on PUMPKIN, a software tool that carries out presynthesis on HardwareC descriptions. The tool is available at

`http://www.ics.uci.edu/~jian/software-distribution.`

We are under the process of adding a VHDL front end so that PUMPKIN will soon be able to perform the same set of presynthesis tasks on VHDL descriptions.

6 Experimental Results

We have run PUMPKIN on named benchmarks to test our presynthesis algorithms. We first present results on source-level optimizations, followed by results related to the extraction of m.e. in-

formation on operations.

6.1 Results on Presynthesis Optimizations

PUMPKIN performs optimization in the order of column reduction, row reduction, and action sharing. To evaluate the effect of each optimization scheme, we turn off other optimizations in the experiments. Since row reduction is typically made possible after column reduction is performed, we evaluate the combined effect of column and row reduction with one group of experiments.

For each group of experiments, our experimental methodology is as follows. The HDL description is compiled into TDT models, run through certain optimizations, and finally output as a HardwareC description. This output is provided to the Olympus High-Level Synthesis System [7] 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.

6.1.1 Results on Column/Row Reduction

Results on column/row reduction are presented in Table 4. 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 'motorcntrl' refers to a trolley controller that is used to transport assembly components on a shop-floor. Description 'ecc' decodes parity encoded data transmitted through a serial line and corrects transmission errors. Description 'parker86' is taken from [21]. Finally, 'traffic' refers to the HDL model of a traffic light controller.

Table 4: Synthesis results: cell counts before and after column/row reduction.

design	circuit size (in number of cells)		Don't Care condition		Δ%
	before optimization	after optimization	condition	user input to PUMPKIN	
gcd	272	230	positive inputs	`xi > 0 && yi >0;`	15
motorcntrl	2952	366	no position variation	`iveer = 0;`	88
motorcntrl	2952	2774	there is always variation	`iveer <>0;`	7
ecc	141	119	correct only single error	`rp[0]+rp[1]+rp[2]<=1;` `cp[0]+cp[1]+cp[2]<=1;`	16
parker86	384	270	in2 + in3 = 0	`in2 + in3 = 0;`	29
traffic	35	34	no external dc specified		3

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. The second one, which is 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' is considered for use in which only single errors are corrected.

As shown in Table 4, use of external conditions can lead to reductions in the size of the hardware circuits. Thus 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 depends strongly upon the external Don't Care information used in presynthesis. This chief function of column/row reduction of the optimizer is to to enable the system designers to specify additional information about system environment and use it for system optimization. As a consequence, it makes the most general purpose modules easily reusable in a similar but different environment.

6.1.2 Results on Action Sharing

Results on action sharing are presented in Table 5. Description 'comm/exec_unit' refers to the execution unit in a ethernet controller. Description 'cruiser/State' models a hardware module for speed regulation in a cruiser. Description 'i8251/xmit' is the transmit process in a HardwareC version of the 'i8251' design. Description 'daio_receiver' is the receiver part of the Digital Audio Output (DAIO) chip. Description 'frisc' refers to a simplified RISC processor. All the designs are from the high-level synthesis benchmark suite [7].

Table 5: Synthesis results: cell counts before and after action sharing.

design	circuit size (cells)		Δ%
	before	after	
comm/exec_unit	864	587	32
cruiser/State	356	270	24
i8251/xmit	971	921	5
daio_receiver	440	388	12
frisc	4353	3940	9

In Table 5, we compare the synthesized circuit sizes of each benchmark description before and after action sharing is performed. The percentage of circuit size reduction is computed for each description and listed in the last column of Table 5. Note that this improvement depends on the

amount of sharable code segments in the input behavioral descriptions.

The overall effect of presynthesis optimizations is to rewrite the description and remove redundancies in the input description either as a part of the original specification or as a result of assertions. This task is often done by the system designer in an attempt to arrive at an efficient implementation. Performing this task by an automated tool such as PUMPKIN makes the results of the synthesis process relatively insensitive to HDL coding-styles, and hence reduces the time designers spend in behavioral specification.

6.2 Results on Identifying m.e. Operators

Our approach for identifying m.e. operator pairs has been implemented as a part of the PUMPKIN presynthesis system. As shown in Figure 2, this part for extracting m.e. information, also referred to as the m.e. analyzer, starts with the optimized TDT representation.

We first run PUMPKIN on the sample description in Example 4.4. For comparison, we also manually run other approaches on the same example. In Table 6, we list all the m.e. operator pairs and mark those identified by each approach in a corresponding column. In the table, Kim's approach refers to Kim and Liu's approach [16]. Approach 'SB' stands for the status bit approach [22]. Approach 'CV' refers to condition vector approach [17]. The approach 'path-based' refers to an approach based on path analysis [23]. Approach 'CG' stands for the usage condition approach using condition graphs [15]. Finally, approach 'TDT' refers to our approach based on TDT modeling and def-use analysis.

Note in particular that the result in the 'CG' column has been obtained by following the algorithms presented in [15]. Though according to the authors of [15] the 'CG' approach can be modified to identify all m.e. operator pairs [24], our chief contribution on m.e. detection lies in that the TDT-based approach provided a combined framework for both presynthesis optimizations and extracting m.e. information on the optimized behavior representations in addition to a clear classification based on the source of each m.e. operator pair.

We have run the comparison experiment on more sample behavior descriptions and presented the results in Table 7. The behavioral descriptions in Table 7 are either picked from previous publications or from the high-level synthesis benchmark suite. Description 'kim' refers to the example used in [16]. Description 'jian' is the example presented in Section 4. Description 'juan' refers to the example used in [15]. Description 'parker' is a HardwareC example from the high-level synthesis benchmark suite [7].

We discuss mutual exclusiveness in the context where operations can share resource in a certain implementation. For example, it won't be useful to consider the the mutual exclusiveness of an integer subtraction and a floating point subtraction. For this reason, we only consider certain types of operators that can be implemented on the same type of function units when we count the

Table 6: A comparison of m.e. operator pairs identified by different approaches.

mutually exclusive operators	approaches					
	Kim's	SB	CV	path-based	CG	TDT
$\{+_1, +_7\}$			√			√
$\{+_1, +_8\}$			√			√
$\{+_1, +_9\}$			√			√
$\{+_2, +_3\}$					√	√
$\{+_2, +_4\}$					√	√
$\{+_2, +_7\}$			√		√	√
$\{+_2, +_8\}$			√		√	√
$\{+_2, +_9\}$			√		√	√
$\{+_3, +_5\}$			√		√	√
$\{+_3, +_6\}$			√		√	√
$\{+_4, +_5\}$	√	√	√	√	√	√
$\{+_4, +_6\}$				√	√	√
$\{+_4, +_7\}$	√	√	√	√	√	√
$\{+_4, +_8\}$	√	√	√	√	√	√
$\{+_4, +_9\}$	√	√	√	√	√	√
$\{+_5, +_6\}$				√	√	√
$\{+_5, +_7\}$	√	√	√	√	√	√
$\{+_5, +_8\}$	√	√	√	√	√	√
$\{+_5, +_9\}$	√	√	√	√	√	√
$\{+_6, +_7\}$	√	√	√	√	√	√
$\{+_6, +_8\}$	√	√	√	√	√	√
$\{+_6, +_9\}$	√	√	√	√	√	√

number of operators and compute the number of mutually exclusive operator pairs. The line 'waka 1' lists the experimental result assuming all addition and subtraction are implemented on one type of adders. The line 'waka 2' shows the result assuming all operations are implemented on ALUs. The line 'waka 3' considers only addition and adders.

Information on m.e. operator pairs can be used in synthesis to obtain optimal scheduling. Consider the behavior description in Example 4.3. Assume that only one adder is used. We use a modified list scheduler which utilizes information on m.e. pairs. When provided with different sets of m.e. operator pairs, different scheduling results are obtained as shown in Table 8. Column 'no'

Table 7: The result for m.e. operator pair identification.

behavioral description	# of operators	total # of m.e. pairs	% of m.e. pairs identified					
			Kim's	SB	CV	path-based	CG	TDT
kim	24	120	100	100	100	100	100	100
jian	9	22	45	45	55	64	86	100
juan	6	7	14	14	43	43	100	100
parker	16	55	78	78	96	78	78	100
waka 1	14	21	76	76	100	76	100	100
waka 2	16	22	73	73	100	73	95	100
waka 3	8	12	83	83	100	83	100	100

indicates that no m.e. information has been incorporated in scheduling. Column 'CG' indicates that the set of m.e. operator paris identified by the 'CG' method has been incorporated in scheduling. Column 'TDT' indicates that information on the full set of m.e. operator pairs, as detected by our TDT-based approach, has been incorporated.

Table 8: Scheduling results when informed of different sets of m.e. pairs.

description	# of control steps		
	no	CG	TDT
jian	9	4	3

7 Conclusion and Future Work

In this paper, we have presented presynthesis on HDL descriptions which consists of two major tasks: (I) optimize input behavioral HDL descriptions before any synthesis task is carried out, and (II) extract information that can be used to improve the results of synthesis optimizations. To implement the presynthesis tasks, we have come up with a novel tabular model called TDT. We have presented a set of behavior-preserving transformations on TDTs that are used in the three presynthesis optimization schemes: (I) column reduction, (II) row reduction, and (III) action sharing. We have also presented the TDT-based def-use analysis which is used to extract information on m.e. operator pairs. The tabular model allows us to specify and use Don't Care information in presynthesis optimizations.

We have presented a software implementation of our presynthesis system code named PUMPKIN. We have run PUMPKIN on named benchmarks. Our experiments show improved synthesis results after presynthesis optimizations are carried out on the input HDL descriptions or the information extracted from the input source has been incorporated in synthesis.

Despite the preliminary results, there are several limitations in the algorithms presented here. First, our merging algorithms handle only one condition loop at a time. When there are nested condition loops, they are represented as hierarchically connected process tables. Hence our scheme fails to remove the control-flow redundancy when there is an assertion concerning more than one loop conditions. Second, a few of our basic TDT transformations takes $O(RN)$ time, when R is the number of rules in a TDT, and N is either the number of action sets or the number of conditions in a TDT. This will be inefficient when the number of rules in a TDT grows exponentially with the number of operations and conditions in the input HDL descriptions. Therefore, as the first step of our future work, we plan to work on algorithms to increase the scope of the presynthesis optimizations, and to improve the efficiency of some of the algorithms.

During the course of this work, we find as by-products a few interesting facts about TDT models. First, the action sharing scheme can be used for code compaction in software generation. Second, the TDT offers a natural starting point for Hardware/Software partition, which is to implement condition part in hardware and actions in software. The ability of indirection [25] to shift the border between conditions and actions makes this model more appealing as an intermediate model for Hardware/Software partitioning. Also, it appears a convenient vehicle to derive timing information and interface driver routines from the TDT models. As the next step of our future work, we plan to investigate the feasibility, the advantages, and the disadvantage of applying TDT to other system design tasks such as scheduling, Hardware/Software partitioning, and interface resolution.

8 Acknowledgment

This research is supported in part by NSF CAREER Award MIP 95-01615. The first author also acknowledges the support from FMC fellowship provided by the FMC foundation and the college of engineering at the University of Illinois.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] R. K. Brayton, R. Camposano, G. D. Micheli, R. Otten, and J. van Eijndhoven, "The Yorktown Silicon Compiler System," in *Silicon Compilation* (D. Gajski, ed.), pp. 204–310, Addison-Wesley, 1988.

[3] A. L. Davis and R. M. Keller, "Data flow program graphs," *IEEE Computer*, vol. 15, no. 2, February 1982.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," *ACM SIGPLAN Notices*, 1989.

[5] M. C. McFarland, "The Value Trace: A data base for automated digital design," Tech. Rep. DRC-01-4-80, Design Research Center, Carnegie-Mellon University, 1978.

[6] A. Parker, J. Pizarro, and M. Milnar, "A program for data path synthesis," in *Proceedings of the 23rd Design Automation Coference*, pp. 461–466, June 1986.

[7] 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.

[8] C. Coelho and G. D. Micheli, "Dynamic scheduling and synchronization synthesis of concurrent digital systems under system-level constraints," *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 175–181, November 1994.

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

[10] 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.

[11] J. Li and R. K. Gupta, "HDL optimization using timed decision tables," in *Proceedings of the Design Automation Conference*, pp. 51–54, June 1996.

[12] K. Rath, V. Choppella, and S. D. Johnson, "Decomposition of sequential behavior using interface specification and complementation," *VLSI Design Journal*, vol. 3, no. 3-4, pp. 347–358, 1995.

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

[14] A. Lew, "On the emulation of flowcharts by decision tables," *Communications of the ACM*, vol. 25, no. 12, pp. 895–905, 1982.

[15] H.-p. Juan, V. Chaiyakul, and D. D. Gajski, "Condition graphs for high-quality behavioral synthesis.," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 170–174, 1994.

[16] T. Kim, J. W. Liu, and C. L. Liu, "A scheduling algorithm for conditional resource sharing," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 84–87, 1991.

[17] K. Wakabayashi and T. Yoshimura, "A resource sharing and control synthesis method for conditional branches," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 62–65, 1989.

[18] 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.

[19] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Languge*. Kluwer Academic Publishers, 1995.

[20] 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.

[21] A. Parker, J. Pizarro, and M. Mlinar, "A Program for Data Path Synthesis," in *Proceedings of the* 23^{rd} *Design Automation Conference*, pp. 461–466, June 1986.

[22] C.-J. Tseng, R.-S. Wei, S. G. Tothweiler, M. M. Tong, and A. K. Bose, "Bridge: A versatile behavioral synthesis system," in *Proceedings of the* 25^{th} *Design Automation Conference*, pp. 84–87, 1988.

[23] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. CAD*, vol. 10, no. 1, pp. 85–93, 1991.

[24] private communication.

[25] K. Rath, M. E. Tuna, and S. D. Johnson, "Specification and synthesis of bounded indirection," Tech. Rep. 398, Indiana University Computer Science Department, February 1994.

List of Figures

List of Tables