Condition Graphs for High-Quality Behavioral Synthesis

Hsiao-Ping Juan
Viraphol Chaiyakul
Daniel D. Gajski

Technical Report #94-32
August 3, 1994

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
(714) 856-7063

hjuan@ics.uci.edu
viraphol@ics.uci.edu
gajski@ics.uci.edu

Abstract

Identifying mutual exclusiveness between operators during behavioral synthesis is important in order to reduce the required number of control steps or hardware resources. The quality of the synthesized design is strongly influenced by the number of mutually exclusive operators that can be identified by synthesis algorithms. To improve the quality of the design, we propose a representation, the *Condition Graph*, and an algorithm for comprehensive identification of mutually exclusive operators. Previous research efforts have concentrated on identifying mutual exclusiveness by examining language constructs such as IF-THEN-ELSE statements. Thus, their results heavily depend on the description styles. The proposed approach can produce results independent of description styles and identify more mutually exclusive operators than any previous approaches. The Condition Graph and the proposed algorithm can be used in any scheduling or binding algorithms. Experimental results on several benchmarks have shown the efficiency of the proposed representation and algorithm.

Contents

List of Figures

1 Introduction

High-level synthesis is a process of producing a register-transfer-level design from a given abstract behavioral description. In general, the major tasks of this process include scheduling the operators from the given behavioral description into control steps and binding the scheduled operators to appropriate resources. For example, two operators in a behavioral description may be scheduled into the same control step but performed by different resources, or they may be performed by the same resource but in different control steps. **However, if we can identify that the results of these two operators will never to be used at the same time, that is, if they are mutually exclusive, then they can be scheduled into the same control step and share the same resource.** Consequently, the number of control steps or the hardware cost is reduced. For example, consider the VHDL description shown in Figure 1(a). Assuming if we can not identify any mutually exclusive operators, then using a simple List Scheduling [4] with a hardware resource constraint of 1 adder and 1 comparator we will obtain a design of 6 control steps as shown in Figure 1(b). However, if we can identify that $+_4$ and $+_5$ are mutually exclusive because they are used in different conditional branches, we can schedule $+_4$ and $+_5$ to the same control step. Thus, one control step is reduced as shown in Figure 1(c). The number of control steps can be further reduced if we can identify more mutually exclusive operators as shown in Figure 1(d) and (e).

Mutual exclusiveness between operators can be determined by analyzing the input descriptions. Sometimes mutual exclusive operators are obvious from the use of language constructs (such as IF-THEN-ELSE), while others need a sophisticated data-flow analysis. For instance, operators $+_4$ and $+_5$ in Figure 1 are mutually exclusive because they are in different branches of the same IF-THEN-ELSE statement. Operators $+_5$ and $+_6$ are also mutually exclusive since their respective conditions, $(\overline{T1} \wedge \overline{X})$ and $(\overline{T1} \wedge X)$, will never be $TRUE$ at the same time. Moreover, from a data flow analysis we can determine that the result of operator $+_2$ ($T2$) is not used when the condition $T1$ is $TRUE$ and the result of $+_3$ ($T3$) is used only if $T1$ is $TRUE$. This means results of operators $+_2$ and $+_3$ will never be used simultaneously, in other words, $+_2$ and $+_3$ are mutually exclusive. The mutual exclusiveness between $+_4$ and $+_6$ can also be discovered from a similar data flow analysis. All possible mutually exclusive operators of example Figure 1(a) is shown in the left most

```
entity exp is
  port(a, b, c, d, e: in integer;
       x: in bit;
       y, z: out integer);
end exp;

architecture exp of exp is
begin
  process
    variable T1: bit;
    variable T2, T3: integer;
    begin
      T1 := ((a +1 b)<c);
      T2 := d +2 e;
      T3 := c +3 1;
      if T1 then
        y <= T3 +4 d;
      else if (not x) then
        y <= T2 +5 d;
      end if;
      if ((not T1) and x) then
        z <= T2 +6 e;
      end if;
  end process;
end exp;
```

(a)

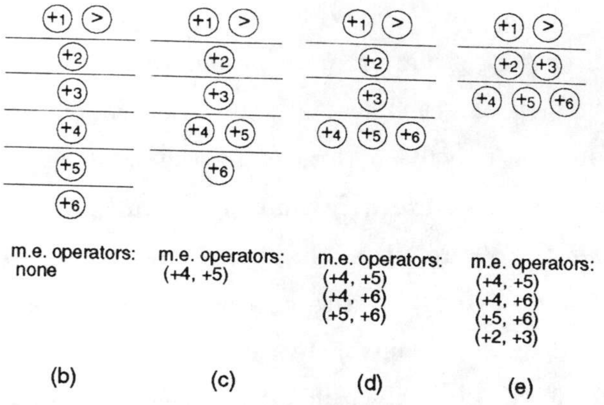
+1
>
+2
+3
+4
+5
+6
m.e. operators: none
m.e. operators: (+4, +5)
m.e. operators: (+4, +5) (+4, +6) (+5, +6)
m.e. operators: (+4, +5) (+4, +6) (+5, +6) (+2, +3)
(b)
(c)
(d)
(e)

Figure 1: An example of behavioral descriptions.

column of Figure 2.

Two previous papers addressed the issue of identifying mutually exclusive operators [6][7]. However, they identified mutually exclusive operators that are obvious from the use of language constructs. Basically, operators in different branches of the same IF or CASE are identified. For example, Kim and Liu [6] proposed an algorithm to transform a data-flow graph with conditional branches into a data-flow graph without conditional branches. This transformation is done by exploring the possibility of conditional resource sharing among the operations in conditional branches. Operators in different conditional blocks are assumed to be NOT mutually exclusive in this transformation. Thus, given the description in Figure 1, their algorithm can only identify the mutual exclusiveness between operators $+_4$ and $+_5$.

Wakabayashi and Yoshimura [7] used *condition vectors* to identify mutually exclusive operations. The condition vector is a one-hot encoding for different branches in the same conditional block. This approach can identify the mutual exclusiveness among operators in conditional branches (eg. operators $+_4$ and $+_5$). Moreover, a data-flow analysis is performed on each of the conditional branches. For instance, by applying a data-flow analysis on the first *if-then-else* statement, this approach can actually realize that $+_2$ is not used when *T1* is $TRUE$, and operators $+_2$ and $+_4$ are consequently found to be mutually exclusive. Same analysis reveals that $+_3$ and $+_5$, and $+_3$ and $+_6$ are mutually exclusive.

Some other algorithms, such as the path-based scheduling algorithm [2], determine the conditional usage of operators by analyzing every execution path in the control-flow graph. For example, there are four execution paths in the description shown in Figure 1: operators $+_1$-$+_2$-$+_3$-$+_4$, $+_1$-$+_2$-$+_3$-$+_5$, $+_1$-$+_2$-$+_3$-$+_4$-$+_6$, , and $+_1$-$+_2$-$+_3$-$+_5$-$+_6$. Operators $+_4$ and $+_5$ are mutually exclusive since they do not appear in the same path. Moreover, by applying false path analysis [1], it can be recognized that $+_1$-$+_2$-$+_3$-$+_4$-$+_6$ and $+_1$-$+_2$-$+_3$-$+_5$-$+_6$ are false paths, i.e., operators $+_4$ and $+_6$, and $+_5$ and $+_6$ are mutually exclusive. However, path analysis can not identify mutual exclusiveness between $+_3$ and $+_5$, $+_3$ and $+_6$, $+_2$ and $+_4$, and $+_2$ and $+_3$.

mutually exclusive operators	approaches		
	Kim's	Wakabayashi's	Path-based
$+_4$, $+_5$	✓	✓	✓
$+_3$, $+_5$		✓	
$+_3$, $+_6$		✓	
$+_2$, $+_4$		✓	
$+_4$, $+_6$			✓
$+_5$, $+_6$			✓
$+_2$, $+_3$			

✓ : identified

Figure 2: The result of applying previous approaches to identify mutually exclusive operators in Figure 1.

Figure 2 summarizes the result of applying previously known approaches to the example in Figure 1. One simple solution to overcome the limitations in previous approaches is to force the users to write descriptions using language constructs and description styles that can be recognized by the mutual exclusiveness identification algorithm used in the synthesis system. This solution is impractical because the users would need to acquire detailed knowledge of the algorithms used in the system.

In this paper, we propose a new approach which can identify mutually exclusive operators in a behavioral description without resorting to language constructs or styles. The overview of our approach is given in the next section. Section 3 outlines the definition and representation of the usage condition of an operator in a description. The algorithm for determining whether two operators are mutually exclusive by evaluating their usage conditions is discussed in detail in Section 4. Finally we present the results of our approach on some HLSW benchmarks and also conclusions.

2 Overview of Our Approach

The main objective of this work is to define operator usage conditions independent of description styles and to identify mutual exclusiveness of operators by evaluating their respective usage conditions.

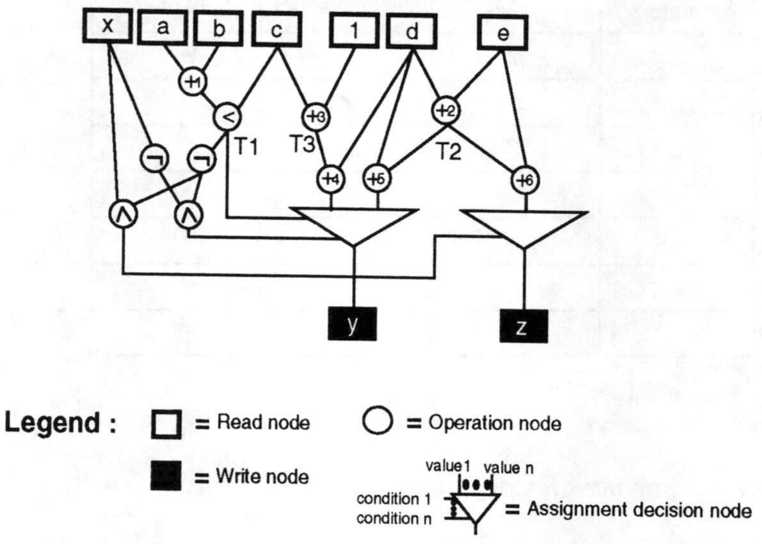

Figure 3: The ADD of the example description.

The first step in our approach is to convert the input behavioral description into an *Assignment Decision Diagram (ADD)* representation [3]. Figure 3 shows an example of the ADD representation which is derived from the description in Figure 1. The fundamental concept of the ADD is to represent a given description as a set of all possible conditional assignments to each output port or internal storage unit. For example, a part of the ADD in Figure 3 can be interpreted as "y will be assigned the value of $((c+1)+d)$ when $(a+b<c)$ is $TRUE$, or the value of $(d+(d+e))$ when $(\neg x \wedge \neg(a+b<c))$ is $TRUE$." If none of the specified conditions evaluate to $TRUE$, then y will retain its value. The property of assignment values and assignment conditions is represented as a triangular *assignment decision node (ADN)* in the ADD graph. It is the unique property of the ADN that is of interest to our work. Basically, the ADN guarantees that only one of its conditions can be $TRUE$ at a given time. Thus, the assignment values to an ADN are mutually exclusive because their values will be used in conditions which are guaranteed NOT to be $TRUE$ at the same time.

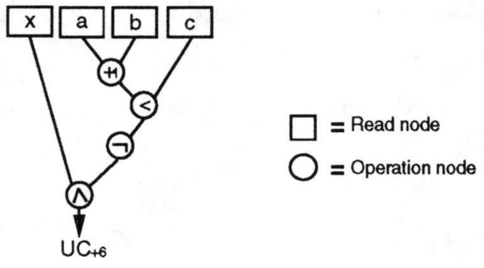

Figure 4: The CG of $+_6$.

The next step is to define and store the usage condition for each operator in the ADD. The usage condition of an operator is defined as the condition under which the result of the operator is to be used. Because the conditions for assignments are explicitly shown in an ADD, the usage conditions of operators can be easily defined in terms of assignment conditions. For example, in Figure 3, it is obvious that the result of operation $+_4$ is assigned to y only when $(a+b<c)$ is $TRUE$. And since the result of $+_3$ is only used by $+_4$, its usage condition is the same as the usage condition of $+_4$. Similarly, the usage conditions of $+_5$ and $+_6$ are $(\neg x \wedge \neg(a+b<c))$ and $(x \wedge \neg(a+b<c))$ respectively. As for $+_2$, it is realized by the data dependencies that the result of $+_2$ is needed when either $+_5$ or $+_6$ is to be executed; therefore, the usage condition of $+_2$ is $((\neg x \wedge \neg(a+b<c)) \vee (x \wedge \neg(a+b<c)))$. The usage conditions are represented and stored using a graph representation called *Condition Graph (CG)*. The CG representation is developed to minimize the space needed for storing the

usage conditions of operators and to facilitate the evaluation of usage conditions. Figure 4 shows an example of a CG which represents the usage condition of $+_6$ in Figure 3. The constituents and constructions of CGs for operators in an ADD will be discussed in greater detail in the next section.

After the CGs for all the operators in the ADD are constructed, the mutual exclusivity of any two operators can be determined by evaluating their CGs. Two operators are mutually exclusive if their CGs never evaluate to $TRUE$ simultaneously. For instance, $+_4$ and $+_5$ are mutually exclusive because their CGs, which represent $(a + b < c)$ and $(\neg x \wedge \neg(a + b < c))$ respectively, would never evaluate to $TRUE$ at the same time. Similarly, by evaluating the CGs of $+_4$ and $+_2$, which represent $(a+b < c)$ and $((\neg x \wedge \neg(a+b < c)) \vee (x \wedge \neg(a+b < c)))$, it can be identified that they are mutually exclusive.

The conversion from a behavioral description to an ADD representation has been discussed in detail in [3]. In this paper, we shall focus on how to construct CGs for the operators in an ADD and also how to identify that mutually exclusive operators by evaluating the CGs.

3 CG and Its Construction

In this section, we shall first explain how to define the usage condition for each operator in an ADD. Afterwards, we shall present how to construct CGs to represent the usage conditions of all the operators in an ADD.

3.1 Defining Usage Condition for an Operator

The usage condition for each operator in an ADD can be written as an arithmetic expression, which always evaluates to either $TRUE$ or $FALSE$. The variables in a usage condition can be bit vectors or integers. The operators in a usage condition consist of three types: (1) *arithmetic operators* such as $\{+, -, \times\}$; (2) *relational operators* such as $\{<, ==, >, \leq, \geq, \neq\}$; and (3) *Boolean operators* such as $\{\wedge, \vee, \neg\}$.

Let UC_i denote the usage condition of an operator op_i and UC_{e_i} denote the usage condition of an edge e_i. The usage condition of any operator in ADD can be defined by the following axioms:

Axiom 1 *Let $\{oe_1, oe_2, \cdots, oe_n\}$ be the set of output edges of an operator op_i (Figure 5(a)),*

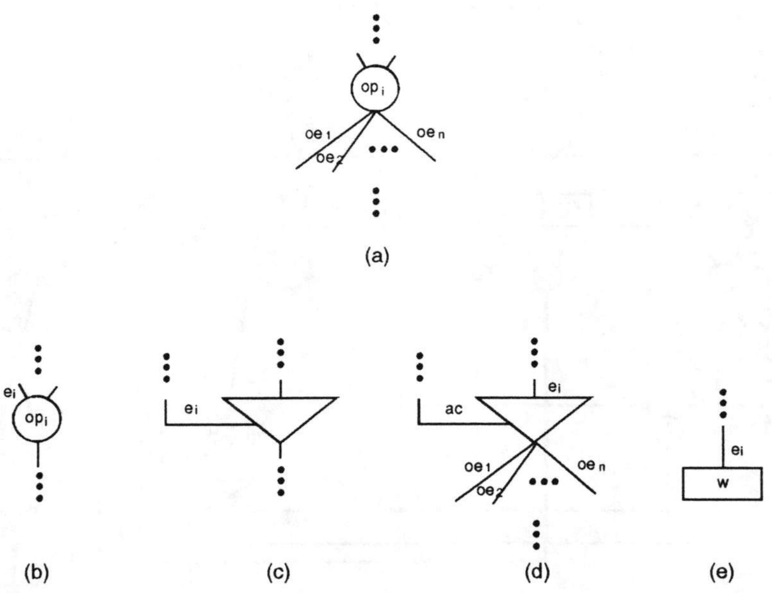

Figure 5: The illustration for defining usage conditions.

then

$$UC_i = UC_{oe_1} \vee UC_{oe_2} \vee \cdots \vee UC_{oe_n}.$$

□

The usage condition of an edge can be defined by either one of the following axioms according to the types of its sink:

Axiom 2 *If the sink of an edge e_i is an operator op_i (Figure 5(b)), then*

$$UC_{e_i} = UC_i.$$

□

Axiom 3 *If the sink of an edge e_i is an ADN,*
(1) if e_i is an assignment condition edge of the ADN (Figure 5(c)), then

$$UC_{e_i} = TRUE;$$

(2) if e_i is an assignment value edge of the ADN and its corresponding assignment condition is ac, let $\{oe_1, oe_2, \cdots, oe_n\}$ be the set of output edges of the ADN (Figure 5(d)), then

$$UC_{e_i} = ac \wedge (UC_{oe_1} \vee UC_{oe_2} \vee \cdots \vee UC_{oe_n}).$$

□

Axiom 4 *If the sink of an edge e_i is a write node w (Figure 5(e)), then*

$$UC_{e_i} = TRUE.$$

□

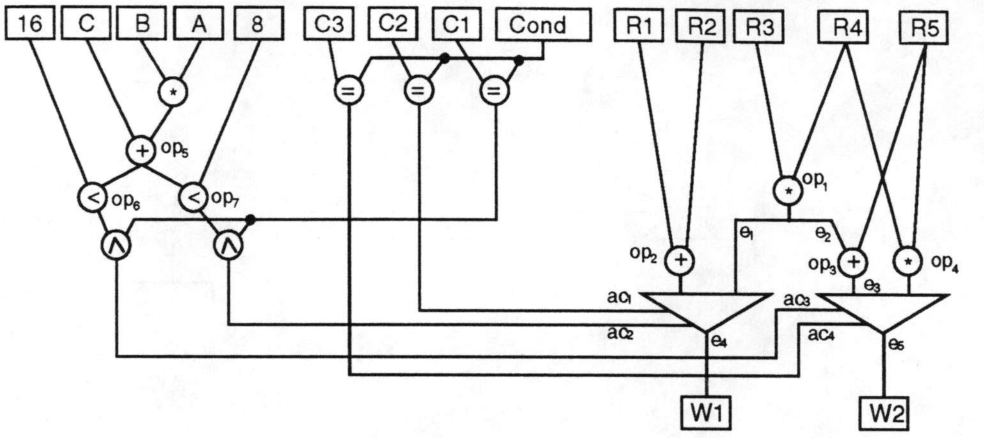

Figure 6: An example of ADD.

The usage condition of an operator can be obtained through a series of applications of the above axioms. For example, consider the operator op_1 in the ADD shown in Figure 6. According to Axiom 1, $UC_1 = UC_{e_1} \vee UC_{e_2}$. To obtain UC_{e_1}, Axiom 3 and 4 can be applied as follows:

$$\begin{aligned} UC_{e_1} &= ac_2 \wedge UC_{e_4} & \textbf{(Axiom3)} \\ &= ((A * B + C < 8) \wedge (Cond == C1)) \wedge TRUE. & \textbf{(Axiom4)} \end{aligned}$$

Similarly, UC_{e_2} can be derived as follows:

$$\begin{aligned} UC_{e_2} &= UC_3 & \textbf{(Axiom2)} \\ &= UC_{e_3} & \textbf{(Axiom1)} \\ &= ac_3 \wedge UC_{e_5} & \textbf{(Axiom3)} \\ &= ((16 < A * B + C) \wedge (Cond == C1)) \wedge TRUE. & \textbf{(Axiom4)} \end{aligned}$$

Therefore,

$$UC_1 = (((A * B + C < 8) \wedge (Cond == C1)) \wedge TRUE) \vee (((16 < A * B + C) \wedge (Cond == C1)) \wedge TRUE).$$

3.2 Constructing CGs

The CG is a graph used to represent a usage condition of an operator. A CG consists of two types of nodes (read nodes and operation nodes) and directed edges connecting the nodes. The read nodes represent the variables in the usage condition. The operation nodes represent the types of operations that are performed to compute the usage condition. Thus, a CG can be viewed as a circuit used to evaluate a usage condition and the output is the result of the evaluation.

From the previous axioms, we observe that the usage condition of an operator is defined based on the usage conditions of its output edges, which are further defined based on their sink's usage conditions; therefore, operators on the same path tend to have common sub-expressions. For example, we know that

$$\begin{aligned} UC_1 &= ((A*B+C<8)\wedge(Cond==C1))\wedge TRUE \vee (((16<A*B+C)\wedge(Cond==C1))\wedge TRUE; \\ UC_3 &= ((16<A*B+C)\wedge(Cond==C1))\wedge TRUE. \end{aligned}$$

Clearly, the usage condition of op_3 is a sub-expression of the usage condition of op_1. If the CG for each operator in an ADD is constructed individually, this would result in a large number of nodes, and many of the nodes represent the same expressions. Therefore, during the construction of CGs, we would like to share nodes as much as possible. An example of sharing nodes between the CGs of op_1 and op_3 is shown in Figure 7(c).

To achieve the sharing of common sub-graphs, the CG of an operator is constructed by using the CGs of its output edges. Similarly, the CG of an edge is constructed by using assignment conditions or the CG of its sink if it is an operator. To demonstrate the construction, we shall walk through an example of constructing the CG for op_1 in Figure 6.

According to the derivation shown in the previous section, the CGs of e_1 and e_2 have to be constructed first. The CG of e_1 is constructed by copying the assignment condition ac_2 and '$\wedge$' ac_2 with a $TRUE$ node. Figure 7(a) shows the CG of e_1. Furthermore, since $UC_{e_2} = UC_3 = UC_{e_3}$, to construct the CG of e_2 is equivalent to constructing the CG for e_3. Similar to e_1, the CG of e_3 is constructed by making a sub-graph for ac_3 first, then '$\wedge$' it with the $TRUE$ node. Note that ac_3 and ac_2 have a common sub-expression $A*B+C$; thus, ac_3 should share the sub-graph $A*B+C$ with ac_2. Figure 7(b) shows the CGs of e_1 and e_2. Now since the CGs of e_1 and e_2 have been constructed, the CG of op_1 can be

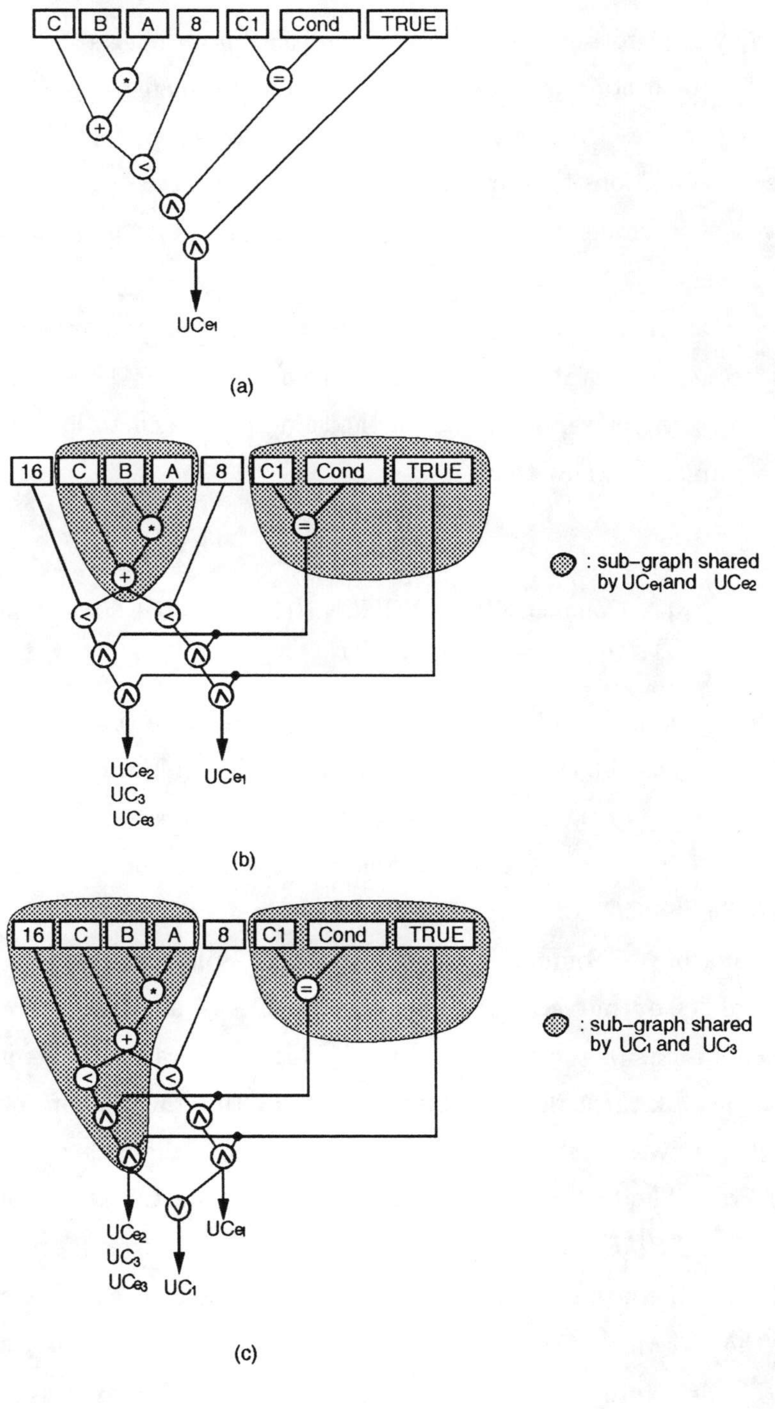
C
B
A
8
C1
Cond
TRUE
16
UC_{e1}
UC_{e2}
UC_3
UC_{e3}
UC_1
(a)
(b)
(c)
: sub-graph shared by UC_{e1} and UC_{e2}
: sub-graph shared by UC_1 and UC_3

Figure 7: The construction of CGs for op_1.

constructed by '$\vee$' the CGs of e_1 and e_2. Figure 7(c) shows the final result.

4 Identifying Mutual Exclusiveness

To identify that two operators, op_1 and op_2, are mutually exclusive we have to show that the two operators are never used at the same time. In other words, the corresponding usage conditions, UC_1 and UC_2, of the two operators will never evaluate to $TRUE$ at the same time. One simple approach to show that op_1 and op_2 are mutually exclusive is to convert UC_1 and UC_2, which could consist of arithmetic sub-expressions, into boolean equations, then prove that $\overline{UC_1 \wedge UC_2}$ will always evaluate to $TRUE$ for all possible values of variables in UC_1 and UC_2 (ie. proving that $\overline{UC_1 \wedge UC_2}$ is a tautology). However, such an approach is impractical since it requires exponential time and exponential space. This is because converting arithmetic expressions that contain variables with significantly large bitwidths (ie. 8 bits, 16 bits, etc.) to a boolean expression requires exponential space. In addition, proving a tautology requires exponential time.

Thus, for practicality, we have constructed a set of lemmas and theorems for identifying sets of mutually exclusive operators that are commonly found in a behavioral description. In addition, we an algorithm to apply the lemmas and theorems to the usage conditions, which are represented in CG, of any two given operators in order to determine whether or not they are mutually exclusive. The algorithm has a $O(n^2)$ time and space complexity, where n is the number of nodes in the CG. The main idea of the algorithm is to "pessimistically" assume that the operators are NOT mutually exclusive UNLESS the mutual exclusion can be proved by the lemmas or theorems. This "pessimistic" approach is essential for synthesis optimization even if it can not tell the exact mutually exclusive relationship between the operators. This is because by "pessimistically" identifying two operators as NOT mutually exclusive, even if they are in fact mutually exclusive, the algorithm will only degrade the optimization rather than producing an incorrect synthesized design.

4.1 Rules for identifying mutual exclusiveness

Given two operators op_1 and op_2, and thier usage conditions UC_1 and UC_2, op_1 is mutually exclusive to op_2 if and only if $UC_1 \otimes UC_2 = TRUE$, where "$\otimes$" represents the tautology statement $\overline{UC_1 \wedge UC_2} = TRUE$. The results of $\otimes$ can be determined by the following:

Lemma 1 *If UC_1 and UC_2 are assignment conditions of the same ADN, then $UC_1 \otimes UC_2 = TRUE$.*

Proof: This is derived from a property of the ADN that says "only one of the assignment conditions to the same ADN can evaluate to true at any given time."

□

Lemma 2 *if UC_i can be statically evaluated to $FALSE$ then we can remove UC_i and its corresponding assignment value from the ADD.*

Proof: If the usage condition is always $FALSE$ then the action that is guarded by the condition will never be executed. This is like a false execution/path in the description. Thus, the condition and its corresponding action can be removed from the representation without affecting the functionality of the design.

□

Lemma 3 *If UC_1 can be statically evaluated to $TRUE$ and UC_2 can not be statically computed, then $UC_1 \otimes UC_2 = FALSE$.*

Proof: Since UC_2 can not be statically computed, "pessimistically" we assume that UC_2 could have value of $TRUE$. Thus, $UC_1 \otimes UC_2 = \overline{UC_1 \wedge UC_2} = \overline{1 \wedge UC_2} = FALSE$, if UC_2 evaluates to $TRUE$.

□

Lemma 4 *$UC_1 \equiv UC_2$ (equivalent) if and only if the CG sub-graph representing UC_1 is identical (isomorphic) to the CG sub-graph representing UC_2.*

Proof: If the CG for UC_1 is identical to the CG for UC_2 then both CGs must be computing the same function. In other words, for all possible combinations of values used in the CGs, UC_1 and UC_2 would give the same value. Hence, UC_1 and UC_2 are equivalent.

□

In our application, isomorphism between two CG sub-graphs is identified in linear time by a one-to-one comparison of the CG without any transformations.

Theorem 1 *if $UC_1 \equiv UC_2$ then $UC_1 \otimes UC_2 = FALSE$.*

Proof: Following from **Lemma 4**, if UC_1 evaluates to $TRUE$ than UC_2 will be true, hence UC_1 and UC_2 can be true at the same time.

$\square$

Theorem 2 *if* $UC_1 \equiv \neg UC_2$ *then* $UC_1 \otimes UC_2 = TRUE$.

Proof: If UC_1 is a negation of UC_2 then UC_1 will be $TRUE$ if and only if UC_2 is $FALSE$, and vice versa. Hence, UC_1 and UC_2 will never be $TRUE$ at the same time.

$\square$

Theorem 3 *Given* UC_1 *and* UC_2 *that are results of relational operations such that,*

$$\begin{aligned} UC_1 &= UC_{11}\ Rop_1\ UC_{12}, \\ UC_2 &= UC_{21}\ Rop_2\ UC_{22}, \end{aligned}$$
$$\text{where } \{Rop_1,\ Rop_2 \in \{<, \leq, ==, \neq, >, \geq\}\},$$

if $(UC_{11} \equiv UC_{21})$ *and* $(UC_{12} \equiv UC_{22})$ *then* $UC_1 \otimes UC_2 = TRUE$ *if one of the following conditions, which involve* Rop_1 *and* Rop_2*, is* $TRUE$*:*

- *Rop_1 is "$<$" and Rop_2 is "$>$".*
- *Rop_1 is "$<$" and Rop_2 is "$\geq$".*
- *Rop_1 is "$\leq$" and Rop_2 is "$>$".*
- *Rop_1 is "$==$" and Rop_2 is "$<$".*
- *Rop_1 is "$==$" and Rop_2 is "$>$".*
- *Rop_1 is "$==$" and Rop_2 is "$\neq$".*

Proof: This theorem is obtained from properties of the arithmetic relation. For example, if UC_1 is a condition $(x + y < z)$ and UC_2 is a condition $(x + y > z)$ (ie., $UC_{11} = x + y$, $UC_{12} = z$, $UC_{21} = x + y$, $UC_{22} = z$, Rop_1 is $<$ and Rop_2 is $>$) then UC_1 and UC_2 are mutually exclusive.

$\square$

Theorem 4 *Given UC_1 and UC_2 that compute relational operations, as shown in* **Theorem 3**, *such that $UC_{11} \equiv UC_{21}$ but $UC_{12} \neq UC_{22}$. If UC_{12} and UC_{22} can be evaluated to constant values then $UC_1 \otimes UC_2 = TRUE$ if one of the following conditions, which involves Rop_1, Rop_2, UC_{12} and UC_{22}, is $TRUE$:*

- *Rop_1 is "$<$", Rop_2 is "$>$", and constants $UC_{12} \leq UC_{22}$.*
- *if Rop_1 is "$<$", Rop_2 is "$\geq$", and constants $UC_{12} \leq UC_{22}$.*
- *if Rop_1 is "$\leq$", Rop_2 is "$>$", and constants $UC_{12} \leq UC_{22}$.*
- *if Rop_1 is "$\leq$", Rop_2 is "$\geq$", and constants $UC_{12} < UC_{22}$.*
- *if Rop_1 is "$==$", Rop_2 is "$<$", and constants $UC_{12} \geq UC_{22}$.*
- *if Rop_1 is "$==$", Rop_2 is "$\leq$", and constants $UC_{12} > UC_{22}$.*
- *if Rop_1 is "$==$", Rop_2 is "$>$", and constants $UC_{12} \leq UC_{22}$.*
- *if Rop_1 is "$==$", Rop_2 is "$\geq$", and constants $UC_{12} < UC_{22}$.*
- *if Rop_1 is "$==$", Rop_2 is "$\neq$", and constants $UC_{12} == UC_{22}$.*
- *if Rop_1 is "$==$", Rop_2 is "$==$", and constants $UC_{12} \neq UC_{22}$.*

□

For example, if UC_1 is a condition $(x == 1)$ and UC_2 is $(x == 2)$ (ie., $UC_{11} = x$, $UC_{12} = 1$, $UC_{21} = x$, $UC_{22} = 2$, Rop_1 = "==" and Rop_2 = "=="), then $UC_1 \otimes UC_2 = TRUE$.

In the case where the usage conditions are complex boolean expressions, the mutual exclusion of the conditions can be proven by decomposing the conditions into sub-conditions and proving the exclusion on the decomposed sub-parts. The decomposition rules are as follows:

Theorem 5 *if $UC_1 \equiv UC_{11} \wedge UC_{12}$, then*

$$(UC_{11} \wedge UC_{12}) \otimes UC_2 = (UC_{11} \otimes UC_2) \vee (UC_{12} \otimes UC_2).$$

Proof:

$$\begin{aligned}(UC_{11} \wedge UC_{12}) \otimes UC_2 &= \overline{(UC_{11} \wedge UC_{12}) \wedge UC_2}\\ &= (\overline{UC_{11}} \vee \overline{UC_{12}}) \vee \overline{UC_2}\\ &= (\overline{UC_{11}} \vee \overline{UC_2}) \vee (\overline{UC_{12}} \vee \overline{UC_2})\\ &= (\overline{UC_{11} \wedge UC_2}) \vee (\overline{UC_{12} \wedge UC_2})\\ &= (UC_{11} \otimes UC_2) \vee (UC_{12} \otimes UC_2).\end{aligned}$$

□

Theorem 6 *if* $UC_1 \equiv UC_{11} \vee UC_{12}$, *then*

$$(UC_{11} \vee UC_{12}) \otimes UC_2 = (UC_{11} \otimes UC_2) \wedge (UC_{12} \otimes UC_2).$$

Proof:

$$\begin{aligned}(UC_{11} \vee UC_{12}) \otimes UC_2 &= \overline{(UC_{11} \vee UC_{12}) \wedge UC_2}\\ &= (\overline{UC_{11}} \wedge \overline{UC_{12}}) \vee \overline{UC_2}\\ &= (\overline{UC_{11}} \vee \overline{UC_2}) \wedge (\overline{UC_{12}} \vee \overline{UC_2})\\ &= (\overline{UC_{11} \wedge UC_2}) \wedge (\overline{UC_{12} \wedge UC_2})\\ &= (UC_{11} \otimes UC_2) \wedge (UC_{12} \otimes UC_2).\end{aligned}$$

□

To demonstrate the use of these lemmas and theorems, consider the usage conditions of op_2 and op_3 from Figure 6:

$$\begin{aligned}UC_2 &= (Cond == C2)\\ UC_3 &= (A * B + C > 16) \wedge (Cond == C1)\end{aligned}$$

Determining the mutual exclusion between op_2 and op_3 using the proposed lemmas and theorems can be accomplished as follows:

$$\begin{aligned}UC_2 \otimes UC_3 &= (Cond == C2) \otimes ((A * B + C > 16) \wedge (Cond == C1))\\ &= ((Cond == C2) \otimes (A * B + C > 16)) \vee ((Cond == C2) \otimes (Cond == C1)) && \textbf{(Theorem5)}\\ &= ((Cond == C2) \otimes (A * B + C > 16)) \vee 1 && \textbf{(Theorem4)}\\ &= 1\end{aligned}$$

4.2 Algorithm for Exclusiveness Detection

Given two nodes from CGs, o_i and o_j, each of which represents a usage condition of an operator node, we can determine the mutual exclusiveness between the two conditions using lemmas and theorems according to the $QueryMuEx$ shown in Figure 8.

```
Algorithm QueryMuEx($o_i, o_j$)

    Inputs: Two nodes in the CGs.
    Output: true if $o_i$ and $o_j$ are mutually exclusive; false otherwise.

begin Algorithm

    if ($ApplyLemmas(o_i, o_j) = unknown$) then
        $return(Decompose(o_j, o_i))$;
    end if;
    return $ApplyLemmas(o_i, o_j)$;

end Algorithm


Algorithm Decompose($o_i, o_j$)

    Inputs: Two nodes from the CGs.
    Output: true if $o_i$ and $o_j$ are mutually exclusive; false otherwise.

begin Algorithm

    if ($o_i = \wedge$) then
        return ($QueryMuEx(o_j, LeftPred(o_i)) \vee QueryMuEx(o_j, RightPred(o_i))$);
    else if ($o_i = \vee$) then
        return ($QueryMuEx(o_j, LeftPred(o_i)) \wedge QueryMuEx(o_j, RightPred(o_i))$);
    else return($FALSE$);
    end if

end Algorithm
```

Figure 8: Algorithm QueryMuEx.

Basically, *QueryMuEx* is a recursive procedure. Each time it is called, it checks whether any of the lemmas can be used to determine the mutual exclusiveness of o_i and o_j. If none of the lemmas is applicable, then *QueryMuEx* calls *Decompose* to decompose the o_i and/or o_j and then recursively applies *QueryMuEx* to the decomposed sub-expression.

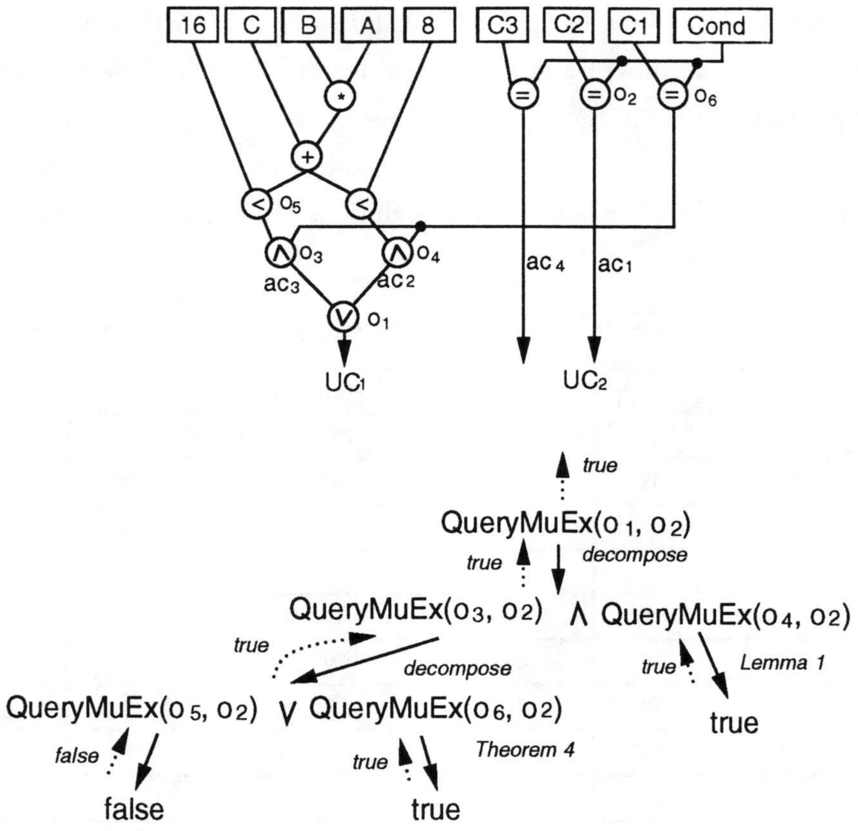

Figure 9: A walk-through example to illustrate the algorithm.

To illustrate the execution of the *QueryMuEx* algorithm, let us consider a CG graph shown in Figure 9. The mutual exclusiveness of usage conditions o_1 and o_2 can be determined by executing *QueryMuEx(*o_1*,* o_2*)*. Since *ApplyLemmas*(o_1, o_2) returns *unknown*, the algorithm decomposes o_1 into o_3 and o_4 and recursively applies lemmas to the sub-conditions. The decomposition as shown in Figure 9 is further performed until the mutual exclusiveness of the expressions can be determined. Then results from the last recursion is returned to the previous level ,as shown in Figure 9 as dotted lines, until the top-most call.

5 Experimental Results

We have tested our algorithm on several benchmarks from the High-Level Synthesis Workshop [5] and the previous publications. Results from a selected set of benchmarks, Wakabayashi's example [7], Kim's example [6], AMD2901, and AMD2910, are shown in Figure 10. For each benchmark we obtained three different VHDL behavioral descriptions. Each of the descriptions differs in the use of language construct (eg. IF-THEN-ELSE, and CASE statements) and description style (eg., grouping of conditional assignments). For example, description 1, 2 and 3 are behavioral descriptions of AMD2901 written in different styles.

Example	# of operators	total # of m.e. pairs	% of m.e. pairs detected			
			Wakabayashi's	Kim's	path-based	ours
Description 1	6	12	100 %	100 %	100 %	100 %
Description 2	6	12	0 %	0 %	100 %	100 %
Description 3	6	12	50 %	50 %	100 %	100 %
Description 4	16	45	100 %	100 %	100 %	100 %
Description 5	16	45	26.7 %	26.7 %	100 %	100 %
Description 6	14	21	100 %	76.2 %	76.2 %	100 %
Description 7	25	140	100 %	100 %	100 %	100 %
Description 8	25	140	0 %	0 %	100 %	100 %
Description 9	25	140	100 %	68.5 %	68.5 %	100 %
Description 10	27	338	100 %	100 %	100 %	100 %
Description 11	27	338	0.9 %	0.9 %	100 %	100 %
Description 12	25	286	100 %	85.3 %	85.3 %	100 %

Figure 10: Experiment results using different approaches.

For each description, we manually compute the number of operators and the total number of pairs of operators that are mutually exclusive. These numbers are shown as the #

of operators and *total # of m.e. pairs*, respectively. It should be noted that even though the total number of mutually exclusive operators are computed manually, the computation process is NOT trivial and it is time consuming. For example it took approximately 6 hours to manually compute these numbers for description 7.

Subsequently, we invoke different algorithms on each description to find all possible pairs of operators that are mutually exclusive for that example. The result of this experiment is reported in terms of the percentage of operator pairs that are found by the algorithm as compared to the number found manually (*total # of m.e. pairs*). Figure 10 shows results obtained using Kim's approach [6], Wakabayashi's approach [7], path-based scheduling approach [2], and our approach.

The results show that our approach can completely identify all possible pairs of mutually exclusive operators. On the other hand, Kim's, Wakabayashi's and path-based approach can identify all possible pairs only for certain description styles.

6 Conclusion

Identification of mutual exclusivity between operators in a behavioral description is an essential issue in optimization of hardware resources and minimization of control steps. In this paper, we have presented an approach that can identify mutually exclusive operators in a description. The proposed approach utilizes the exclusivity between conditions under which each operator is used as the basis of the identification process. Basically, operators are mutually exclusive if, and only if, they are used in conditions that will never be true at the same time. The paper presents a method of deriving operator usage-condition, *Condition Graph*, a unique and efficient representation for usage conditions, and a set of lemmas, theorems, and an algorithm for determining exclusivity of usage conditions.

We demonstrated efficiency of the proposed approach on several benchmarks of the High-level Synthesis Workshop. The results show that the proposed approach can identify all possible mutual exclusive operators in the benchmark, and out perform all previously known approaches. In addition, unlike previous approaches, the proposed approach is independent of language constructs and description styles used in the description.

7 References

[1] R.A. Bergamaschi, "The Effects of False Paths in High-Level Synthesis," *Proc. ICCAD 91*, 1991.

[2] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Trans. CAD*, Vol.10, no.1, Jan. 1991.

[3] V. Chaiyakul, D.D. Gajski and L. Ramachandran, "High-level Transformations for Minimizing Syntactic Variances," *Proc. 30th DAC*, 1993.

[4] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[5] *Benchmarks for the Sixth International Workshop on High-Level Synthesis*, 1992.

[6] T. Kim, J.W.S. Liu, and C.L. Liu, "A Scheduling Algorithm For Conditional Resource Sharing," *Proc. ICCAD 91*, 1991.

[7] K. Wakabayashi and T. Yoshimura, "Global Scheduling Independent of Control Dependencies Based on Condition Vectors," *Proc. 29th DAC*, 1992.