Decomposition of Timed Decision Tables and its Use in Presynthesis Optimizations

Jian Li
Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, Illinois 61801

Rajesh K. Gupta
Information & Computer Science
University of California, Irvine
Irvine, California 92697

Abstract

In this paper we introduce the decomposition of Timed Decision Tables (TDT), a tabular model of system behavior. The decomposition can be used in system partitioning or in HDL code restructuring to improve synthesis results. The TDT decomposition is based on the kernel extraction algorithm. By experimenting using named benchmarks, we demonstrate how TDT decomposition can be used in *presynthesis optimizations*. Presynthesis optimizations transform a behavioral HDL description into optimized HDL description that results in improved synthesized circuits.

1 Introduction

Presynthesis optimizations have been introduced in [1] as source-level transformations that produce "better" HDL descriptions. For instance, these transformations are used to reduce control-flow redundancies and make synthesis result relatively insensitive to the HDL coding-style. They are also used to reduce resource requirements in the synthesized circuits by increasing component sharing at the behavior-level [2].

The TDT representation consists of a main table holding a set of rules, which is similar to the specification in a FSMD [3], an auxiliary table which specifies concurrencies, data dependencies, and serialization relations among data-path computations, or *actions*, and a delay table which specifies the execution delay of each action.

The rule section of the 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. We associate an execution delay with each action. Actions are grouped into

action sets, or compound actions. With each action set, we associate a concurrency type of *serial*, *parallel*, or *data-parallel* [4].

Condition Stub	Condition Entries
Action Stub	Action Entries

Figure 1: Basic structure of TDTs.

The structure of the rule section 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.

In additional to the set of rules specified in a main table (the rule section), the TDT representation includes two auxiliary tables to hold additional information. Information specified in the auxiliary tables include the execution delay of each action, serialization, data dependency, and concurrency type between each pair of actions.

Example 1.1. Consider the following TDT:

	$a_{1,1}$	$a_{1,2}$	$a_{2,1}$	$a_{2,2}$	$a_{3,1}$	$a_{3,2}$
$a_{1,1}$		s ↗				
$a_{1,2}$						
$a_{2,1}$				d ↗		
$a_{2,2}$						
$a_{3,1}$						p
$a_{3,2}$						

c_1	Y	Y	N	
c_2	Y	N	X	**delay**
$a_{1,1}$	1	0	0	1
$a_{1,2}$	1	0	0	3
$a_{2,1}$	0	1	0	4
$a_{2,2}$	0	1	0	2
$a_{3,1}$	0	0	1	6
$a_{3,2}$	0	0	1	1

When c_1 = 'Y' and c_2 = 'Y', actions $a_{1,1}$ and $a_{1,2}$ are selected for execution. Since action $a_{1,2}$ is specified as a successor of $a_{1,1}$, action $a_{1,1}$ is executed with a one cycle delay followed by the execution of $a_{1,2}$. Symbols 'd' and 'p' indicate actions that are data-parallel (i.e. parallel modulo data dependencies) and parallel actions respectively. An arrow '↗' at row $a_{1,1}$ and column $a_{1,2}$ indicates that $a_{1,1}$ appears before $a_{1,2}$. In contrast, an arrow '↙' at row $a_{1,1}$ and column $a_{1,2}$ indicates that $a_{1,1}$ appears after $a_{1,2}$. □

The execution of a TDT consists of two steps: (1) select a rule to apply, (2) execute the action sets that the selected rule maps to. More than two action sets may be selected for execution. The order in which to execute those action sets are determined by the concurrency types, serialization relations, and data dependencies specified among those action sets [4], indicated by 's', 'd', and 'p' in the table above.

An action in a TDT may be another TDT. This is referred to as a *call* to the TDT contained as an action in the other TDT, which corresponds to the hierarchy specified in HDL descriptions. Consider the following example.

Example 1.2. Consider the following calling hierarchy:

TDT_1

c_1	Y	N
a_1	1	0
TDT_2	0	1

TDT_2

c_2	Y	N
a_2	1	0
a_3	0	1

Here when c_1 = 'N' the action needs to be invoked is the call to TDT_2, forces evaluation of condition c_2 resulting in actions a_2 or a_3 being executed. No additional information such as concurrency types needs to be specified between action a_1 and TDT_2 since they lie on different control paths. For the same reason, we omit the auxiliary table for TDT_2. □

Procedure/function calling hierarchy in input HDL descriptions results in a corresponding TDT hierarchy. TDTs in a calling hierarchy are typically merged to increase the scope of presynthesis optimizations. In the process of presynthesis optimizations, merging flattens the calling hierarchy specified in original HDL descriptions. In this paper we present TDT decomposition which is the reverse of the merging process. By first flattening the calling hierarchy and then extracting the commonalities, we may find a more efficient behavior representation which leads to improved synthesis results. This allows us to restructure HDL code. This code structuring is similar to the heuristic optimizations in multilevel logic synthesis. In this paper, we introduce code-restructuring in addition to other presynthesis optimization techniques such as column/row reduction and action sharing that have been presented earlier [1, 4, 2].

The rest of this paper is organized as follows. In the next section, we introduce the notion of TDT decomposition and relate it to the problem of kernel extraction in an algebraic form of TDT. Section 3 presents an algorithm for TDT decomposition based on kernel extraction. Section 4 shows the implementation details of the algorithm and presents the experimental results. Finally, we conclude in Section 5 and presents our future plan.

2 TDT Decomposition

TDT decomposition is the process of replacing a flattened TDT with a hierarchical TDT that represents an equivalent behavior. As we mentioned earlier, decomposition is the reverse process of merging and together with merging, it allows us to produces HDL descriptions that are optimized for subsequent synthesis tasks and are relatively insensitive to coding styles. Since this decomposition uses procedure calling abstraction, arbitrary partitions of the table (condition/action) matrices are not useful. To understand the TDT structural requirements consider the example below.

Example 2.1. Consider the following TDT.

c_1	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N
c_2	Y	Y	Y	N	N	N	N	N				
c_3				Y	Y	Y	Y	N				
c_4	Y	N	N									
c_5		Y	N									
c_6				Y	Y	N	N		Y	Y	N	N
c_7				Y	N	Y	N		Y	N	Y	N
a_1	1	1	1					1	1	1	1	1
a_2	1	1	1	1	1	1	1		1	1	1	1
a_3	1		1									
a_4	1											
a_5	1	1										
a_6				1		1			1		1	
a_7				1	1	1			1	1	1	
a_8				1	1		1		1	1		1
a_9								1				

Notice the common patterns in condition rows in c_6 and c_7, and action rows in a_6, a_7, and a_8. □

Above in Example 2.1 is a flattened TDT. The first three columns have identical condition entries in c_1 and c_2, and identical action entries in a_1 and a_2. These columns differ in rows corresponding to conditions $\{c_4, c_5\}$ and actions $\{a_3, a_4, a_5\}$, which appear only in the first three columns. This may result, for example, from merging a sub-TDT consisting of only conditions $\{c_4, c_5\}$ and actions $\{a_3, a_4, a_5\}$.

Note the common pattern in the flattened TDT may result from merging a procedure which is called twice from the main program. Or it may simply correspond to commonality in the original HDL description. Whatever the cause, we can extract the common part and make it into a separate sub-TDT and execute it as an action from the main TDT.

Figure 2 shows a hierarchy of TDTs which specify the same behavior as the TDT in Example 2.1 under conditions explained later. The equivalence can be verified by merging the hierarchy of TDTs [4]. Note that the conditions and actions are partitioned among these TDTs, i.e, no conditions and actions are repeated amongs the TDTs.

TDT_1:

c_1	Y	Y	Y	N
c_2	Y	N	N	
c_3		Y	N	
a_1	1		1	1
a_2	1	1		1
TDT_2	1			
TDT_3		1		1
a_9			1	

TDT_2:

c_4	Y	N	N
c_5		Y	N
a_3	1		1
a_4	1		
a_5	1	1	

TDT_3:

c_6	Y	Y	N	N
c_7	Y	N	Y	N
a_6	1		1	
a_7	1	1	1	
a_8	1	1		1

Figure 2: One possible decomposition of the TDT in Example 2.1.

It is not always possible to decompose a given TDT into a hierarchical TDT as shown in Figure 2 above. Neither is it always valid to merge the TDT hierarchy into flattened TDT [4].

These two transformations are valid only when the specified concurrency types, data dependencies, and serializations are preserved. In this particular example, we assume that the order of execution of all actions follows the order in which they appear in the condition stub. For the transformations to be valid, we also require that:

- Actions a_1 and a_2 do not modify any values used in the evaluation of conditions c_4 and c_5.
- Actions a_1 and a_2 do not modify any values used in the evaluation of conditions c_6 and c_7.

Suppose we are given a hierarchical TDT as shown in Figure 2 to start with. After a merging phase, we get the flattened TDT as shown in Example 2.1. In the decomposition phase, we can choose to factor only TDT_3 because it is called more than once. Then the overall effect of merging followed by TDT decomposition is equivalent to in-line expansion of the procedure corresponding to TDT_2. This will not lead to any obvious improvement in hardware synthesis. However, it reduces execution delay if the description is implemented as a software component because of the overhead associated with software procedure calls.

The commonality in the flattened TDT may not result from multiple calls to a procedure as indicated by TDT_3 in Figure 2. It could also be a result of commonality in the input HDL specification. If this is the case, extraction will lead to a size reduction in the synthesized circuit.

The structural requirements for TDT decomposition can be efficiently captured by a two-level algebraic representation of TDTs [2]. This representation only captures the control dependencies in action sets and hence is strictly a sub set of TDT information. As we mentioned earlier, TDTs are based on the notion of conditions and actions. For each condition variable c, we define a positive condition literal, denoted as l_c, which corresponds to an 'Y' value in a condition entry. We also define a negative condition literal, denoted as $l_{\bar{c}}$, which corresponds to an 'N' value in a condition entry. A pair of positive and negative condition literals are related only in that they corresponds to the same condition variable in the TDT.

We define a '$\cdot$' operator between two action literals and two conditions literals which represents a conjunction operation. This operation is both commutative and associative.

A TDT is a set of rules, each of which consists of a condition part which determines when the rule is selected, and an action part which lists the actions to be executed once a rule is selected for execution. The condition part of a rule is represented as

$$\prod_{ce(i)\neq \text{'}X\text{'};\ i=1,ncond} \kappa_i \tag{1}$$

$$\kappa_i = \begin{cases} l_{c_i}, \text{when } ce(i) = \text{'Y'} \\ l_{\bar{c}_i}, \text{when } ce(i) = \text{'N'} \end{cases}$$

where *ncond* is the number of conditions in the TDT and $ce(i)$ is the condition entry value at the ith condition row for this rule. The action part of a rule is represented as

$$\prod_{ae(i)\neq \text{'}0\text{'};\ i=1,nact} l_{a_i} \tag{2}$$

where *nact* is the number of actions in the TDT and $ae(i)$ is the action entry value at the ith action row for this rule. A rule is a tuple, denoted by

$$(\mathcal{K} : \alpha)$$

As will become clear later, for the purpose of TDT decomposition a rule can be expressed as a product of corresponding action and condition literals. We call such a product a *cube*. For a given TDT, T, we define an algebraic expression, E_T, that consists of disjunction of cubes corresponding to rules in T.

For simplicity, we can drop the '$\cdot$' operator and ':' denotation and use 'c' or 'a' instead of l_c and l_a in the algebraic expressions of TDTs. However, note in particular that 'c' and '$\bar{c}$' are short-hand notations for 'l_c' and '$l_{\bar{c}}$' and they do not follow Boolean laws such as $c\bar{c} = 0$. These symbols follow only algebraic laws for symbolic computation. For treatment of this algebra, the reader is referred to [4].

Example 2.2. Here is the algebraic expression for the TDT in Example 2.1.

$$\begin{aligned}
E_{TDT\ 2.1} &= c_1c_2c_4a_1a_2a_3a_4a_5 \\
&+ c_1c_2\bar{c}_4c_5a_1a_2a_5 \\
&+ c_1c_2\bar{c}_4\bar{c}_5a_1a_2a_3 \\
&+ c_1\bar{c}_2c_3c_6c_7a_2a_6a_7a_8 \\
&+ c_1\bar{c}_2c_3c_6\bar{c}_7a_2a_7a_8 \\
&+ c_1\bar{c}_2c_3\bar{c}_6c_7a_2a_6a_7 \\
&+ c_1\bar{c}_2c_3\bar{c}_6\bar{c}_7a_2a_8 \\
&+ c_1\bar{c}_2\bar{c}_3a_1a_9 \\
&+ \bar{c}_1c_2c_3a_1a_2a_6a_7a_8 \\
&+ \bar{c}_1c_2\bar{c}_3a_1a_2a_7a_8 \\
&+ \bar{c}_1\bar{c}_2c_3a_1a_2a_6a_7 \\
&+ \bar{c}_1\bar{c}_2\bar{c}_3a_1a_2a_8
\end{aligned}$$

Note that there is no specification on delay, concurrency type, serialization relation, and data dependency. Also notice that 'c', '$\bar{c}$', and 'a' are short-hand notations for 'l_c', '$l_{\bar{c}}$', 'l_a' respectively. □

2.1 Kernel Extraction

During TDT decomposition, it is important to keep an action literal or condition literal within one sub-TDT, that is, the decomposed TDTs must partition the condition and action literals. To capture this, we introduce the notion of *support* and *TDT support*.

Definition 2.1 The **support** of an expression is the set of literals that appear in the expression.

Definition 2.2 The **TDT-support** of an expression E_T is the set of action literals and positive condition literals corresponding to all literals in the support of the expression E_T.

Example 2.3. Expression $c_1\bar{c}_2c_3\bar{c}_6\bar{c}_7a_2a_8$ is a cube. Its support is $\{c_1, \bar{c}_2, c_3, \bar{c}_6, \bar{c}_7, a_1, a_8\}$. Its TDT support is $\{c_1, c_2, c_3, c_6, c_7, a_1, a_8\}$. □

We consider TDT decomposition into sub-TDTs that have only disjoint TDT-supports. TDT decomposition uses algebraic division of TDT-expressions by using divisors to identify sub TDTs. We define the algebraic division as folllows:

Definition 2.3 Let $\{f_{dividend}, f_{divisor}, f_{quotient}, f_{remainder}\}$ be algebraic expressions. We say that $f_{divider}$ is an **algebraic divisor** of $f_{divider}$ when we have $f_{dividend} = f_{divisor} \cdot f_{quotient} + f_{remainder}$, the TDT-support of $f_{divisor}$ and the TDT-support of $f_{quotient}$ are disjoint, and $f_{divisor} \cdot f_{quotient}$ is non-empty.

An algebraic divisor is called a *factor* when the remainder is void. An expression is said to be *cube free* when it cannot be factored by a cube.

Definition 2.4 A **kernel** of an expression is a cube-free quotient of the expression divided by a cube, which is called the **co-kernel** of the expression.

Example 2.4. Rewrite the algebraic form of $TDT_{Example\ 2.1}$ as follows.

$$\begin{aligned} E_{TDT2.1} &= c_1c_2a_1a_2(c_4a_3a_4a_5 + \bar{c}_4c_5a_5 + \bar{c}_4\bar{c}_5a_3) \\ &+ c_1\bar{c}_2c_3a_2(c_6c_7a_6a_7a_8 + c_6\bar{c}_7a_7a_8 + \bar{c}_6c_7a_6a_7 + \bar{c}_6\bar{c}_7a_8) \\ &+ c_1\bar{c}_2\bar{c}_3a_1a_9 \\ &+ \bar{c}_1a_1a_2(c_6c_7a_6a_7a_8 + c_6\bar{c}_7a_7a_8 + \bar{c}_6c_7a_6a_7 + \bar{c}_6\bar{c}_7a_8) \end{aligned}$$

The expression $c_4a_3a_4a_5 + \bar{c}_4c_5a_5 + \bar{c}_4\bar{c}_5a_3$ is cube-free. Therefore it is a kernel of $TDT_{Example\ 2.1}$. The corresponding co-kernel is $c_1c_2a_1a_2$. Similarly, $c_6c_7a_6a_7a_8 + c_6\bar{c}_7a_7a_8 + \bar{c}_6c_7a_6a_7 + \bar{c}_6\bar{c}_7a_8$ is also a kernel of $TDT_{Example\ 2.1}$, which has two corresponding co-kernels: $c_1\bar{c}_2c_3a_2$ and $\bar{c}_1a_1a_2$. □

3 Algorithm for TDT Decomposition

In this section, we present an algorithm for TDT decomposition. The core of the algorithm is similar to the process of multi-level logic optimization. Therefore we first discuss how to compute algebraic kernels from TDT-expressions before we show the complete algorithm which calls the kernel computing core and addresses some important issues such as preserving data-dependencies between actions through TDT decomposition.

3.1 Algorithms for Kernel Extraction

A naive way to compute the kernels of an expression is to divide it by the cubes corresponding to the power set of its support set. The quotients that are not cube free are weeded out, and the others are saved in the kernel set [5]. This procedure can be improved in two ways: (1) by introducing a recursive procedure that exploits the property that a kernel of a kernel of an expression is also the kernel of this expression, (2) by reducing the search by exploiting the commutativity of the '$\cdot$' operator. Algorithm 3.1 shows a method adapted from a kernel extraction algorithm due to Brayton and McMullen [6], which takes into account of the above two properties to reduce computational complexity.

Algorithm 3.1 A Recursive Procedure Used in Kernel Extraction

INPUT: a TDT expression e, a recursion index j;
OUTPUT: the set of kernels of TDT expression e;

```
extractKernelR(e, j) {
    K = 0;
    for i = j to n do
        if (| getCubeSet(e, l_i)| ≥ 2) then
            C = largest cube set containing l_i s.t. getCubeSet(e, C) = getCubeSet(e, l_i);
            if (l_k ∉ C∀k < i) then
                K = K ∪ extractKernelR(e/e^C, i + 1)
    endfor
    K = K ∪ e;
    return(K);
}
```

In the above algorithm, $getCubeSet(e, C)$ returns the set of cubes of e whose support includes C. We order the literals so that condition literals appear before action literals. We use n as the index of the last condition literal since a co-kernel containing only action literals does not correspond a valid TDT decomposition. Notice that l_c and $l_{\bar{c}}$ are two different literals as we explained earlier. The algorithm is applicable to cube-free expressions. Thus, either the function e is cube-free or it is made so by dividing it by its largest cube factor, determined by the intersection of the support sets of all its cubes.

Example 3.1. After running Algorithm 3.1 on the algebraic expression of $TDT_{2.1}$ we get the following set of kernels:

$$
\begin{aligned}
k_1 &= E_{TDT\ 2.1};\\
k_2 &= c_2a_1a_2c_4a_3a_4a_5 + c_2a_1a_2\bar{c}_4c_5a_5 + c_2a_1a_2\bar{c}_4\bar{c}_5a_3\\
&+ \bar{c}_2c_3a_2c_6c_7a_6a_7a_8 + \bar{c}_2c_3a_2c_6\bar{c}_7a_7a_8 + \bar{c}_2c_3a_2\bar{c}_6c_7a_6a_7 + \bar{c}_2c_3a_2\bar{c}_6\bar{c}_7a_8;\\
k_3 &= c_4a_3a_4a_5 + \bar{c}_4c_5a_5 + \bar{c}_4\bar{c}_5a_3;\\
k_4 &= c_6c_7a_6a_7a_8 + c_6\bar{c}_7a_7a_8 + \bar{c}_6c_7a_6a_7 + \bar{c}_6\bar{c}_7a_8;\\
k_5 &= c_5a_5 + \bar{c}_5a_3;\\
k_6 &= c_7a_6 + \bar{c}_7;\\
k_7 &= c_7a_6a_7 + \bar{c}_7a_8;
\end{aligned}
$$

Note that k_6 has a cube with no action literals. This indicates a TDT rule with no action selected for execution if k_6 leads to a valid TDT decomposition. However, k_6 will be eliminated from the kernel set as we explained later. □

3.2 TDT Decomposition

Now we present a TDT decomposition algorithm which is based on the kernel extraction algorithm presented earlier. The decomposition algorithm works as follows. First, the algebraic expression of a TDT is constructed. Then a set of kernels are extracted from the algebraic expression. The

kernels are eventually used to reconstruct a TDT representation in hierarchical form. Not all the algebraic kernels may be useful in TDT decomposition since the algebraic expression carries only a subset of the TDT information. We use a set of filtering procedures to delete from the kernel sets kernels which corresponds to invalid TDT transformations or transformations producing models that results in inferior synthesis results.

Algorithm 3.2 TDT Decomposition

INPUT: a flattened TDT *tdt*;
OUTPUT: a hierarchical TDT with root *tdt'*;

```
decomposeTDT(tdt)
{
    sop ←constructAlgebraicExpression(tdt);
    K ←extractKernel(sop);
    trimKernel1(K, sop);
    trimKernel2(K, sop, td);
    trimSelf(k, sop);
    trimKernel3(k, sop);
    trimKernel4(k, sop);
    tdt' ←reconstruct_TDT_with_Kernels(tdt, K);
    return tdt'
}
```

The procedure *constructAlgebraicExpression*() builds the algebraic expression of *tdt* following Algorithm 3.2. The function *expression*() builds an expression out of a set of sets according to the data structure we choose for the two-level algrebraic expression for TDTs. The complexity of the algorithm is $O(AR + CR)$ where A is the number of action in *tdt*, R is the number of rules in *tdt*, and C is the number of conditions in *tdt*. The symbol 'ϕ' in the algorithm denotes an empty set.

Algorithm 3.3 Constructing Algebraic Expressions of TDTs

```
constructAlgebraicExpression(tdt) {
    for i = 1, C do
        construct a positive condition literal l_{c_i};
        construct a negative condition literal l_{c̄_i};
    endfor
    for i = 1, A do
        construct an action literal l_{a_i}
    endfor
    R ←φ; // empty set
    for i = 1, R do
        r ←φ;
        for j = 1, C do
            if (ce(i, j) == 'Y') then
                r ←r ∪ {l_{c_i}};
            if (ce(i, j) == 'N') then
```

```
            r ←r ∪ {l_c̄i};
        endfor;
        R ←R ∪ r;
    endfor
    return expression(R);
}
```

Procedure $extractKernel(sop)$ calls the recursive procedure $extractKernelR(sop, 1)$ to get a set of kernels of *sop*, the algebraic expression of *tdt*.

Some kernels appear only once in the algebraic expression of a TDT. These kernels would not help in reducing the resource requirement and therefore they are trimmed from K using procedure $trimKernel1()$. Algorithm 3.4 below shows the details of $trimKernel1()$. The function $co - Kernels(k, e)$ returns the set of co-kernels of kernel k for expression e. The number of co-kernels corresponds to the number of times sub-TDT that corresponds to a certain kernel is called in the hierarchy of TDTs.

Algorithm 3.4 Removing Kernels Which Correspond to Single Occurrence of a Pattern in the TDT Matrices

```
trimKernel1(K, e) {
    foreach k ∈ K do
        if (| co-Kernels(k, e) | == 1 ) then
            K ←K - {k};
    endforeach
}
```

Example 3.2. Look at the kernels in Example 3.1. The kernel $k_3 = c_4a_3a_4a_5 + \bar{c}_4c_5a_5 + \bar{c}_4\bar{c}_5a_3$ will be trimmed off by $trimKernel1()$ since it has only one co-kernel. □

Since information such as data dependency are not captured in algebraic form of TDTs, the kernels in K may not corresponds to a decomposition which preserves data-dependencies specified in the original TDT. These kernels are trimmed using procedure $trimKernel2()$.

Algorithm 3.5 Removing Kernels Which Corresponds to an Invalid TDT Transformations

```
trimKernel2(K, e, tdt) {
    foreach k ∈ K do
        flag ←0;
        foreach q ∈ co − Kernels(k, e) do
            foreach action literal l_a of q do
                if action a modifies any condition corresponding to a condition literal of k then
                    foreach action literal l_α in k do
                        if (l_a is specified to appear before l_α) then
                            flag ←1;
                    endforeach
            endforeach
```

```
        endforeach
        if (flag == 0) then
            K ←K − {k};
    endforeach
}
```

The worst case complexity of this algorithm is $O(AR+CR)$ since the program checks no more than once on each condition/action literal corresponding to a condition entry or action entry of tdt.

Example 3.3. Suppose in Example 2.1, a_2 modifies c_6 and the result of a_2 is also used a_6. Because a_2 modifies c_6, in the hierarchical TDT we need to specify that c_6 comes after TDT_2 to preserve the behavior. However, this violates the data dependency specification between a_2 and a_6. Therefore, under the condition given here, kernel $k_4 = c_6c_7a_6a_7a_8 + c_6\bar{c}_7a_7a_8 + \bar{c}_6c_7a_6a_7 + \bar{c}_6\bar{c}_7a_8$ will be removed by $trimKernel2()$. □

An expression may be a kernel of itself with a co-kernel of '1' if it is kernel free. However this kernel is not useful for TDT decomposition. We use a procedure $trimSelf()$ to delete an expression from its kernel set that will used fro TDT decomposition. Also, as we mentioned earlier, a kernel of an expression's kernel is a kernel of this expression. However, in this paper, we limit our discussion on TDT decomposition involving only two levels of calling hierarchies. For this reason, after removing an expression itself from its kernel sets, we also delete "smaller" kernels which are also kernels of other kernels of this expression.

Algorithm 3.6 Other Kernel Trimming Routines

```
trimSelf(K, exp) {
    K ←K - {exp};
}

trimKernel3(K, exp) {
    foreach k ∈ K do
        compute q and r s.t. exp = k · q + r
        if TDTsupport(k) and TDTsupport(r) are not disjoint then
            K ←K − {k};
    endforeach }

trimKernel4(K, exp) {
    foreach k ∈ K do
        foreach q ∈ K different from k do
            if q is a kernel of k then
                K ←K − {q};
        endforeach
    endforeach
}
```

Example 3.4. Look at the kernels of $E_{TDT2.1}$. Kernels k_5 will be eliminated by $trimKernel3()$ since a_5 and a_8 are also used in other cubes. For the same reason, k_6 and k_7 are also eliminated. □

Finally, we reconstruct a hierarchical TDT representation using the remaining algebraic kernels of the TDT expression. The algorithm is outlined below. It consists two procedures: *reconstruct_TDT_with_Kernel*(), and *constructTDT*() which is called by the other procedure to build a TDT out of an algebraic expression. Again, the worse case complexity of the algorithm is $O(CR + AR)$.

Algorithm 3.7 Construct a Hierarchical TDT Using Kernels

INPUT: a flattened TDT *tdt*, its algebraic expression *exp*, a set of kernels K of *exp*;
OUTPUT: a new hierarchical TDT;
re_Construct_TDT_with_Kernels(*tdt*, K, *exp*) {
  foreach $k \in K$ **do**
    $t \leftarrow$constructTDT(*tdt*, k)
    generate a new action literal l_t for t;
    compute q and r s.t. $exp = k \cdot q + r$;
    $e \leftarrow l_t \cdot q + r$;
  endforeach
  return constructTDT(*tdt*, e);
}
constructTDT(*tdt*, e) {
  form condition stub using those conditions of *tdt* each of which has at least a corresponding condition literal in e;
  form condition matrix according to the condition literals appearing in each cube of e;
  form action stub using those conditions of *tdt* each of which has at least a corresponding action literal in e and those "new" action literals corresponding to extracted sub-TDTs;
  form action matrix according to the action literals appearing in each cube of e;
  form tdt' using the above components;
  return tdt';
}

Example 3.5. Assume expression $c_6c_7a_6a_7a_8 + c_6\bar{c}_7a_7a_8 + \bar{c}_6c_7a_6a_7 + \bar{c}_6\bar{c}_7a_8$ is the only kernel left after trimming procedures are performed on the kernel set K of the algebraic expression of $TDT_{Example\ 2.1}$. A hierarchical TDT as shown in below will be constructed after running *reconstruct_TDT_with_Kernels*().

TDT_1:

c_1	Y	Y	Y	Y	Y	N
c_2	Y	Y	Y	N	N	
c_3				Y	N	
c_4	Y	N	N			
c_5		Y	N			
a_1	1	1	1		1	1
a_2	1	1	1	1		1
a_3	1		1			
a_4	1					
a_5	1	1				
TDT_3				1		1
a_9					1	

TDT_3:

c_6	Y	Y	N	N
c_7	Y	N	Y	N
a_6	1		1	
a_7	1	1	1	
a_8	1	1		1

□

4 Implementation and Experimental Results

To show the effect of using TDT decomposition in presynthesis optimizations, we have incorporated our decomposition algorithm in PUMPKIN, the TDT-based presynthesis optimization tool [4]. Figure 3 shows the flow diagram of the process presynthesis optimizations. The ellipse titled "kernel extraction" in Figure 3 show where the TDT decomposition algorithm fits in the global picture of presynthesis optimization using TDT.

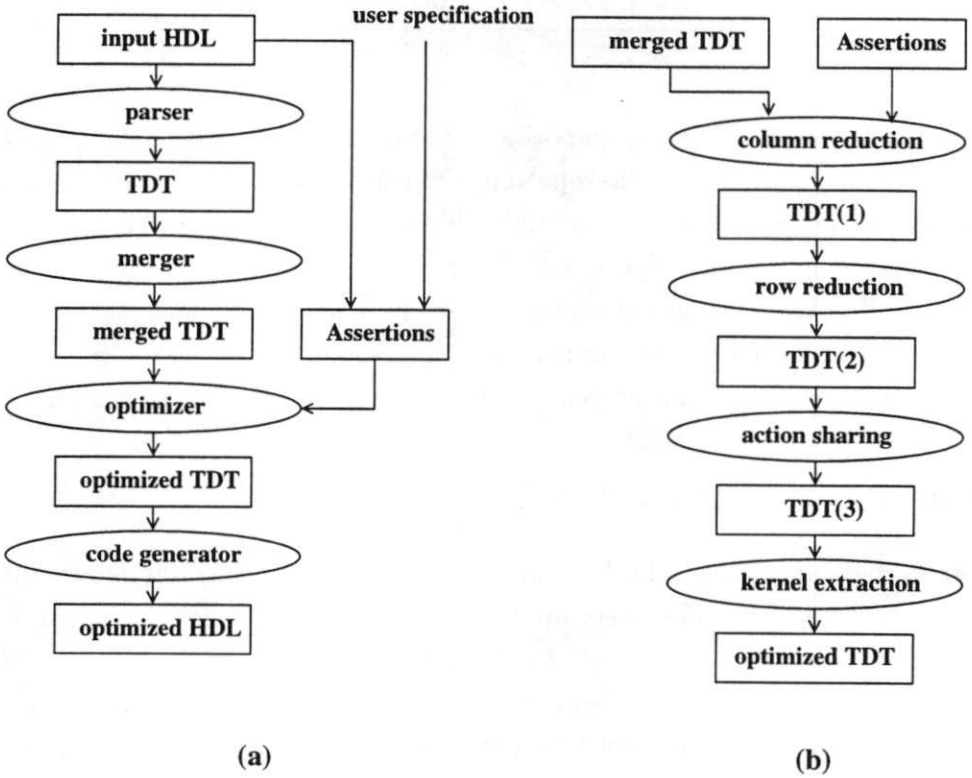

Figure 3: Flow diagram for presynthesis optimizations: (a) the whole picture, (b) details of the optimizer.

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 [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 term of number of cells. In addition to the merging algorithms, the column and row optimization algorithms originally implemented in PUMPKIN [1], we have added another optimization step of TDT decomposition. To evaluate the effectiveness of this step, we turn off column reduction, row reduction, and action sharing pahses and run PUMPKIN with several high-level synthesis benchmark designs.

Table 1: Synthesis Results: cell counts before and after TDT decomposition is carried out.

design	module	circuit size (cells)		Δ%
		before	after	
daio	phase_decoder	1252	1232	2
	receiver	440	355	19
comm	DMA_xmit	992	770	22
	exec_unit	864	587	32
cruiser	State	356	308	14

Table 1 shows the results of TDT decomposition on examples designs. The design 'daio' refers to the HardwareC design of a Digital Audio Input-Output chip (DAIO) [8]. The design 'comm' refers to the HardwareC design of an Ethernet controller [9]. The design 'cruiser' refers to the HardareC design of a vehicle controller. The description 'State' is the vehicle speed regulation module. All designs can be found in the high-level synthesis benchmark suite [7]. The percentage of circuit size reduction is computed for each description and listed in the last column of Table 1. Note that this improvement depends on the amount of commonality existing in the input behavioral descriptions.

5 Conclusion and Future Work

In this paper, we have introduced TDT decomposition as a complementary procedure to TDT merging. We have presented a TDT decomposition algorithm based on kernel extraction on an algebraic form of TDTs. Combining TDT decomposition and merging, we can restructure HDL descriptions to obtain descriptions that lead to either improved synthesis results or more efficient compiled code. Our experiment on named benchmarks shows a size reduction in the synthesized circuits after code restructuring.

Sequential Decomposition (SD) has been proposed in [10] to map a procedure to a separate hardware component which is typically specified with a process in most HDLs. Using SD, a procedure can be mapped on an off-shelf component with fixed communication protocol while a complement protocol can be constructed accordingly on the rest (synthesizable part) of the system. Therefore as a future plan of the research presented in this paper, we plan to combine SD and TDT decomposition to obtain a novel system partitioning scheme which works on tabular representations. We will investigate the possible advantages/disadvantages of this approach over other partitioning approaches.

References

[1] J. Li and R. K. Gupta, "HDL Optimization Using Timed Decision Tables," in *Proceedings of the 33rd Design Automation Conference*, pp. 51–54, June 1996.

[2] J. Li and R. K. Gupta, "Limited exception modeling and its use in presynthesis optimizations," in *Proceedings of the* 34^{th} *Design Automation Conference*, June 1997.

[3] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[4] J. Li and R. K. Gupta, "System modeling and presynthesis using timed decision tables," Tech. Rep. UCI ICS-TR-97-12, University of California, March 1997.

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

[6] R. Brayton and C. McMullen, "The decomposition and factorization of boolean expressions," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1982.

[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] M. M. Ligthard, A. Bechtolsheim, G. D. Micheli, and A. E. Gamal, "Design of a digital input output chip," in *Custom IC Conference*, May 1989.

[9] D. Ku and G. D. Micheli, *High-level Synthesis of ASICs under Timing and and Synchronization Constraints.* Kluwer Academic Publishers, 1992.

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