# Identifying Mutually Exclusive Operators in Behavioral Descriptions Using Timed Decision Tables

[†]Jian Li and [‡]Rajesh K. Gupta

Technical Report #96-47
August, 1996

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

[‡]Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425

j-li3@uiuc.edu, rgupta@ics.uci.edu

**Abstract**

Scheduling and binding are two major tasks in architectural synthesis. The information about mutually exclusive pairs of operations is very useful in reducing both the total delay of the schedule and the resource usage in the final implementation. In this paper, we present an efficient scheme which identifies all the mutually exclusive operation pairs in behavioral descriptions. Our algorithm uses data-flow analysis on a tabular model of system functionality, and is shown to work better than existing methods for identifying mutually exclusive operations.

# Identifying Mutually Exclusive Operators in Behavioral Descriptions Using Timed Decision Tables

Jian Li Rajesh K. Gupta

Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, Illinois 61801.

**Abstract**

Scheduling and binding are two major tasks in architectural synthesis. The information about the mutually exclusive pairs of operations is very useful in reducing both the total delay of the schedule and the resource usage in the final implementation. In this paper, we present an efficient scheme which identifies all the mutually exclusive operation pairs in behavioral descriptions. Our algorithm uses data-flow analysis on a tabular model of system functionality, and is shown to work better than the existing methods for identifying mutually exclusive operations.

## 1 Introduction

There are two major tasks in architectural synthesis: *scheduling* and *binding* [1]. 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* [2]. 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.

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 1.1.** Consider the following HDL description in HardwareC. It is modified from the example in [3].

```
process example(a, b, c, d, e, x, y, u, v)
in port a[8], b[8], c[8], d[8], e[8];
in port x, y;
out port u[8], v[8];
{
        static T1;
        static T2[8];
        static T3[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
                u = T3 + e;                         /* -- 7 -- */
    }
```

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

## 1.1 Related Work

In [4], Kim and Liu proposed an algorithm that can identify mutually exclusive operators based on language constructs. In [5], status bits are assigned to determine the active basic blocks. The mutual exclusiveness of two basic blocks are determined by checking the intersection of the active cube sets of their status bits. These two approaches only identifies structural m.e. pairs.

Wakabayashi and Yoshimura proposed a scheme using condition vectors (CV) [6]. This approach identifies all structural m.e. pairs and some data-flow m.e. pairs. Due to an incomplete data-flow analysis, it does not identiy all data-flow m.e. pairs. Also, due to the lack of analysis on condition dependencies in the behavioral description, it does not identify any behavioral m.e. pairs.

Path-based scheduling algorithm [7] determines the conditional usage of operators by analyzing every execution path in the control-flow graph. Operators are mutually exclusive if they do not appear in the same path. A path analysis alone identifies only structural and behavioral m.e. pairs.

Juan, Chaiyakul, and Gajski proposed condition graph to solve this problem which perform better than previous approaches. However, their approach presented in [3] also fails to identify all data-flow m.e. pairs. Furthermore, there is no efficient implementation presented and the results were obtained through a manual process.

To summarize, we list in Table 1 the results of applying all approaches we have discussed to the example in Example 1.1. Our approach to m.e. determination is indicated by column "TDT". TDT stands for Timed Decision Table, a behavioral model introduced in [8] for hardware presynthesis optimization. In this paper, we show how data-flow analysis can be combined with TDT optimizations to build an efficient algorithm for mutual exclusion determination.

| mutually exclusive operators | approaches | | | | | |
|---|---|---|---|---|---|---|
| | Kim's | SB | CV | Path-based | CG | TDT |
| $\{+_1, +_7\}$ | | | ✓ | | | ✓ |
| $\{+_2, +_3\}$ | | | | | ✓ | ✓ |
| $\{+_2, +_4\}$ | | | | | ✓ | ✓ |
| $\{+_2, +_7\}$ | | | ✓ | | ✓ | ✓ |
| $\{+_3, +_5\}$ | | | ✓ | | ✓ | ✓ |
| $\{+_3, +_6\}$ | | | ✓ | | ✓ | ✓ |
| $\{+_4, +_5\}$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $\{+_4, +_6\}$ | | | | ✓ | ✓ | ✓ |
| $\{+_4, +_7\}$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $\{+_5, +_6\}$ | | | | ✓ | ✓ | ✓ |
| $\{+_5, +_7\}$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $\{+_6, +_7\}$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 1: Mutual exclusive operatos idenfied by different approaches.

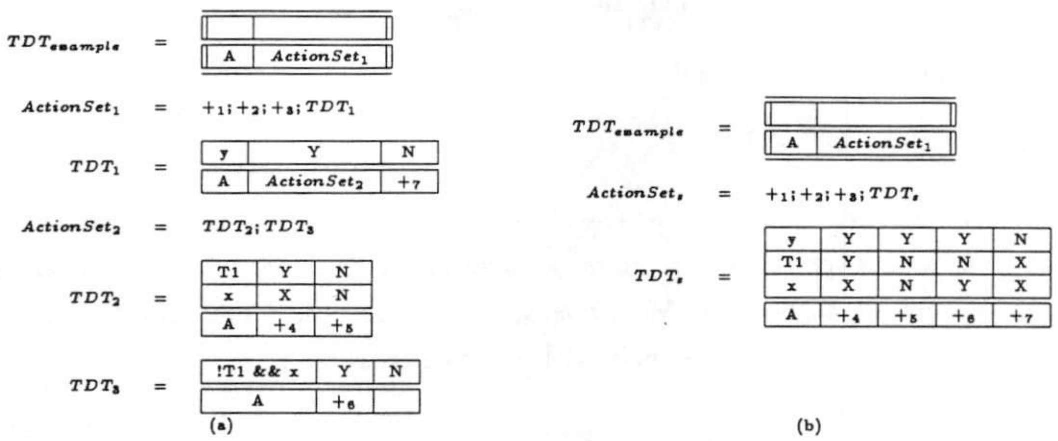
The rest of this papers is organized as follows. Section 2 gives an overview of our approach which takes three steps to identify each type of m.e. operator pairs. Section 3 shows in more details how behavioral m.e. pairs are identified. Section 4 presents a data-flow analysis based procedure for identifying data-flow m.e. pairs. We conclude in Section 5 by presenting the experimental result.

## 2 Overview of Our Apporach

The first step in our approach is to translate the input behavioral description into the TDT representation. We assume that the behavioral description is specified using a Hardware Description Language (HDL). In particular, we support input descriptions in HardwareC [9] and VHDL.

In the TDT representation, a system is modeled as interacting and concurrently executing processes. Each process is modeled as a *process TDT*, which is executed repeatedly. The body of a process TDT is modeled as hierarchically connected TDTs and *action sets*. An action set is a list of actions with a concurrency type. A set of actions are considered of the type 'data-parallel' when any two actions in an action set can be executed simultaneously unless there are data dependencies between the two actions. Other possible concurrency types that can be specified in an action sets are serial and parallel [8]. TDTs are of two kinds: *process TDTs* and *procedure TDTs*. Process TDTs represent either a process or a condition loop. Procedure TDTs represent conditional branches or nested conditional branches. Unlike a process TDT, a procedure TDT is executed only once when invoked.

In Figure 1(a), we show how the input HDL is modeled in the TDT representation. The double outlines surrounding the first table indicate that this is a process table. This table represents the HardwareC process `example` in Example 1.1. The semi-columns ';' in



TDT_example = A | ActionSet_1
ActionSet_1 = +1; +2; +3; TDT_1
TDT_1 = y | Y | N ; A | ActionSet_2 | +7
ActionSet_2 = TDT_2; TDT_3
TDT_2 = T1 | Y | N ; x | X | N ; A | +4 | +5
TDT_3 = !T1 && x | Y | N ; A | +6 |
(a)
TDT_example = A | ActionSet_1
ActionSet_e = +1; +2; +3; TDT_e
TDT_e = y | Y | Y | Y | N ; T1 | Y | N | N | X ; x | X | N | Y | X ; A | +4 | +5 | +6 | +7
(b)

Figure 1: The TDT representations of the example behavioral description: (a) the TDT representation directly converted from the input HDL, (b) the TDT merged representation.

$ActionSet_1$ and $ActionSet_2$ indicate that a *data-parallel* type is specified in both action sets. $TDT_1$ calls $ActionSet_2$ which contains $TDT_2$ and $TDT_3$. $TDT_2$ and $TDT_3$ are connected in a sequence in their enclosing action set.

When a procedure TDT is invoked for execution, the conditions are first checked to determined which action set in the corresponding column is to be executed. Take for example, when $TDT_2$ is executed, first the value of `T1` is checked. If `T1` evaluates to `FALSE`, $+_5$ is executed. Otherwise, the operation for $+_4$ is carried out. More details of the TDT model can be found in [2, 8]. Related work on tabular representations can be found in [10, 11].

In the TDT model, operators in different columns of a TDT are mutually exclusive. Thus, after converting a behavioral description into TDT, all structural m.e. pairs can be easily identified. For example, in the given HardwareC description, $\{+_4, +_5\}$ can be identified as a m.e. pair after the conversion.

The second step in our approach is merging smaller TDTs to create bigger ones. Figure 1(b) shows the merged TDT representation of the behavioral description in Example 1.1. Both structural and behavioral m.e. pairs can be identified by asserting any two different operators from different columns of a TDT as mutually exclusive. Figure 1(b) shows the merged TDT representation of the behavioral description in Example 1.1. Consider, for example, operators $+_4$ and $+_6$ from two different `if` statements in the behavioral description in Example 1.1. After merging, they appear in different columns of the same merged TDT and can be determined as a behavioral m.e. pair.

The third step in our approach performs a def-use analysis and identifies data-flow m.e. pairs. The *def* set of an operator refers to the set of operators that define a variable used in this operation. The *use* set of an operator is the set of operators that use the variable defined by this operation. In our example, we have

- *use*($+_2$) = { $+_5$, $+_6$}, and
- *use*($+_3$) = { $+_4$, $+_7$}.

Since all four pairs $\{+_5, +_4\}$, $\{+_5, +_7\}$, $\{+_6, +_4\}$, and $\{+_6, +_7\}$ are mutually exlusive, $\{+_2, +_3\}$ is a m.e. pair because in no invocation of the specified system will the results of both $+_2$ and $+_3$ be needed. M.E. operators thus identified are data-flow m.e. operators. To summarize, we list each m.e. pair with its type in Table 2.

| M.E. Pair | Type | M.E. Pair | Type |
|---|---|---|---|
| $\{+_1, +_7\}$ | data-flow | $\{+_2, +_3\}$ | data-flow |
| $\{+_2, +_4\}$ | data-flow | $\{+_2, +_7\}$ | data-flow |
| $\{+_3, +_5\}$ | data-flow | $\{+_3, +_6\}$ | data-flow |
| $\{+_4, +_5\}$ | structural | $\{+_4, +_6\}$ | behavioral |
| $\{+_4, +_7\}$ | structural | $\{+_5, +_6\}$ | behavioral |
| $\{+_5, +_7\}$ | structural | $\{+_6, +_7\}$ | structural |

Table 2: Classification of m.e. pairs.

## 3 Identification of Behavioral M.E. Pairs

To identify behavioral m.e. pairs, we merge leaf TDTs directly translated from the behavioral descriptions. 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. The merging of process TDTs follows a similar procedure.

### 3.1 Merging TDTs in a Sequence

Two procedure TDTs in a sequence can be merged if (1) they appear in an enclosing action set of concurrency type data-parallel, and (2) they share no columns except Don't Care columns or columns that contain no action sets. A Don't Care column is column that will never be selected for execution [8]. The result of merging in this case is a TDT which contains the union of the columns in the original TDTs if the two condition stubs are identical. Otherwise transformations are needed to first change the conditions stub into the same. Four transformations can be applied to a TDT for this purpose: (a) *adding a Don't Care row*, (b) *splitting a row*, (c) *negating a row*, and (d) *swapping orders of conditions.*

These transformations are part of the functionality preserving TDT transformations presented in [12]. Most of these transformations are self-explanatory, except row-splitting that is briefly discussed next. Given a binary logic operator `<op>` in an condition of form `<expr1>` `<op>` `<expr2>`, the corresponding row of this condition can be split into two rows corresponding to `<expr1>` and `<expr2>` respectively. The procedure of this splitting is shown briefly in Figure 2.

---

1. Replace the condition with $sub\_condition_1$ and $sub\_condition_2$ in the condition stub.
2. Replace each condition entry with all possible value combinations of `<exp1>` and `<exp2>` such that `<expr1>` `<op>` `<expr2>` assumes the value of this condition entry.
3. Duplicate condition entries and action entries accordingly if more than one column is obtained in Step 2.

---

Figure 2: Algorithm for row splitting.

In the following, we show one example of TDT merging that involves two TDTs in a sequence.

**Example 3.1.** The TDT sequence $\{TDT_2;TDT_3\}$ in Figure 1 satisfies the conditions for merging TDTs in a sequence. Before merging, we perform transformation (b) to convert $TDT_3$ to $TDT_3'$ and then transfomation (c) to convert$TDT_3'$ 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 | $+_4$ | $+_5$ | $+_6$ |

After merging, we have $ActionSet_2 = TDT_m$. □

## 3.2 Merging TDTs in a Hierarchy

Procedure TDTs in a hierarchy result from nested branches in behavioral HDL descriptions. In Figure 3, we present an algorithm for merging two procudure TDTs in a hierarchy. In the algorithm, *calling TDT* refers to the outmost TDT and *called TDT* refers to the inner TDT in the hierarchy. The called TDT is an action in the action set of one column in the calling TDT. There often exist dependencies among conditions, which are essentially assetions as

1. Merge the two procdure TDTs assuming conditions in two condition stub are independent.
   (a) Merge the conditions in input TDTs to form the condition stub of the resulting TDT.
   (b) Duplicate condition entries and action entries in the calling TDT according to the number of columns in the called TDT.
   (c) Add new conditon rows in the calling TDTs that correpsons to those newly-added conditions.
   (d) In the newly added rows
      - Copy values of the condition entries in the called TDT to entries in the duplicated columns resulted from Step 1(b).
      - Set the values of other condition entries to Don't Cares.
2. Identify relations among conditions and use the information obtained to reduce the size of resulting TDT from Step 1.

Figure 3: Algorithm for Merging Two Procedure TDTs.

discussed in [12]. We employ the techniques presented in [8] to reduce the size of the TDTs using assertions. After merging $TDT_2$ and $TDT_3$ to form $TDT_m$ shown in Example 3.1, $TDT_1$ and $TDT_m$ are connected in a herarchy. We can apply the merging algorithm in Figure 3 to obtain $TDT_s$ in Figure 1(b).

## 3.3 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: (a) when the action set appears before the TDT, the two can always be merged; (b) 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. The condition stub and the condition entries in the resulting TDT remains the same. The action set needs to be inserted in a proper postition in each column of the resulting table.

Special care needs to be taken to keep track of action sharing during this merging process. One way to help bookkeeping actions is to put the action stub and action entries in limited-entry form. Take $TDT_s$ in Figure 1(b) for example, the values of the condition entries can only be Boolean values 'Y' or 'N' or a Don't Care value 'X', the condition part is then said to be in *limited-entry form*, while the action entries can assume many different values, the action part is then said to be in *extended-entry from*. The action part of TDTs can also be put in limited-entry form. We show how the action part of $TDT_s$ can be put in limited-entry form in Figure 4(a). A check symbol '$\surd$' in an action entry in row $i$ and column $j$ indicates that the action (set) in row $i$ will be executed if column $j$ is selected for execution.

Putting the action part in limited-entry form makes the representation of shared action

|  |  |  |  |  |
|---|---|---|---|---|
| y | Y | Y | Y | N |
| T1 | Y | N | N | X |
| x | X | N | Y | X |
| $+_4$ | √ | - | - | - |
| $+_5$ | - | √ | - | - |
| $+_6$ | - | - | √ | - |
| $+_7$ | - | - | - | √ |

(a)

|  |  |  |  |  |
|---|---|---|---|---|
| y | Y | Y | Y | N |
| T1 | Y | N | N | X |
| x | X | N | Y | X |
| $+_2$ | √ | √ | √ | √ |
| $+_3$ | √ | √ | √ | √ |
| $+_4$ | √ | - | - | - |
| $+_5$ | - | √ | - | - |
| $+_6$ | - | - | √ | - |
| $+_7$ | - | - | - | √ |

(b)

Figure 4: TDTs with action stub and action entries arranged in Limited-entry form.

in the merging process simpler. We show the result of merging $+_2$ and $+_3$ with $TDT_s$ in Figure 4(b). Note that $+_1$ can not be merged into this TDT since it computes T1 which is a condition in $TDT_s$.

## 4 Identification of Data-flow M.E. Pairs

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.*

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

Table 3: Use sets of operators in the example in Figure 1.

Use sets of all operators in a behavioral description can be computed using standard data-flow techniques as discussed in [13]. We list the operator use sets of the description example in Table 3. An '*OUT*' indicates that the result of the operator is written to an output port or sent to another process via a messaging channel.

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. This theorem can be easily proved by following basic definitions.

**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 $\forall \alpha \in USE(o_1), \forall \beta \in USE(o_2)$, $\alpha$ and $\beta$ are mutually exclusive;*

*(b) $o_1$ and $o_2$ are mutually exclusive if $\forall \alpha \in USE(o_1)$, $\alpha$ 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 $\alpha$ and $\beta$ are not mutually exclusive;*

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

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 recursively to determine all data-flow m.e. pairs. The order to apply Theorem 4.1 is presented in the algorithm in Figure 5.

```
Create def-use graph $G = \{V, E\}$ s.t. $V = \{o | o$ is an operator $\} \cup \{OUT\}$, $E = \{ (o_1, o_2) \mid o_2 \in USE(o_1)\}$;
$Visited \leftarrow \{OUT\}$;
foreach edge $e = (o_1, o_2)$ do
    $me(o_1, o_2) \leftarrow$'N';
foreach pair $(o_1, o_2)$ s.t. $o_1$ and $o_2$ are in different columns of the same TDT do
    $me(o_1, o_2) \leftarrow$'Y';
reapeat
    Pick $o \in V - Visited$ s.t. $\forall \alpha \in Use(o)$, we have $\alpha \in Visited$;
    foreach $\beta \in Visited$ determine me$(o, \beta)$ following Theorem 4.1;
    $Visited \leftarrow Visited \cup \{o\}$;
until (all nodes in $V$ are now in $Visited$)
```

Figure 5: Algorithm for identifying data-flow m.e. pairs.

It is not difficult to see that each step of the above algorithm takes polynomial time and hence the whole algorithm takes polynomial time. Due to limited space, we leave out the detailed analysis.

## 5 Results and Discussion

Our approach for identifying mutually exclusive operations has been implemented as a part of the PUMPKIN presynthesis system [12]. We have run our system on several high-level synthesis benchmarks and behavioral description examples that appeared in previous publication. For comparison, we have also run other approaches that identifies mutually exclusive operations on the same set of behavioral descriptions. The result of our experiments has been summarized in Table 4.

| behavioral | | total # of | % of m.e. pairs identified | | | | | |
|---|---|---|---|---|---|---|---|---|
| description | # of operators | m.e. pairs | Kim's | SB | CV | path-based. | CG | TDT |
| kim | 24 | 120 | 100 | 100 | 100 | 100 | 100 | 100 |
| jian | 7 | 12 | 33 | 33 | 50 | 66 | 92 | 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 |

Table 4: The result.

The behavioral descriptions in Table 4 are either picked from previous publications or from the high-level synthesis benchmark suite. Description 'kim' refers to the example used in [4]. Description 'jian' is the example presented in this paper. Description 'juan' refers to the example used in [3]. Description 'parker' is a HardwareC example from the high-level synthesis benchmark suite.

For comparison, we have run other approaches along with ours on above mentioned examples. Kim's refers to Kim and Liu's approach [4]. Approach 'SB' stands for the status bit approach [5]. Approach 'CV' refers to condition vector approach [6]. The approach 'path-based' refers to an approach based on path analysis [7]. Approach 'CG' stands for the usage condition approach using condition graphs [3]. Finally, approach 'TDT' refers to our approach based on TDT modeling and def-use analysis.

We discuss mutual exclusiveness in the context where oeprations 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 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 can be 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.

The result in Table 4 shows that the TDT based approach performs better than previous approaches.

## References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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* 25th *Design Automation Conference*, pp. 84–87, 1988.

[6] 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.

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

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

[9] D. Ku and G. D. Micheli, "HardwareC - A Language for Hardware Design (version 2.0)," CSL Technical Report CSL-TR-90-419, Stanford University, Apr. 1990.

[10] K. Rath, M. E. Tuna, and S. D. Johnson, "Behavior tables: A basis for system representation and transformation system synthesis," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 736–740, 1993.

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

[12] J. Li and R. K. Gupta, "Timed Decision Table: A model for system representation and optimization," Technical Report UIUCDCS-R-96-1971, University of Illinois, 1996.

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