Limited Exception Modeling and Its Use in Presynthesis Optimizations

[†]Jian Li and [‡]Rajesh K. Gupta

Technical Report #96-48
October, 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

In behavioral descriptions, statements that allow limited control jumps, such as Verilog `disable` statements on named blocks, are often used for describing system behavior in presence of exceptions. In this paper, we extend the Timed Decision Tables (TDT), a tabular model of behavioral descriptions, to represent more general control structures including exceptions. We introduce the notion of action sharing that allows us to reduce resource requirements using existing high-level synthesis tools on descriptions with control exceptions. We present presynthesis algorithms that works on the extended TDT model and algorithms to perform action sharing in TDT models. Our experiments on well-known HardwareC benchmarks show size reduction resulting from sharing actions in the input behavioral descriptions.

Limited Exception Modeling and Its Use in Presynthesis Optimizations

Abstract

In behavioral descriptions, statements that allow limited control jumps, such as Verilog `disable` statements on named blocks, are often used for describing system behavior in presence of exceptions. In this paper, we extend the Timed Decision Tables (TDT), a tabular model of behavioral descriptions, to represent more general control structures including exceptions. We introduce the notion of action sharing that allows us to reduce resource requirements using existing high-level synthesis tools on descriptions with control exceptions. We present presynthesis algorithms that works on the extended TDT model and algorithms to perform action sharing in TDT models. Our experiments on well-known HardwareC benchmarks show size reduction resulting from sharing actions in the input behavioral descriptions.

Key words: Exception Handling. Control Flow Modeling. Timed Decision Tables. Presynthesis Optimization.

Area of Interest: 5.1, 4.4

1 Introduction

Due to the maturity of optimization and synthesis tools at logic and register transfer level, system specification is increasingly being done at behavioral level using Hardware Description Languages (HDL). Behavioral descriptions in HDLs use control-flow constructs such as conditional branches and loops, which are also commonly used in programming languages. In addition to the normal control flow constructs such as `if` statement or `while` loop, some HDLs also support mechanisms for exception handling. There are two kinds of exceptions: (1) immediate transfer of control such as a `goto` statement inside a loop, (2) interruption, in which the interrupted control flow is resumed after exception processing is completed. The exception handling is frequently used for concurrent system simulation, although its use in synthesis has been very limited. In this paper, we focus on circuit synthesis from descriptions with control exceptions.

As an example of exception modeling mechanism, consider the Verilog `disable` statement. Verilog disable statements are defined on named blocks [1]. A Verilog block is defined by a pair of `begin` and `end` and groups together one or more behavioral statements. Verilog blocks can be assigned names. The Verilog block with a name is called the *named block*. A `disable` statement is defined within a named block. Semantically, a disable statement on a named block is equivalent to a `goto` to the end of this block. Disable statements are used to break out of a loop or nested branches or to continue executing with the next iteration of the loop. `Disable` can also be used when there are are multiple processes [1]. `Disable` in multiple processes is used to model interruption. In this paper, we focus on modeling control exceptions within a single process.

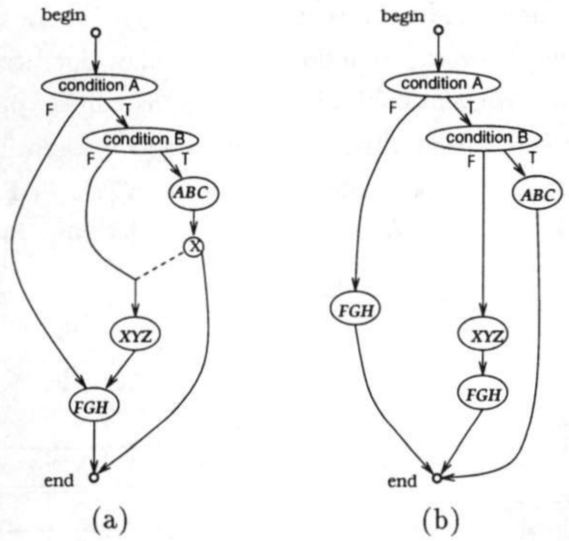
```
begin: blockA
    if (A)
      begin
        if(B)
          begin
              ABC;
              disable blockA;
          end

        XYZ;
      end

    FGH;
end;
```

Figure 1: A Verilog description with a named block and a `disable` statement.

Consider the Verilog description in Figure 1 showing a named block and a `disable` statement. Here the outermost block is named `blockA`. The statement "`disable blockA;`" breaks out of the nested branches.

When using structured programming languages such as C without `goto` statements or Verilog without **disable** statements, the scope of the blocks are statically defined and any two scopes are either nested or disjoint. In other words, blocks do not intersect each other. The control flow in such descriptions can be represented by serial-parallel (SP) graphs, or equivalently as regular expressions such as Control-flow Expression (CFE) [2].

High-level synthesis of digital systems consists of subtasks such as scheduling, resource allocation/binding and control generation. Algorithms for these subtasks are developed for input description without exceptions. Consequently, high-level synthesis systems such as ADAM [3], Olympus [4], SAW [5] and YSC [6], consider only structured inputs with very limited exception modeling such as RESET signals. The latter is incorporated in synthesized circuit by a specific choice of storage elements (flip-flop with asynchronous set/reset). In presence of control exception the input control flow is no longer structured (i.e. SP). It is possible to rewrite the description such that it can then be synthesized by duplicating appropriate actions. For instance, consider the example in Figure 1. The original non-SP control flow can be converted into SP control flow as shown below. The generated SP description can be used for synthesis. This description, however, typically results in a higher resource requirement and lead to sub-optimal synthesis results.



(a) (b)

```
// code generated from (b)

begin: blockA

  if (A)
    if (B)
      ABC;
    else
      begin
        XYZ;
        FGH; // duplicated code
      end
  else
    FGH;

end
```

(c)

Figure 2: (a) A non-SP control flow graph which corresponds to the original description with `disable`. (b) SP control flow with a duplicated node. (c) Generated description from the SP control flow graph.

In [7] we introduced a tabular representation to model structured HDL descriptions and apply behavior preserving presynthesis optimizations using assertions and behavioral Don't Cares. In this paper we extend the TDT model to include exceptions. We minimize code duplication needed for traditional high-level synthesis as shown in Figure 2(c) above by a transformation called action sharing. This extension is based on transforms in the action part of TDTs.

This paper is organized as follows. In the next section, we present the extended TDT model with an explanation of the concept of behavioral Don't Cares in TDT. In Section 3, we show the TDT-based algorithms for presynthesis optimization. In Section 4, we present the experimental results. In Section 5, we conclude and present our future plans.

2 TDT Representations with Actions in Limited-Entry Form

Tabular representations have been used for hardware modeling at different abstraction levels [7, 8, 9]. The TDT model was first introduced in [7]. It is based on the notions of condition and action. A TDT can be viewed as a set of rules. The rules are selected for execution according to the values of TDT conditions. Once a rule is selected, the action part of this rule is executed. A TDT consists of condition stub, condition entries, action stub, and action entries. The condition stub (or action stub) and condition entries (or action entries) can be arranged in either limited-entry form, or in extended-entry form. In conventional TDTs presented in [7], action stubs and action entries are arranged in extended-entry form, while condition stub and condition entries are arranged in limited-entry form. All the algorithms previously presented work with this arrangement. In a conventional TDT with the action part in extended-entry form, the action part specifies operations/action on specific variables in the action matrix. In a conventional TDT with the condition part in limited-entry from, the condition sub list all the conditions and the condition matrix is essentially a Boolean matrix. Figure 3(a) shows an behavioral description fragment taken from [7]. In Figure 3(b) we give one example of the conventional TDT representation which models the same behavior as the description in Figure 3(a).

```
if C1 {
   if C2
      a1;
   else
      a2;
}
else
   a3;
```

(a)

C1	Y	Y	N
C2	Y	N	X
A	a1	a2	a3
delay	1	2	1

(b)

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

(c)

Figure 3: (a) A behavioral description in HardwareC, (b) its TDT representation as shown in [1], (c) its representation in the extended TDT models.

To overcome the limitations of the conventional TDT model, we put the action stub and action entries also in limited-entry form, specify a more general execution semantics, and re-write our presynthesis algorithms accordingly. Figure 3(c) shows the TDT representation of 3(a) with actions re-arranged in limited-entry form. When arranged in the limited-entry form, the action stub enumerates all possible action sets, and the section for action entries is essentially a Boolean matrix. A '1' in the action entries indicates that the action set in the corresponding row will be executed when the rule in the corresponding column is selected. In contrast, a '0' in the action entries indicates that the action set in the corresponding row will not be executed when the rule in the corresponding column is selected.

The execution semantics which is an extension of [7] is as follows: (1) select the set of rules to apply, (2) execute the action that the selected rules map to. Step 1 is the same as in [7]. More than one action sets can be executed in Step 2 when one rule is selected for execution. The order of execution in such case depends on the concurrency type, data dependency, and serialization relation specified between action sets in the action stub. The concurrency type between two action sets can be serial, parallel, or data-parallel. When an concurrency type of parallel is specified between two action sets, they are invoked simultaneously if both are selected for one rule. When an concurrency type of serial is specified bwtween two action sets, a serialization order also needs to be specified between these two actions. Normally, we use the order in which these actions appear in the action stub unless the order is otherwise specified. Execution of the two action sets follow the serialization order if both action sets are selected for execution in one TDT rule. When an concurrency type of data-parallel is specified between two action sets, either a serialization order or an data-dependence relation may be specified between the two to determine the order of execution when both action sets are selected for execution in one TDT rule. If neither data dependency nor serialization order is specified, the two actions may be run in parallel. Figure 2(c) shows an example of a TDT model with action part arranged in limited-entry form. The delay part of the TDT models are often omitted since this information is not crucial for high-level synthesis or needed in the presynthesis optimization.

In below we show one example of the extended TDT representation. It models the same behavior as the Verilog description in Figure 1. Note that action set 'FGH' are shared between two control paths and that both action set 'XYZ' and action set 'FGH' will be invoked for execution once column one is selected.

Example 2.1. Consider the description fragment in Example 1.1. It can be model using a TDT with action stub and action entries in limited-entry form.

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

The conditions 'A' and 'B' are taken from the condition expressions in the original Verilog description. There are three action sets 'ABC', 'XYZ', and 'FGH'. This TDT lists all the control paths in the behavioral model. □

One of the basic operations on TDT is merging different TDTs into a larger one. Merging is used to increase the scope of TDT-based presynthesis optimization. The extended TDT model support two additional merging cases. First, a TDT with actions in limited-entry form can be merged with a following or preceding action set. Second, any two procedure TDTs in a sequence can be merged as long as proper concurrency types, data dependencies and serialization are specified among the action sets in the resulting TDT. Merging is used to increase the scope of presynthesis optimization, a better chance for merging certainly means a better chance for increasing the scope of presynthesis optimization. Example 2. 2 shows how a TDT is merged with a following action set. A TDT can always be merged with a following action set. However, to merge with a preceding action set, none of the conditions of the TDT should depend on this action set.

Example 2.2. We assume a_3 follows TDT1 in an action set of concurrency type serial. We can then merge a_3 and TDT1 to produce a new table TDT2 with the same functionality and timing semantics.

TDT1:

C	Y	N
a_1	1	0
a_2	0	1

a_3;

$\Longrightarrow$ TDT2:

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

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

Behavioral Don't Cares in TDT. Behavioral Don't Cares are widely used in presynthesis optimization [7]. The meaning of these behavioral Don't Cares is closely related to the concepts of behavior ON-set, behavior OFF-set, and behavior DC-set of an operation in a behavioral description in [10]. These concepts are well defined in the extended TDT representation.

Take an arbitrary TDT, denoted *tdt*. We first define *condition vector* as a column in the condition part of a TDT, which is essentially one set of possible values the condition variables of the TDT can assume. Then the behavior ON-set, OFF-set, and DC-set of an operation $\mathcal{O}$ are defined as follows.

- Behavior ON-set of operation $\mathcal{O}$ in *tdt* is the set of condition vectors (condition columns) in *tdt* that select $\mathcal{O}$ for execution.
- Behavior OFF-set of operation $\mathcal{O}$ in *tdt* is the set of condition vectors (condition columns) in *tdt* that select no action or actions other than $\mathcal{O}$ for execution.
- Behavior DC-set of operation $\mathcal{O}$ is the set of condition vectors (condition columns) in *tdt* that violate any one of the assertions specified on the environment or extracted from the data-flow.

The DC-set of operation $\mathcal{O}$ depends on the assertions either explicitly specified on the environment of a system by the designer or extracted from the data flow. When a column in the DC-set of an operation is selected for execution, we also say that this operation evaluates to a behavioral Don't Care, which means that the result of the computation of this operation has no effect on the behavior of the specified system. We give two examples in below to illustrate the concepts. Example 2. 3 shows a TDT with two action sets o_1 and o_2, and it explains the behavior ON-set, OFF-set and DC-set of o_1 and o_2. Example 2.4 introduces an assertion in the previous TDT and explains what is a behavioral Don't Care.

Example 2.3. Assume we have a simple behavioral model that is captured in the following TDT.

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

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

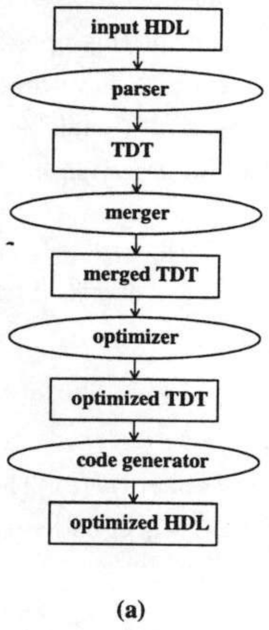
Example 2.4. Take the same TDT, but assert that c_1 is always true. The third column then violates the specified assertion. We mark the third column explicitly as shown in below and call it a Don't Care column.

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

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

3 Algorithms

In Figure 4, we show a flow diagram of presynthesis optimization using TDT models. TDT based optimization is carried out in three steps: (1) reducing the number of columns, (2) reducing the number of rows, (3) sharing identical actions. Columns reduction can be formalized into a two-level logic minimization problem and the column optimizer has been implemented by calling an efficiently-implemented two-level logic minimizer [7]. We have present a set of algorithms in [11] to carry out column reduction and row reduction. In this section, we present algorithms for sharing identical actions. We also present generalization of row and column operations for the extended TDT models with exceptions. We present algorithms following the order of presynthesis flow as show in Figure 4(a).



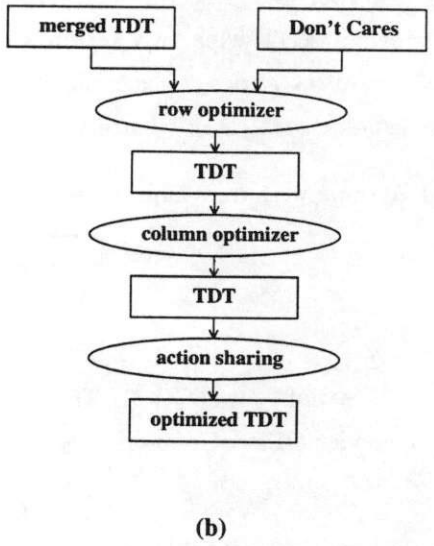


Figure 4: Flow diagram for presynthesis: (a) the whole picture, (b) details of the optimizer.

3.1 Merging TDT with Actions in Limited-entry Form

As mentioned earlier, merging is used to increase the scope of presynthesis optimization. TDTs resulting from the parsing phase are typically small. Small TDTs are merged together by recursively applying several basic merging algorithms. There are three cases: (1) merging TDTs in a sequence, (2) merging TDTs in a hierarchy, (3) merging a TDT with a preceding or following action set. When merging TDTs in a hierarchy, we refer to the outermost TDT as the calling TDT, and inner TDT as the called TDT. One common case of merging TDTs in a hierarchy is when the called TDT contains only one condition. Merging in this case is referred to as "basic merge" in [7]. We present in below an basic merge algorithm which works with TDT with actions in limited-entry form.

Algorithm 3.1 Basic Merge for TDTs with Actions in Limited-entry Form

```
modified_basic_merge( tdt, sub_tdt)
begin
    if (sub_tdt is the jth action set of tdt) and (sub_dt has only one condition) then
        condition ←(condition of sub_tdt);
        conditionList ←(conditions of tdt );
        if (condition ∈ conidtionList) then
            i ←(the index of condition in conditionList);
            J ←(the set of indices of columnes in which sub_tdt is selected for execution)
            foreach j in J do
                if ( tdt→conditionEntry[i][j] = 'Y' ) then
                    if(yes-action-set of sub_tdt not in action sets of tdt) then
                insert the yes-action-set in the action stub of tdt in the position
                    between the original jth and (j+1)th action sets;
                insert a new row in the action entries of tdt with all '0' in the position
                    between the original jth and (j+1)th rows;
            endif
            mark the corresponding action entry as '1';
        else if ( tdt→conditionEntry[i][j] = 'N') then
            if(no-action-set of sub_tdt not in action sets of tdt) then
                insert the no-action-set in the action stub of tdt in the position
                    between the original jth and (j+1)th action sets;
                insert a new row in the action entries of tdt with all '0' in the position
                    between the original jth and (j+1)th rows;
            endif
            mark the corresponding action entry as '1';
        else if ( tdt→conditionEntry[i][j] = 'X') then
            split column j;
            assign the two actions in sub_tdt to the two newly created rules;
        endif
      endforeach
  else (* if having a new condition *)
      add a new row for this new condition;
      i ←(the index of this new condition in conditionList );
      J ←(the set of indices where sub_tdt is marked '1');
      foreach j in J do
        split the column j;
        assign 'Y' 'N' respectively to the two new entries at
            row i and columns corresponding to original column j;
        assign all other new condition entries to '0';
        if(yes-action-set of sub_tdt not in action sets of tdt) then
            insert the yes-action-set in the action stub of tdt in the position
                between the original jth and (j+1)th action sets;
            insert a new row in the action entries of tdt with all '0' in the position
                between the original jth and (j+1)th rows;
        endif
        if(no-action-set of sub_tdt not in action sets of tdt) then
            insert the no-action-set in the action stub of tdt in the position
                between the original jth and (j+1)th action sets;
            insert a new row in the action entries of tdt with all '0' in the position
                between the original jth and (j+1)th rows;
        endif
        mark the two action entries corresponding to the yes-action-set and
            no-action-set with '1' respectively;
      endforeach
    endif
  endif
end
```

A TDT with only one condition is also referred to as *unit TDT*. In a unit TDT, the *yes-action-set* refers to the action set which is selected when the TDT condition evaluates to true. In contrast, the *no-action-set* refers to the action set which is selected when the TDT condition evaluates to false. The two `foreach` loops in the merging algorithms are used to support action sharing since in the extended TDT representation the called TDT may be shared by more than one control paths.

3.2 TDT Construction from HDL Descriptions with Exceptions

Translating HDL description with `disable` statements involving modifying both the parser and merger. Algorithm 3.2 shows the major changes. Since each column in a TDT represents a control path and all action sets are presented, modeling a control jump in TDTs simply requires deletion of action sets between the control jump and the jump target point in each path that contains the control jump.

Algorithm 3.2 Translate HDL Description with Disable into TDT and Perform Merging Operation

```
/* consider step 1-3 as a modified parser */
1. process disable in the same way as all other ALU statements;
2. add a special endblock statement for each named block
3. call the original parser;
/* consider step 4-7 as the modified merger */
4. call other merging algorithms;
5. foreach column with a pair of disable and endblock statements do
       mark all action sets in between as '0';
6. remove all disable statements;
7. remove all endblock statements;
```

3.3 Action Sharing

As mentioned previously, action sharing refers to identification of duplicate actions created due to modeling of exception control flow as shown in Figure 2. Action sharing is performed in two major steps: (1) searching for identical action sets, (2) merging corresponding rows in the action part of the table if merging is valid. Details of this process is shown in Algorithm 3.3. Before merging identical action sets in Step 3 of Algorithm 3.3 , we check to see if the merging violates any originally specified concurrency relations or serializations in Step 2.

Algorithm 3.3 Sharing identical action sets

```
1. Search for identical action sets in the action stub
2. Decide whether or not to merge these actions sets into a shred copy
   (a) check on the set of concurrency types, the set of serialization relations each
       copy of the identical action set has in relation to other action sets
   (b) continue with Step 3 only if these concurrency types and serialization relations are identical
       between any two copies of the identical action sets
   (c) go back to Step 1 otherwise
3. Merge action sets found in Step 1 into a shared copy
   (a) merge action entries into one row, a column of which should be marked as
       selected ('1') in this row, if the same column of any other row are marked as selected
   (b) modify the data dependence relation so that the shared copy will inherit all the original
   relations specified on the action stub
4. Repeat Step 1-3 until no more action can be shared
```

3.4 Generating HardwareC Code with Duplicated Actions Removed

To use the existing tools, we translate TDT models back into behavioral descriptions. Algorithm 3.4 shows how to generate HardwareC code from optimized TDT models. Note that when one action set is selected for execution in all paths in a sub-tdt, only one copy of the code corresponding to this action set is generated. However, as we mentioned earlier, if control jump structures are not allowed in a HDL, it is not always possible to rewrite a description without duplicating identical code segments. In cases when several but not all columns shared an action set in a TDT, often we have to duplicate the shared action sets during code generation.

Algorithm 3.4 Generating HardwareC Code from TDTs Actions in Limited-entry Form

```
procedure gencodeFromTDT(tdt)
begin
   if (there is a row in entries with all '1') then
      split tdt into $tdt_1$, $actionSet_m$, and $tdt_2$;
      call gencodeFromTDT($tdt_1$);
      call gencodeFromActionSet($actionSet_m$);
      call gencodeFromTDT($tdt_2$);
   elseif (there is a still a row with share actions) then
      separate the shared part from tdt if behavior can be preserved
      call gencodeFromActionSet on the separated action sets, call
         gencodeFromTDT on the reset of tdt;
      put two pieces of HardwareC code generated above according to the way the action set
         is separated from tdt;
   elseif (tdt is a unit TDT with one condition) then
      emit "if (condition of tdt) ";
      call gencodeFromActionSet(yes-action-set of tdt);
      emit "else";
      call gencodeFromActionSet(no-action-set of tdt);
   else
      pick in the condition entries a row with no Don't Cares;
      creates two tables $tdt_A$ and $tdt_B$ as follows
         copy all columns in tdt with 'Y' in row i to $tdt_A$;
         copy all columns in tdt with 'N' in row i to $tdt_B$;
         delete row i from $tdt_A$ and $tdt_B$;
      generate HardwareC code as follows
         emit "if ($C_i$)";
         call gencodeFromTDT($tdt_A$);
         emit "else";
         call gencodeFromTDT($tdt_B$);
   endif
end


procedure gencodeFromActionSet(actionSet)
begin
   foreach action in actionSet do
      emit proper delimiter according to the concurrency type;
      call gencodeFromAction(action);
      emit proper delimiter according to the concurrency type;
end


procedure gencodeFromAction(action)
```

```
begin
    switch (action→type)
        case TDT: call gencodeFromTDT(action→tdt);
        case ActionSet: call gencodeFromActionSet(action→subActionSet);
        case ALU: ...
        case IO: ...
        case MessagePassing: ...
        ...
    end switch
end
```

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

1. A concurrency type of data-parallel is specified on the action stub and the shared action appears as the first action set.
2. A concurrency type of data-parallel is specified on the action stub and the shared action appears as the last action set.
3. Cases that can be transformed into the above cases via behavior preserving transformations. For example, the order of two action set can be swapped in a TDT if a concurrency type of data-parallel is specified and there is no data-dependency specified between the two action sets.

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

In below we show an example to demonstrate how algorithms for action sharing (Algorithm 3.3) and HardwareC code generation (Algorithm 3.4) presented in this section can be applied.

Example 3.1. Consider the description fragment shown in (a). It is first translated into a TDT model in (b). Following Algorithm 3.3, we search for identical action sets in TDT (b) and merge them to form TDT (c). Then, following Algorithm 3.4, we generate the optimized code in (d) from TDT (c).

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

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

(b)

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

(c)

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

Here we merge the two copies of the "receive(c)" operation. □

4 Experimental Results

In addition to the merging algorithms, the column and row optimization algorithms originally implements in PUMPKIN [7], we have added another optimization step for sharing identical code. To evaluate the effectiveness of this step, we turn off column and row optimization and run PUMPKIN with several high-level synthesis benchmark designs. Our experimental methodology is as follows. The HDL description is compiled into TDT models, run through the optimizations, and finally output as a HardwareC description. This output is provided to the Olympus High-Level Synthesis System [4] for hardware synthesis under minimum area objectives. We use Olympus synthesis results to compare the effect of optimizations on hardware size on HDL descriptions. Hardware synthesis was performed for the target technology of LSI Logic 10K library of gates. Results are compared for final circuits sizes, in terms of numbers of cells used.

In Table 1 we show the results of action sharing on examples designs. Description 'comm/exec_unit' refers to the execution unit in a ethernet controller. Description 'cruiser/State' models a hardware module for speed regulation in a cruiser. Description 'i8251/xmit' is the transmit process in a HardwareC version of the 'i8251' design. Description 'daio_receiver' is the receiver part of the Digital Audio Output (DAIO) chip. Description 'frisc' refers to a simplified RISC processor. All the designs are from the high-level synthesis benchmark suite [4].

Table 1 lists the synthesized circuit sizes of each benchmark description before and after action sharing is performed. The percentage of circuit size reduction is computed for each description and listed in the last column of Table 1. Note that this improvement depends on the amount of sharable code segments in the input behavioral descriptions.

The overall effect of presynthesis optimization is to rewrite the description and remove redundancies in the input description either as a part of the original specification or as a result of assertions and behavioral Don't Cares. This task is often done by the system designer in an at-

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

Table 1: Synthesis Results: cell counts before and after shared action are identified.

tempt to arrive at an efficient implementation. However, in the case where there are sharable code segments, it is not always possible to rewrite the code to put identical code segments together in one shared copy without using control transfer structures such as Verilog `disable`.

5 Conclusion and Future Work

In this paper, we have extended the TDT representation to model exception handling and resulting action sharing. We have presented algorithms for row and column optimizations and for action sharing. Our experimental results on high-level synthesis benchmarks show a circuit size reduction 5-32% depending on the among of sharable code segments in the input behavioral descriptions.

For our future work, we plan to carry out further synthesis tasks starting from the TDT representation because of the ease it offers for modeling shared action sets. We also plan to work on the exception handling on multiple processes.

References

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

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

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

[4] G. D. Micheli, D. C. Ku, F. Mailhot, and T. Truong, "The Olympus Synthesis System for Digital Design," *IEEE Design and Test Magazine*, pp. 37–53, Oct. 1990.

[5] D. Thomas, E. Lagnese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn, *Algorithmic and Register-Transfer Level: The System Architect's Workbench.* Kluwer Academic Publishers, 1990.

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

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

[8] 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.

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

[10] 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.

[11] 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.