

# An Algorithm To Determine Mutually Exclusive Operations In Behavioral Descriptions

## **Abstract**

Scheduling and binding are two major tasks in architectural synthesis from behavioral descriptions. 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 circuit implementation. In this paper, we present an algorithm to identify the largest set of 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

Architectural (or high-level) synthesis attempts to build a macrolevel circuit consisting of major functional blocks and their interconnection from a given behavioral description. Two of the major tasks in architectural synthesis are operation scheduling and resource binding [1]. Scheduling determines the start time of each operation while binding maps operations to hardware components. Binding and scheduling are inter-related problems. Decisions made in binding often affect the result of scheduling and vice versa. For instance, an assignment of two operations to a functional blocks prevents placement of the operations to the same control step. 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 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. One exception to this tradeoff is in case of “mutually exclusive” operations that can share resource without increasing the total delay.

We consider two operations as mutually exclusive (m.e.) if their results are never needed at the same time. This definition subsumes previous definitions [2] as we show later. There are three different situations where the results of two operations are not needed in an execution of a behavior at the same time:

1. When two operations lie in different branches of a conditional statement, they will never need to be executed together. An operation pair that can be determined to be m.e. based on the language structures in HDL descriptions is called a *structural m.e. pair*.
2. Two operations not in the different branches of a conditional statement may still be m.e. if they lie on different control paths. Such a pair of operations is referred to as a *behavioral m.e. pair*.
3. Two operations are considered *data-flow m.e. pair* if they produce data used by operations that are pair-wise mutually exclusive. The three cases of m.e. operations are illustrated in the example below:

**Example 1.1.** Consider the following HDL description in HardwareC. It is modified from the example in [2].

```

process jian(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.  $\square$

## 1.1 Related Work

Kim and Liu [3] proposed an algorithm that can identify mutually exclusive operators based on language constructs. In [4] 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 identify structural m.e. pairs.

Wakabayashi and Yoshimura proposed a scheme using condition vectors (CV) [5]. 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 identify 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 [6] 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 [2] proposed condition graph to solve this problem which perform better than other previous approaches. However, their approach also fails to identify all data-flow m.e. pairs.

Table 1: A comparison of m.e. operator pairs identified by different approaches.

mutually exclusive operators	approaches					
	Kim's [3]	SB [4]	CV [5]	path-based [6]	CG [2]	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}	✓	✓	✓	✓	✓	✓

Table 1 summarizes the results of applying all above approaches to Example 1.1. Our approach is indicated by column “TDT”. TDT stands for Timed Decision Table, a behavioral model introduced in [7] for hardware presynthesis optimizations. In this paper, we show how data-flow analysis can be combined with TDT optimizations to build an efficient algorithm for mutual exclusion determination.

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 present the experimental result in Section 5. Finally we conclude in Section 6.

## 2 Overview of Our Approach

There are major three steps in our approach.

Step 1. 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 HDL. In particular, we support input descriptions in HardwareC [8] and VHDL.

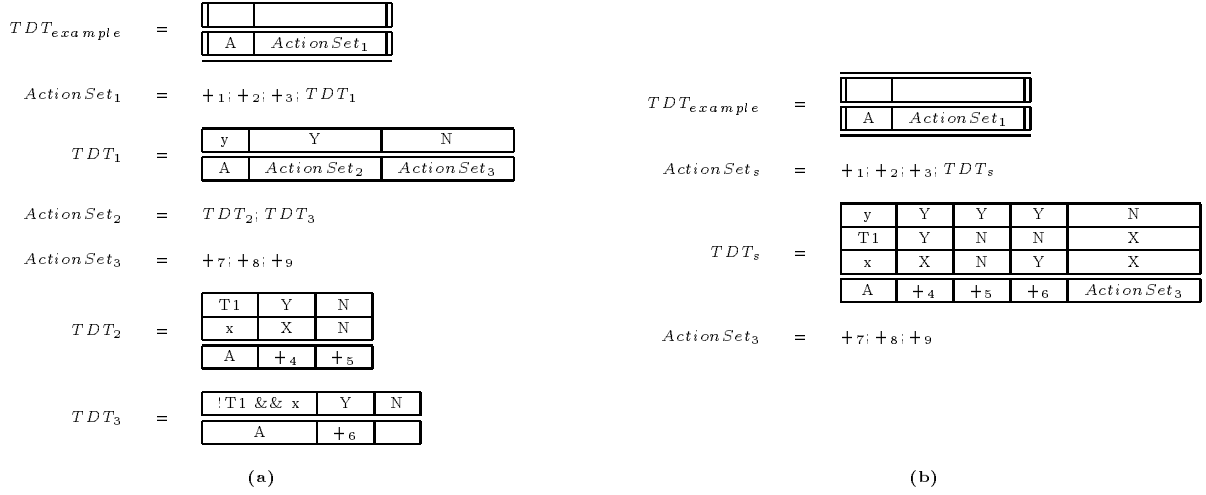


Figure 1: The TDT representations of the example behavioral description: (a) the TDT representation directly converted from the input HDL, (b) the merged TDT representation.

In the TDT representation, a hardware system is modeled as a set of interacting and concurrently executing processes. Each process is represented by a *process TDT* which is executed repeatedly. The body of a process TDT is modeled as hierarchically connected TDTs and *action sets*. In contrast to process TDTs, some other TDTs may be executed only once when they are invoked. These TDTs are called *procedure TDTs*. A TDT consists of four quadrants: condition stub, condition matrix, action stub and action matrix. A TDT represents a set of mappings from conditions to 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 [7].

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 *ActionSet<sub>1</sub>*, *ActionSet<sub>2</sub>*, and *ActionSet<sub>3</sub>* indicate that a *data-parallel* type is specified in those action sets. *TDT<sub>1</sub>* calls *ActionSet<sub>2</sub>* which contains *TDT<sub>2</sub>* and *TDT<sub>3</sub>*. *TDT<sub>2</sub>* and *TDT<sub>3</sub>* are connected in a sequence in their enclosing action set.

When a procedure TDT is invoked for execution, the conditions are first checked to determine which action set in the corresponding column is to be executed. Take for example, when *TDT<sub>2</sub>* is executed, first the value of T1 is checked. If T1 evaluates to **FALSE**, +<sub>5</sub> is executed. Otherwise, the operation for +<sub>4</sub> is carried out. More details of the TDT model

can be found in [9, 7]. 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, after the conversion, operators  $+_4$  and  $+_5$  in the given HardwareC description appear in different columns of  $TDT_2$  as shown in Figure 1(a). Therefore  $\{+_4, +_5\}$  can be identified as a m.e. operator pair.

Step 2. The second step in our approach is merging smaller TDTs to create bigger ones. After merging, both structural and behavioral m.e. pairs can be identified by asserting that any two different operators from different columns of a TDT are m.e. operators. 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. After merging, they appear in different columns of  $TDT_s$  and can be determined as a behavioral m.e. pair.

Step 3. The third step in our approach performs a def-use analysis to identify 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 exclusive,  $\{+_2, +_3\}$  is a m.e. pair because in no invocation of the specified system will the results of both  $+_2$  and  $+_3$  be needed at the same time. All 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.

Table 2: Classification of m.e. pairs.

M.E. Pair	Type	M.E. Pair	Type	M.E. Pair	Type	M.E. Pair	Type
$\{+_1, +_7\}$	data-flow	$\{+_1, +_8\}$	data-flow	$\{+_1, +_9\}$	data-flow	$\{+_2, +_3\}$	data-flow
$\{+_2, +_4\}$	data-flow	$\{+_2, +_7\}$	data-flow	$\{+_2, +_8\}$	data-flow	$\{+_2, +_9\}$	data-flow
$\{+_3, +_5\}$	data-flow	$\{+_3, +_6\}$	data-flow	$\{+_4, +_5\}$	structural	$\{+_4, +_6\}$	behavioral
$\{+_4, +_7\}$	structural	$\{+_4, +_8\}$	structural	$\{+_4, +_9\}$	structural	$\{+_5, +_6\}$	behavioral
$\{+_5, +_7\}$	structural	$\{+_5, +_8\}$	structural	$\{+_5, +_9\}$	structural	$\{+_6, +_7\}$	structural
$\{+_6, +_8\}$	structural	$\{+_6, +_9\}$	structural				

### 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 two merging cases: (I) merging TDTs in a sequence, (II) merging TDTs in a hierarchy. In this paper, we focus our discussion on the merging cases that involves only procedure TDTs, since a description with condition loops can be transformed into one without condition loops while preserving the specified system behavior [12].

#### 3.1 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. A Don't Care column is column that will never be selected for execution [7]. 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 the condition rows of a TDT for this purpose: *row insertion*, *row splitting*, *row negation*, and *row swapping*. These transformations are part of the behavior-preserving TDT transformations presented in [13]. The transformation row insertion refers to adding a row with all Don't Care entry values. The transformation row negation refers to negating a condition and the entry values in its row accordingly. Any two condition rows may be swapped without changing the specified behavior. This is referred to as row swapping. The transformation row splitting is applied to a row with a condition which is a logic expression. The procedure of this splitting is outlined in [13]. 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 row splitting to convert  $TDT_3$  to  $TDT'_3$  and then row negation to convert  $TDT'_3$  to  $TDT''_3$  as shown below.

$$TDT'_3 = \begin{array}{|c|c|c|c|} \hline T1 & Y & Y & N \\ \hline x & Y & N & X \\ \hline A & +6 & & \\ \hline \end{array} \quad TDT''_3 = \begin{array}{|c|c|c|c|} \hline T1 & N & N & Y \\ \hline x & Y & N & X \\ \hline A & +6 & & \\ \hline \end{array}$$

$TDT_2$  and  $TDT''_3$  can then be merged into  $TDT_m$  where

$$TDT_m = \begin{array}{|c|c|c|c|} \hline T1 & Y & N & N \\ \hline x & X & N & Y \\ \hline A & +4 & +5 & +6 \\ \hline \end{array}$$

After merging, we have  $ActionSet_2 = TDT_m$ .  $\square$

#### 3.2 Merging TDTs in a Hierarchy

Procedure TDTs in a hierarchy result from nested branches in behavioral HDL descriptions. Due to space limit, we refer interested reader to [13] for the detailed algorithms. In below

we give one example.

**Example 3.2.**  $TDT_1$  in Figure 1 has two action sets  $ActionSet_2$  and  $ActionSet_3$  in its two different control paths. From Example 3.1, we know that  $ActionSet_2$  is itself a TDT denoted  $TDT_m$ :

$$\begin{aligned}
 TDT_1 &= \begin{array}{|c|c|c|} \hline y & Y & N \\ \hline A & ActionSet_2 & ActionSet_3 \\ \hline \end{array} \\
 \\ 
 ActionSet_2 &= TDT_m \\
 &= \begin{array}{|c|c|c|c|} \hline T1 & Y & N & N \\ \hline x & X & N & Y \\ \hline A & +_4 & +_5 & +_6 \\ \hline \end{array}
 \end{aligned}$$

The above two tables form a calling hierarchy and they can be merged into the following table which is also denoted as  $TDT_s$  in Figure 1.

$$TDT_s = \begin{array}{|c|c|c|c|c|} \hline y & Y & Y & Y & N \\ \hline T1 & Y & N & N & X \\ \hline x & X & N & Y & X \\ \hline A & +_4 & +_5 & +_6 & ActionSet_3 \\ \hline \end{array}$$

□

As we mentioned earlier, after merging, both structural and behavioral m.e. pairs can be identified by asserting that any two different operators from different columns of a TDT are m.e. operators.

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

Data-flow m.e. pairs are identified with the help of a def-use analysis. 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 uses the variable defined by  $o$ .*

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

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

Use sets of all operators in a behavioral description can be computed using standard data-flow techniques [14]. We list the operator use sets of the example behavior description 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. All 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  $\forall \alpha \in USE(o_1), \alpha$  and  $o_2$  are mutually exclusive;*
- (b)  $o_1$  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;*
- (c)  $o_1$  and  $o_2$  are not mutually exclusive if  $\exists \alpha \in USE(o_1)$  such that  $\alpha$  and  $o_2$  are not mutually exclusive.*
- (d)  $o_1$  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;*

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

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

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

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

It is straightforward to prove (c) and (d) by following basic definitions.  $\square$

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(a) recursively to determine all data-flow m.e. pairs. The order to apply Theorem 4.1(a) is presented in the Algorithm 4.1. The rest of the Theorem can be used to prove that Algorithm 4.1 identifies the complete set of data-flow m.e. pairs.

**Algorithm 4.1 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.    $me(o_1, o_2) \leftarrow \text{'N'}$ ;
  5. **foreach** pair  $(o_1, o_2)$  **do**
  6.   **if**  $o_1$  and  $o_2$  are in different columns of a TDT **then**
  7.      $me(o_1, o_2) \leftarrow \text{'Y'}$ ;
  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(a);
  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(a).

## 5 Results and Discussion

Our approach for identifying m.e. operations has been implemented as a part of the PUMPKIN presynthesis system [15]. We have run our system on several high-level synthesis benchmarks and behavioral description examples that appeared in previous publications on detection of m.e. operations. For comparison, we have also run other approaches that identifies m.e. operations on the same set of behavioral descriptions. The result of our experiments is summarized in Table 4.

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 [3]. Description ‘jian’ is described in Example 1.1. Description ‘juan’ refers to the example used in [2]. 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 [3]. Approach ‘SB’ stands for the status

Table 4: 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 [3]	24	120	100	100	100	100	100	100
jian	9	22	45	45	55	64	86	100
juan [2]	6	7	14	14	43	43	100	100
parker	16	55	78	78	96	78	78	100
waka [5] 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

bit approach [4]. Approach ‘CV’ refers to the condition vector approach [5]. The approach ‘path-based’ refers to an approach based on path analysis [6]. Approach ‘CG’ stands for the usage condition approach using condition graphs [2]. Finally, approach ‘TDT’ refers to our approach based on TDT modeling and def-use analysis.

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 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 m.e. 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. The ‘CG’ approach outperforms all other previous approaches. However, it does not detect all data-flow m.e. pairs, especially when the result of one operation is used in a condition checking. For example, the ‘CG’ approach does not identify operator pair  $\{+_1, +_7\}$  as a m.e. pair, nor does it identify m.e. operator pairs  $\{+_1, +_8\}$  and  $\{+_1, +_9\}$ . Although it may be possible to improve the set of axioms presented in [2] to identify more data-flow pairs, the TDT approach has the following additional advantages compared to all previous approaches:

1. The TDT conversion and merging steps are also required for the presynthesis optimization which removes specification redundancies in control-flow [7]. Presynthesis optimizations are necessary since m.e. information about redundant operators is useless and creates additional work for the scheduler when considered in scheduling. Therefore, when presynthesis optimizations are carried out on input HDL descriptions, the

detection of both structural and behavioral m.e. pairs comes for “free”.

2. To detect the data-flow m.e. pairs, we have presented an algorithm which takes  $O(n^3)$  in time, where  $n$  is the number of operations in the input behavior description. This algorithm is based on def-use analysis and is easy to implement following standard techniques [16].

Information on m.e. operator pairs can be used in synthesis to obtain optimal scheduling. Consider the same example behavior description in Example 1.1. 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 5. Column ‘no’ indicates that no m.e. information has been incorporated in scheduling. Column ‘CG’ indicates that the set of m.e. operator pairs 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 5: Scheduling results when informed of different sets of m.e. pairs.

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

## 6 Conclusion and Future Work

In this paper, we have given a classification of m.e. operator pairs based on how they can be detected. We divide m.e. pairs into three categories: structural, behavioral, and data-flow. Both structural and behavioral m.e. pairs can be detected directly after input HDL description has been converted into the TDT representation and merging is carried out. Both conversion and merging are also required in the presynthesis optimization which removes specification redundancies in the control-flow. Therefore no additional work is needed when m.e. detection is carried out after presynthesis optimizations. In addition, we have presented an efficient algorithm for identifying data-flow m.e. pairs. None of the previous work on m.e. operation detection identifies all data-flow pairs. Our algorithm takes  $O(n^3)$  in time where  $n$  is the number of operators in the input HDL description.

To our knowledge, no previous work has discussed how to detect m.e. operations in behavior descriptions with condition loops. Therefore, as a future plan of this work, we will investigate the possibility of sharing resource without increasing schedule length when condition loops are involved.

## References

- [1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
- [2] 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.
- [3] 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.
- [4] 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<sup>th</sup> Design Automation Conference*, pp. 84–87, 1988.
- [5] 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.
- [6] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. CAD*, vol. 10, no. 1, pp. 85–93, 1991.
- [7] J. Li and R. K. Gupta, "HDL Optimization Using Timed Decision Tables," in *Proceedings of the 33<sup>rd</sup> Design Automation Conference*, pp. 51–54, June 1996.
- [8] 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.
- [9] 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.
- [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] R. Camposano, "Behavior-preserving transformations for high-level synthesis," in *Proc. Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, New York: Springer Verlag, 1989.
- [13] 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.
- [14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
- [15] 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.
- [16] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, ch. Code Generation, pp. 557–565. Addison Wesley, 1986.