Limited Exception Modeling and Its Use in Presynthesis Optimizations

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

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

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 Timed Decision Tables (TDT), a tabular behavior model, 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 work on the extended TDT model and an algorithm that performs action sharing in TDT models. Our experiments on well-known HardwareC benchmarks show size reduction resulting from sharing actions in the input behavioral descriptions.

1 Introduction

Behavioral descriptions in HDLs use control-flow constructs such as conditional branches and loops. 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. The disable statements are used to break out of a loop or nested branches or to continue executing with the next iteration of the loop. A disable statement can also be used when there are multiple processes to model interruption [1]. In this paper, we focus on modeling control exceptions within a single process.

Example 1.1. Consider the following Verilog description with a named block and a disable statement.

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

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 sub-tasks such as scheduling, resource allocation/binding and control generation. Algorithms for these sub-tasks 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 Verilog description in Example 1.1. 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 leads to sub-optimal synthesis results.

![Diagram](https://via.placeholder.com/150)

Figure 1: (a) A graph which shows both the control-flow and the syntactical structure of the Verilog description in Example 1.1, (b) a derived non-SP control-flow graph, (c) a corresponding SP control-flow with a duplicated node, (d) 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 by a transformation called action sharing.

This paper is organized as follows. The next section presents the extended TDT model. Section 3 shows the TDT-based algorithms for presynthesis optimization. In Section 4, we present the experimental results. We conclude and present our future plan in Section 5.

2 TDT Representations with Actions in Limited-Entry Form

Tabular representations have been used for hardware modeling at different abstraction levels [8, 9, 7]. The TDT model was first introduced in [7]. 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 limited-entry form the stub contains a list of possible conditions (or actions) and the entry section is a matrix with binary values that selects appropriate conditions (or actions). In contrast, in extended-entry form, the entries may assume a range of values. In conventional TDTs presented in [7], action stubs and action entries are arranged in limited-entry form, while condition stub and condition entries are arranged in extended-entry form. All the algorithms previously presented work with this arrangement. Figure 2(a) shows a behavioral description fragment taken from [7]. In Figure 2(b) we give one example of the conventional TDT representation which models the same behavior as the description in Figure 2(a).

```
if C1 {  
  if C2  
    a1;  
  else    
    if (A)      
      if (B)       
        ABC;       
      else        
        XYZ;       
         // duplicated code   
      end             
    else      
      XYZ;       
         // duplicated code   
    end             
  else    
    a2;  
} else    
  a3;  
end
```

![Table](https://via.placeholder.com/150)

Figure 2: (a) A behavioral description in HardwareC, (b) its TDT representation as shown in [7], (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 2(c) shows the TDT representation of 2(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 ‘delay’ column lists the execution delay of each action in terms of number of cycles.

The execution semantics which is an extension of [7] is as follows: (1) select a rule to apply, (2) execute the action sets that the selected rule maps 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 a concurrency type of parallel is specified between two action sets, they are invoked simultaneously if both are selected for one rule. When a concurrency type of serial is specified between 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. In [10] we show how concurrency type, serialization relation, or data dependency between a pair of action sets are specified in TDT.

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

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>Y</th>
<th>Y</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>X</td>
</tr>
<tr>
<td>ABC</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>XYZ</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>FGH</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

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'. Note that action set 'FGH' is 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.

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 optimizations. The extended TDT model supports 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. 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, for merging with a preceding action set to be behavior-preserving, there should be no data-dependency between the action set and any condition in the TDT.

Example 2.2. We assume \( a_2 \) follows TDT1 in an action set of concurrency type serial. We can then merge \( a_2 \) and TDT1 to produce a new table TDT2 with the same functionality and timing semantics.

```
TDT1: | C | Y | N |
     | a1 | 1 | 0 |
     | a2 | 0 | 1 |
```

\[ a_2 \]

```
TDT2: | C | Y | N |
     | a1 | 1 | 0 |
     | a2 | 0 | 1 |
     | a3 | 1 | 1 |
```

To preserve the behavior, we specify a concurrency type of serial between \( \{a_1, a_2\} \) and \( \{a_2, a_3\} \) in the action stub of TDT2. The serialization order follows the order in which these actions appear in the action stub.

3 Algorithms

In Figure 3, we show a flow diagram of presynthesis optimizations using the TDT representation. The 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 presented a set of algorithms in [10] 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.

![Figure 3: Flow diagram for presynthesis optimizations](image)

(a) (b)

3.1 Merging TDT with Actions in Limited-entry Form

As mentioned earlier, merging is used to increase the scope of presynthesis optimizations. 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. TDTs in a hierarchy corresponds to nested
branches in HDL descriptions. 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 [10]. 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

\begin{verbatim}
modified_basic_merge(tdt, sub_tdt)
begin
  condition ← (condition of sub_tdt);
  conditionList ← (conditions of tdt);
  if (condition ∈ conditionList) then
    i ← [the index of condition in conditionList];
    J ← [the set of indices where sub_tdt is marked ‘1’];
    foreach j in J do
      if (tdt[i][j] = ‘Y’) then
        if (yes-action-set of sub_tdt ∉ action sets of tdt) then
          insert yes-action-set in the action stub of sub_tdt
          if (j+1)th action sets;
        endif
        insert 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’;
    endforeach
    else if (tdt[i][j] = ‘N’) then
      if (no-action-set of sub_tdt ∉ action sets of tdt) then
        insert no-action-set and its row similar as yes-action-set;
      endif
      mark the corresponding action entry as ‘1’;
    else if (tdt[i][j] = ‘E’) then
      assign the two actions in sub_tdt to the two
      newly created rules;
    endif
  endif
end
\end{verbatim}

This algorithm calls the parser and merger presented in [10]. The additional part for processing disable and endblock takes \(O((2A+2R)D)\), where \(R\) is the number of rules, and \(D\) is the number of disable statements.

3.3 Action Sharing

As mentioned previously, action sharing refers to reduction of duplicated actions. An algorithm is outlined below.

Algorithm 3.3 Action Sharing

\begin{verbatim}
performActionSharing(tdt)
begin
  R ← set of action rows in tdt;
  repeat
    pick \(a \in R\);
    if \(a \in R\) then
      R ← R \(\{a\}\);
    end
    foreach \(b \in R\) do
      if (identical(a, b)) then
        if (mergeable(a, b)) then
          mergeRules(a, b);
          fixSerialization(tdt[a, b];
          fixDataDpendency(tdt[a, b];
          R ← R \(\{b\}\);
        endif
      endif
    endforeach
  until (R is empty)
end
\end{verbatim}

The function identical(a, b) returns TRUE when action set \(a\) and action set \(b\) are identical. Not all identical action sets are mergeable. For example, when action sets \(a\) and \(c\) are specified as running in parallel in one control path while action sets \(b\) and \(c\) are executed in a sequence in another, action sets \(a\) and \(b\) can not be merged even if they are identical. The function mergeable(a, b)
is called to filter out those un-mergeable cases. The procedure `mergeRows(a, b)` merges the two action rows corresponding to action sets `a` and `b`. For example, row \[ A \ 1 \ 0 \ 0 \] and row \[ A \ 0 \ 0 \ 1 \] can be merged into \[ A \ 1 \ 0 \ 1 \]. After merging, the shared copy inherits all the serialization relations and data dependencies specified on each duplicated copy. Two procedures `fixSerialization()` and `fixDataDependency()` are called to modify the serialization and dependency specification in the TDT to guarantee that the behavior is preserved after action row merging. For example, if in one control path action set `a` depends on `b`, while in another control path a duplicate of `c` depends on `c`, then `fixDataDependency()` will specify a data dependency on both pair `(a, b)` and pair `(a, c)` in the resulting TDT. The complexity of this algorithm is \( O(A^2 R) \).

### 3.4 Generating HardwareC Code

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. The procedure `genCodeFromTDT` calls procedure `genCodeFromActionSet`, which calls procedure `genCodeFromAction`. Procedure `genCodeFromAction` may call either `genCodeFromActionSet` or `genCodeFromTDT` depending the type of action. Due to space constraint, we show only the procedure `genCodeFromTDT`.

**Algorithm 3.4 Generating HardwareC Code from TDTs**

```
begin
  if (there is a row in entries with all '1') then
    split tdt into tdt1, actionSet_m, and tdt2;
    call genCodeFromTDT(tdt1);
    call genCodeFromActionSet(actionSet_m);
    call genCodeFromTDT(tdt2);
  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 rest of tdt;
    put two pieces of HardwareC code generated above
    according to how 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
```

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 implemented 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 optimizations and run PUMPKIN with several high-level syn-
thesis 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 term of number of cells.

Table 1 shows 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]. 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.

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

<table>
<thead>
<tr>
<th>design</th>
<th>circuit size (cells)</th>
<th>Δ%</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm/exec_unit</td>
<td>864</td>
<td>32</td>
</tr>
<tr>
<td>before</td>
<td>587</td>
<td></td>
</tr>
<tr>
<td>after</td>
<td>921</td>
<td>12</td>
</tr>
<tr>
<td>cruiser/State</td>
<td>356</td>
<td>24</td>
</tr>
<tr>
<td>before</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>after</td>
<td>388</td>
<td></td>
</tr>
<tr>
<td>i8251/xmit</td>
<td>971</td>
<td>5</td>
</tr>
<tr>
<td>before</td>
<td>921</td>
<td></td>
</tr>
<tr>
<td>after</td>
<td>940</td>
<td></td>
</tr>
<tr>
<td>frisc</td>
<td>4353</td>
<td>9</td>
</tr>
<tr>
<td>before</td>
<td>3940</td>
<td></td>
</tr>
<tr>
<td>after</td>
<td>4064</td>
<td></td>
</tr>
</tbody>
</table>

The overall effect of presynthesis optimizations 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 attempt 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 [11].

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 5–22% depending on the amount of sharable code segments in the input behavioral descriptions.

The TDT model presented in this paper models only limited exceptions such as disable on named block in a single Verilog process. To handle process TDT which corresponds to condition loops, we have introduced additional state bits in TDT [10]. With state bits incorporated, interruption, which is modeled using disable statements involving multiple Verilog processes, can be modeled in TDT as a column, or a rule, where the state bits take Don’t Care values. As a future plan of this research, we will investigate the modeling of interruption in TDT and how it can be used in synthesis optimizations and/or high-level synthesis.

Acknowledgment. This research is supported in part by NSF CAREER Award MIP 95-01615 and an FMC Education Fund Fellowship. We would like to thank Mr. Tai Ly for his discussion with us on our previous work, which leads to some initial ideas in this research.

References