HDL Code Restructuring Using Timed Decision Tables

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

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

http://www.ics.uci.edu/~iesag

Abstract

Behavioral HDL descriptions from designers are structured in a program calling hierarchy for the purposes of programming convenience and conceptualization. This code structure is often not suitable for direct synthesis into digital hardware. For instance, information regarding operation exclusivity and resource sharing can be explored by restructuring the code. In this paper, we present a method to restructure behavioral HDL code using merging and decomposition of Timed Decision Tables (TDTs).

1 Introduction

With the maturity of synthesis tools at various levels of design abstractions, hardware design is getting closer to software programming [1]. As in programming, designers often use language level structures such as subprograms (functions/procedure calls) to organize the HDL descriptions. This structure in HDL descriptions is useful for programming convenience and design conceptualization.

However, the subprogram calling hierarchy most suitable for design development and design comprehension is not always suitable for synthesis. In this paper, we propose a method to change the subprogram hierarchy in designer-specified HDL descriptions into one that is more suitable for synthesis. The method is implemented using a tabular model called Timed Decision Tables (TDTs). We first use TDT merging to flatten subprogram calling hierarchy. Then we use TDT decomposition to re-build a calling hierarchy under cost matrices related to operation/subprogram size and calling frequency.

Due to space limitation, we focus on use of TDT transformations rather than transformation algorithms. Specifically, we use TDT merging and decomposition transformations. We explain concepts of TDT modeling, TDT merging, and TDT decomposition using examples in HardwareC [2].

The rest of this paper is organized as follows. Section 2 introduces the TDT model and merging and decomposition transformations on TDTs. Section 3 demonstrates code restructuring using an example followed by a discussion of implementation and results.

2 Hierarchy Merging & Decomposition

In this section we first briefly introduce the TDT model before we present TDT merging and decomposition.

2.1 The TDT Model

The TDT representation models a hardware component with a set of three tables: a *control table*, a *dependency table*, and a *delay table*. The control table models the control-flow of input HDL description. It is organized in the form of decision tables [3, 4]. A decision table consists of four quadrants: *condition stub, condition entries, action stub,* and *action entries.* The condition stub list conditions which are condition checking in the conditional branches and conditional loops in the input HDL. The action stub list actions which are operations in the behavioral code. Condition entries and action entries are two matrices, each column of which forms a control path. The dependency table specifies information such as data-dependency, concurrency type [2], and serialization relation between each pair of operations. The delay table lists the execution delay associated with each action. More details of TDT can be found in [5].

Example 2.1. Consider the following control table of a TDT model with two action sets a_2 and a_3 in its two different control paths. In this simple case we ignore the dependency table and take the control table as the whole representation.

TDT_1 =

c_1	1	0
a_1	1	0
a_2	0	1

Note that a '1' in the condition part indicates that condition c_1 takes a logic value TRUE while a '0' in the condition indicates

a logic value FALSE. A '1' in the action part indicates that the action in the same row is selected for execution while a '0' indicates that the corresponding action is not selected for execution in this control path. □

A TDT may call another TDT as its action. This forms a hierarchical TDT representation.

Example 2.2. Now suppose a_1 in Example 1.1. is models by TDT_2 below.

a_1 = TDT_2 =

c_2	1	0	0
c_3	-	1	0
a_3	1		
a_4		1	
a_5			1

Then we have a hierarchical TDT representation consisting of TDT_1 calling TDT_2. Note that for clarity, we have dropped the '0's in the action part. □

There are two types of TDTs: *procedure TDTs*, and *process TDTs*. A procedure TDT corresponds to nested conditional branches or a subprogram (procedure/function). It is executed only once every time it is invoked. Both TDT_1 and TDT_2 are examples of procedure TDTs. A process TDT corresponds to a process or a loop in a behavioral HDL. It is executed repeatedly once invoked. A process TDT representing a conditional loop exits at a certain point while a process TDT representing a process runs indefinitely.

Example 2.3. A simple loop "while(c) a; " may be represented as a process TDT:

S	0	0
c	1	0
a	1	
Sn	0	1

S is a state variable. The double outlines indicate that this is a process table. It starts when $S == 0$ and stops when $S == 1$. □

2.2 Transformations on TDTs

This section briefly describes important TDT transformations for hierarchy optimization. There are two categories of TDT transformations: (1) transformations involving multiple TDTs, and (2) transformations on a single TDT.

Transformations involving multiple TDTs include TDT merging and TDT decomposition. Merging is the process of creating a flattened TDT representation from a hierarchical TDT representation. Decomposition is the process of breaking a flattened TDT into TDTs organized in a calling hierarchy.

Transformations on a single TDT are either used for the purpose of reducing the size of TDTs or make other TDT transformations possible. We have presented transformations targeting at size reduction in [5]. Here we show one transformation which is used in merging.

Serialization specified in the dependency table may be explicitly included in the control table by introducing a state variable in the control table. In the sequel, we first show a TDT with a dependency table, followed by an equivalent TDT with serialization information included in the control table.

Example 2.4. Consider the following TDT representation:

TDT_4:

a_1	a_2	a_3
		m
s		
	s	

c_1	1	0	0
c_2	-	1	0
a_1	1	1	
a_2	1	1	
a_3	1		0

The 'm' in the dependency table indicates that action a_3 modifies condition c_2. After introducing a state variable S to explicitly serialize a_1 and a_2 in the control table , we get the following TDT:

TDT_{4r}:

a_1	a_2	a_3
		m
	s	

S	0	1	0	1	-
c_1	1	1	0	0	0
c_2	-	-	1	1	0
a_1	1		1		
a_2		1		1	
a_3		1			1
Sn	1	2	1	2	2

The new TDT is a process TDT, which starts execution from $S == 0$ and stops when $S == 2$. Actions a_2 and a_3 can also be serialized in a similar fashion. □

2.3 TDT Merging

Merging is the process of creating a big flattened TDT representation from a hierarchical TDT representation. Merging involving only procedure TDTs results in a procedure TDT. Merging involving a process TDT always results in a process TDT.

2.3.1 Three Basic Merging Cases

Merging involving only procedure TDTs may be classified into three types: (1) merging TDTs in a hierarchy, (2) merging TDTs in a sequence, and (3) merging TDT with a following or preceding action set. Below we show one example for each merging case.

Base Case 1: merging TDTs in a hierarchy. This corresponds to merging TDTs converted from behavioral HDL code with subprogram calls or nested conditional branches.

Example 2.5. Given the hierarchical TDT representation in Example 2.2. which consists of TDT_1 calling TDT_2 as an action set. They can be merged into one table as follows.

c1	1	1	1	0
c2	1	0	0	-
c3	-	1	0	-
a_3	1			
a_4		1		
a_5			1	
a_2				1

□

Base Case 2: merging TDTs in a sequence. Merging in this case is valid when a concurrency type of parallel is specified, or when a type of data-parallel is specified and no action in a preceding TDT modifies conditions of following TDTs. The dependency tables need to be modified accordingly when where is data dependency between actions in different TDTs.

Example 2.6. Given a TDT sequence $\{TDT_5;TDT_6\}$ with details of the two TDTs shown below.

TDT_5:

c_1	0	0	1
c_2	1	0	-
a_1	1		

TDT_6:

c_1	0	0	1
c_2	1	0	-
A		a_2	a_3

These two TDTs can then be merged into TDT_m where

TDT_m =

c_1	1	0	0
c_2	-	0	1
a_1	1		
a_2		1	
a_3			1

□

Base Case 3: merging a TDT with an action set. This transformation is invalid when one action modifies the condition in a following TDT.

Example 2.7. We assume a_3 follows TDT_1 shown in Example 2.1. in an action set of concurrency type serial. Action a_3 and TDT_1 can be merged as shown in the following.

a_1	a_2	a_3
s	s	

C	1	0
a_1	1	0
a_2	0	1
a_3	1	1

Note that to preserve the specified behavior, we have to modify the dependency table. The symbol 's' in column a_1 indicates that if both a_1 and a_3 are selected for execution, they are run in a sequential order where a_3 is alway run after a_1 finishes. □

2.3.2 Merging Involving Data-Dependent Conditions

To handle a data-dependent condition, we introduce a state variable in TDTs to serialize the modification and use of the condition. This may be better explained using one example.

Example 2.8. Given a HardwareC code fragment:

```
c = e > f; if ( c ) a2 ; else a3;
```

The if statement can be converted into a TDT with one condition c. We call this table TDT_8:

TDT_8:

c	1	0
A	a_1	a_2

We also mark "`c = e > f;` " as action a_1. Since a_1 modifies the condition in TDT_4, they can not be merged following merging case 3. Instead, we introduce a state variable to serialize them in a resulting TDT:

S	*ini*	*br*	*br*
c	-	1	0
a_1	1		
a_2		1	
a_3			1
Sn	*br*	*ex_r*	*ex_br*

Here the symbols *ini*, *br*, and *ex_br* are symbolic state values. The execution starts with S === *ini*. It stops when S == *ex_br*. We may assign arbitrary numerical value to these states as in Example 2.4. Note that an additional action for computing next sate has been added in each column. □

2.3.3 Merging Involving Process TDT

When the calling TDT is a process TDT, we first proceed in the same way as if the calling TDT is a procedure TDT. Then, we mark the result as a procedure TDT. When a called TDT is a process TDT, it must result from a conditional loop in the behavioral TDT. We represent it using TDT with a state variable before proceeding as if it is a procedure TDT.

Example 2.9. Consider an action set with two actions including a process TDT which corresponds to a while loop. Suppose it comes from the following HardwareC code:

```
[ a1; while(c) a2; ]
```

The sequence may be represented in the following TDT with an additional state variable.

S	*ini*	*lp*	*lp*
c	-	1	0
a1	1		
a2		1	
Sn	*lp*	*lp*	*ex_lp*

□

2.4 TDT Decomposition

TDT decomposition is the process of replacing a flattened TDT with a hierarchical TDT that represents an equivalent behavior. It is the reverse process of merging. The decomposition process must preserve data dependency, concurrency type, serialization relation among actions. As a result, the decomposition process involves significant modification and use of the dependency table in addition to changing the control table.

Example 2.10. Consider TDT_{4r} in Example 2.4. It may be represented as a hierarchical TDT as follows:

TDT_s	a_3
	m
s	

c_1	1	0	0
c_2	-	1	0
TDT_s	1	1	
a_3	1		1

where TDT_s is a sub-TDT containing two serialized actions a_1 and a_2:

TDT_s:

a_1	a_2
s	

S	0	1
a_1	1	0
a_2	0	1
Sn	1	2

Note that the dependency table has been split accordingly as well. □

In [6] we have presented an algorithm to automatically decompose a flattened TDT. This algorithm uses a two-level algebraic representation of the control table. It consists four major steps:

1. Construct the two-level algebraic representations of the control table.
2. Extract the algebraic kernels of the above expression.
3. Select kernels that lead to valid transformations using specification in the dependency table. (these kernels are called *compatible primary kernels* [6])
4. Construct the hierarchical TDT representation using original TDT and the compatible primary kernels.

Example 2.11. The two-level algebraic expression of control table TDT_{src} is

$$\overline{S}c_1a_1 + Sc_1a_2a_3 + \overline{S}\overline{c}_1c_2a_1 + S\overline{c}_1c_2a_2 + \overline{c}_1\overline{c}_2a_3$$

If we introduce one more state bit S_1 to include in control table the serialization between a_2 and a_3, we get

$$\overline{SS}_1c_1a_1 + S\overline{S}_1c_1a_2 + SS_1a_3 + \overline{S}\overline{c}_1c_2a_1 + S\overline{c}_1c_2a_2 + \overline{c}_1\overline{c}_2a_3$$

A primary kernel which has two co-kernels and does not appear to be a kernel of another such kernel is $\overline{S}a_1 + Sa_2$. This kernel is essentialy two actions a_1 and a_2 executing in a sequence. They may be factored out as a procedure if this leads to resource reduction in synthesis. □

The decomposition algorithm is driven by cost functions related to maximizing the number of times a subtable is called, and to minimizing the size of the subtable. Decomposition is done in an iterative manner with merging to construct alternative hierarchical structure.

3 HDL Code Restructuring

The merging and decomposition algorithms have been implemented in a program called PUMPKIN. We use TDT merging and TDT decomposition in HDL code restructuring. In this section we give one example to show how restructuring is done.

Example 3.1. Consider the following code fragment of a HardwareC version of the i8251 UART design. It contains the receiver synchronous component of the i8251 UART, written as a HardwareC process `rcvr_sync` which calls a procedure named `hunt_mode`. The process `rcvr_sync` reads data from a serial line, packs it up according to a control command from the main control process called `i8251`, and sends it to process `i8251`. Procedure `hunt_mode` finds the synchronization point in time for process `rcvr_sync` before `rcvr_sync` starts packing and sending data.

```
procedure hunt_mode( rxd, drdy, sync1, sync2, mode )
    in port rxd, drdy;
    in port sync1[8], sync2[8], mode[8];
{
    boolean done, data[8], ncount[3];
    done = FALSE;
    while ( ! done ) {
        data = 0xff;
        while ( data != sync1 ) [
k1          wait ( drdy );
k1          data[7:7] = read ( rxd );
k1          data = data >> 1;
            done = TRUE;
        ]
        if ( mode[7:7] == 0 )  [
   k2       ncount[2:2] = 1;
   k2       ncount[0:1] = mode[1:2];
   k2       while ( ncount ) [
k1 k2           wait ( drdy );
k1 k2           data[7:7] = rxd;
k1 k2           data = data >> 1;
   k2           ncount = ncount - 1;
   k2        ]
             done = (data == sync2);
        ]
    }
}
process rcvr_sync(rxd, drdy, valid, mode, control,
        sync1, sync2)
    in port rxd, drdy, valid;
    in port mode[8], control[8], sync1[8], sync2[8];
{
    boolean sync_mode, ncount[3], data[8];
    channel i8251;
    if ( valid ) {
        sync_mode = (mode[6:7] == 0);

        if ( sync_mode ) [
            if ( control[7:7] )
                hunt_mode(rxd, drdy, sync1, sync2, mode );
   k2       ncount[2:2] = 1;
   k2       ncount[0:1] = mode[1:2];
   k2       while ( ncount ) [
k1 k2           wait ( drdy );
k1 k2           data[7:7] = read ( rxd );
k1 k2           data = data >> 1;
   k2           ncount = ncount - 1;
   k2       ]
            send( i8251, data );
        ]
    }
}
```

□

We first convert the HardwareC programs into a hierarchical TDT representation. Then we perform merging to get a flattened TDT representation. Using the algorithms presented in [6], we have found two primary compatible kernels $k1$ and $k2$ as indicated in Example 3.1. above. We pick $k2$ containing a subtraction `ncout -1` operation and a shifting operation `data >> 1` since it leads to bigger potential size reduction. We reconstruct a sub-TDT using this primary compatible kernel. From this sub-TDT, we generate a HardwareC procedure as shown in Figure 1.

Using the co-kernels and the flattened TDT representation, we can construct the main TDT in the

```
procedure sharable(rxd, drdy, data, mode)
in port rxd, drdy; in port mode[8];
out port data[8];
[
    boolean ncout[3];
    ncount[2:2] = 1;
    ncount[0:1] = mode[1:2];
    while ( ncount ) [
        wait ( drdy );
        data[7:7] = rxd;
        data = data >> 1;
        ncount = ncount - 1;
    ]
]
```

Figure 1: The extracted sharable procedure.

resulting hierarchical TDT representation. From this TDT representation, we generate the rest of the process `rcvr_sync` as shown in Figure 2.

4 Discussion and Summary

We have presented a TDT-based method to modify the subprogram hierarchy in behavioral HDL descriptions. This code restructuring is a generalization of code transformation and motion techniques in compilers.

Earlier work on code structuring forms a part of compiler optimizations in high level programming languages [7, 8]. The TDT based code-restructuring is different from conventional compiler optimizations [7] in several ways. First, there are multiple levels of nested concurrent operations in a behavioral HDL description in contrast to a sequential code. While parallelism between operations is defined and used extensively, the correctness of the final result is still defined by the original sequential code (i.e., sequential consistency must be maintained). In HDL code, however, correctness of the transformed code is defined by the partial order and specification of concurrent operations in the original HDL. For this reason, timing semantics of HDL statements in terms of number of cycles taken by an operation and partial ordering of operations must be maintained through the transformations. We capture the timing attributes of operations in the delay table of TDTs and use it to generate concurrent final HDL code. Second, the objective in TDT-based optimizations is very different from that in conventional compilers. Compilers mainly optimize to reduce the execution time of software programs while in HDL code transformations, we are targeting at reducing sensitivity of synthesis results to HDL coding styles, increasing resource sharing, as well as reducing the schedule length when possible.

We are conducting experiments on benchmark designs to identify exactly what hierarchical structure is

```
process rcvr_sync(rxd, drdy, valid, mode, control,
        status, sync1, sync2)
  in port rxd, drdy, valid;
  in port mode[8], control[8], data[8];
  in port sync1[8], sync2[8];
{
  boolean sync_mode;
  channel i8251;
  if ( valid ) {
    sync_mode = (mode[6:7] == 0);

    if ( sync_mode ) [
      if ( control[7:7] ) {
        boolean done;
        done = FALSE;
        while ( ! done ) {
          data = 0xff;
          while ( data != sync1 ) [
            wait ( drdy );
            data[7:7] = read ( rxd );
            data = data >> 1;
            done = TRUE;
          ]
          if ( mode[7:7] == 0 )
            sharable(rxd, drdy, data, mode);
          done = (data == sync2);
        }
      }
      sharable(rxd, drdy, data, mode);
      send( i8251, data );
    ]
  }
}
```

Figure 2: The rest of the `rcvr_sync` process which calls procedure `sharable` twice.

needed that leads to most efficient synthesis. Results from the experiments will be presented at the workshop.

References

[1] R. K. Gupta and S. Y. Liao, "Using a programming language for digital system design," *IEEE Design and Test of Computers*, pp. 72–80, Apr.-Jun. 1997.

[2] D. Ku, *HardwareC – A Language for Hardware Design Version 2.0.*

[3] J. R. Metzner and B. H. Barnes, *Decision Table Languages and Systems.* Academic Press, 1977.

[4] P. J. H. King, "Decision tables," *The Computer Journal*, vol. 10, no. 2, August 1967.

[5] J. Li and R. K. Gupta, "HDL Optimization Using Timed Decision Tables," in *Proc. DAC*, 1996.

[6] J. Li and R. K. Gupta, "Decomposition of Timed Decision Tables and its Use in Presynthesis Optimizations," in *Proc. ICCAD*, 1997.

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

[8] B. S. Baker, "An algorithm for structuring flowgraphs," *Journal of the ACM*, vol. 24, no. 1, pp. 98–120, January 1977.