Reducing Coding Style Effects in High-Level Specifications

Claudionor José Nunes Coelho Jr.
Antônio Otávio Fernandes

Computer Science Department
Universidade Federal de Minas Gerais
Belo Horizonte, MG 30380-260
Brazil

Giovanni De Micheli

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
USA

Abstract

We present a technique to reduce the dependence of an implementation with respect to the coding style of the specification. This dependence in conventional high-level synthesis tools may cause unpredictable implementation results, even for small code changes.

The dependence reduction is achieved by revisiting the control-flow/dataflow model that is used for synthesis. We incorporate register variables and exception handling in the control-flow, which is later translated into a control-unit implementation as a finite-state machine. Due to the additional complexity in the control-flow with exception handling and register variables, the control-unit generation process is implemented using symbolic techniques with Binary Decision Diagrams.

1 Introduction

Synthesis tools at higher levels of abstraction are a reality today. Several tools [6, 3, 22, 13, 18] have been announced in the past years to perform high-level synthesis for systems described at an algorithmic level. Despite their increased use, these tools have a large variance of the control-unit and datapath implementations with respect to the coding style. This variance is the result of the strict line drawn between what is considered to be a control-flow and what is considered to be a dataflow.

In this paper, we characterize the limits between control-flow and dataflow with respect to common specification styles in order to reduce the variance of the implementation. This is achieved by a careful analysis and partitioning of the control-data flow graph of the original specification.

We consider specifications written in a hardware description languages supporting sequential, alternative, concurrent, repetition and exception handling behaviors [8]. Today, most of the hardware description languages support these behaviors, including VHDL [15], Verilog HDL [20], HardwareC [14], StateCharts [7] and Esterel [1]. Because the technique presented here will have a large impact on control-dominated specifications [12], we will focus on these types of specifications, although our technique could be equally applied to dataflow intensive specifications.

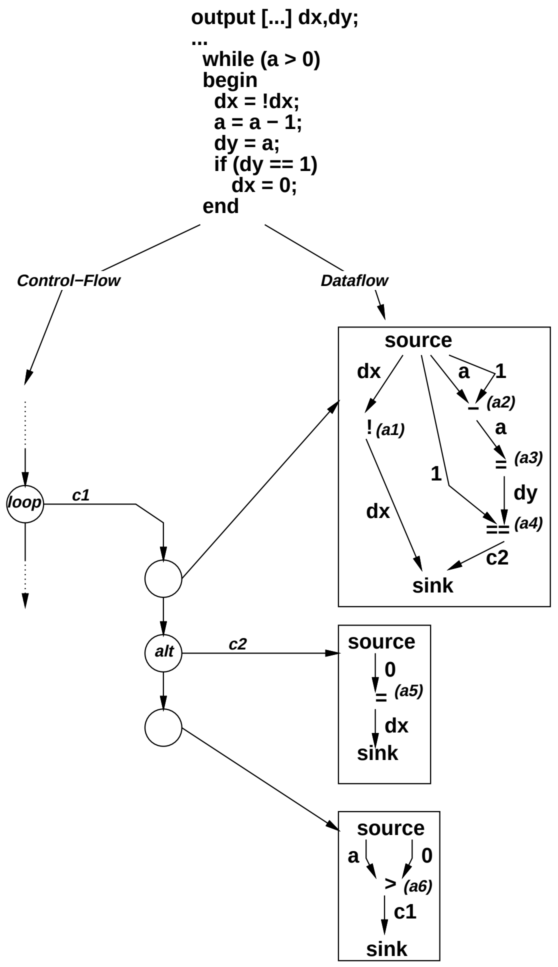
This paper is outlined as follows. In the next section, we present a review of the control-dataflow model that conventional synthesis tools use to represent design specifications. Then, we present the technique of extending control-flow by incorporating register variables and their values into the control-flow. This technique is useful only if a powerful control-unit synthesis tool is able to handle the increased complexity of the control-flow, which is presented next. Finally, we present some results and concluding remarks.

2 Modeling Synchronous Systems from Hardware Description Languages

We focus in this section on a model for control-dominated descriptions. Several models for specifications have been proposed in the past that separate the behaviors in terms of their control-flows and dataflows.

We refer the reader to [4, 16, 11] for an introduction to these models. In this paper, we consider a generic model of the system in terms of these control-flow and dataflow components. We assume the specification contains a single hierarchical control-flow component, but it may contain multiple dataflow components, each corresponding to a different block of operations in the specification.

In the following example, we show how a description language such as Verilog HDL can be modeled in terms of its control-flow and dataflow components.



```
output [...] dx,dy;
...
 while (a > 0)
 begin
   dx = !dx;
   a = a - 1;
   dy = a;
   if (dy == 1)
      dx = 0;
 end
```

Figure 1: *Partitioning of specification into control-flow/dataflow*

Example 1 In Figure 1, we show the representation of a specification in terms of its control-flow and dataflow graphs.

The vertices *loop* and *alt* in the control-flow graph represent iterative and alternative behavior, respectively.

We labeled each operation in the dataflows by events $a_1, \ldots, a_6$. Such events are generated by the control-flow and determine when the corresponding operations will execute. Event a_1, for example, triggers the execution of the negation of dx. These events determine the dependency of the dataflow with respect to the control-flow. Each dataflow also contains two vertices, *source* and *sink* that do not correspond to any operation in the specification. They mark the beginning and end of execution of the dataflow, respectively.

The dataflow of Figure 1 generates input events c_1 and c_2 that trigger the execution of the loop and the execution of the alternative path, respectively. These events determine the dependency of the control-flow in terms of the dataflow.

The reader should note that the control-flow does not make any assumptions on the possible values of its input events over time. In this example, we assume that entering the loop (when event c_1 is generated) and exiting the loop are equally probable, for example. □

As pointed out in the previous example, the control-flow does not make any assumptions on the possible values of variables, nor does the dataflow make any assumption on the control-flow. Since control-flow and dataflow follow different paths in a synthesis tool, the higher the interaction between these two models, the more redundancy in the final implementation. This interaction is affected by coding styles, which is usually presented in commercial tools as application notes or guidelines for "good" synthesis tool usage. In the next section, we present a method that reduces the relationship between coding style and implementation results by moving register variables and their values to the control-flow, and by symbolically traversing the resulting finite-state machine when we generate the control-unit implementation.

3 Control-Flow Register Variables

In order to evaluate the importance of adding register variables to the control-flow of a specification, let us consider Figure 2. If we adopt the conventional control-flow/dataflow partitioning paradigm, variable *state* is placed into the dataflow. Note, however, that this variable is not connected with any other part of the dataflow, yet it triggers the execution of some parts of a control-flow expression. This means that if we move variable *state* into the control-flow, the communication between the control-flow and dataflow will be reduced. This has some advantages from a synthesis perspective. First, since the *state* variable is now incorporated into the control-flow, the redundancy of control in the dataflow can be eliminated, thus reducing the size of the final implementation. Second, when imposing constraints to the design, we will have a more accurate execution model for the control-flow, which will be more independent on the dataflow abstraction.

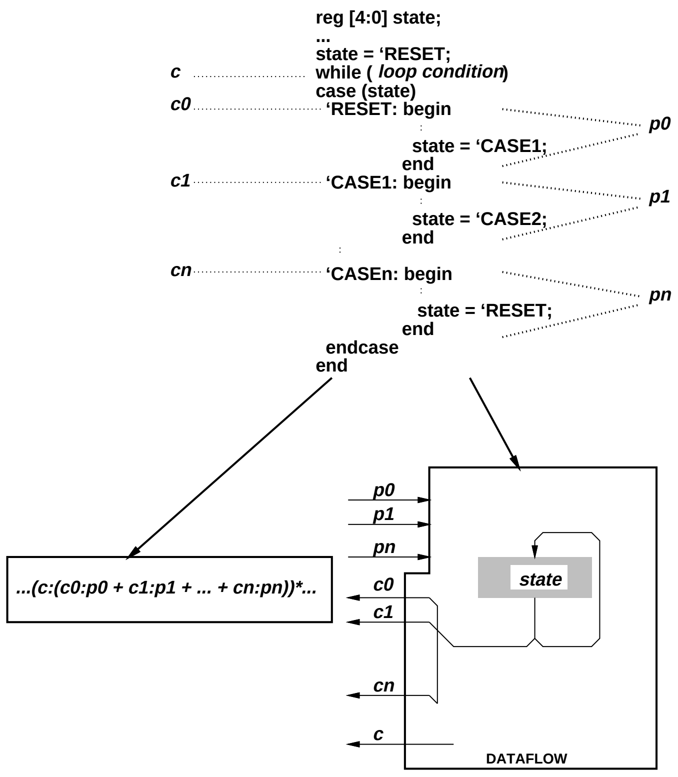


Figure 2: *Program-State Machine Specification*

Control-flow/dataflow transformations have been regarded in the past as useful transformations [5, 19, 4]. However, only *ad hoc* methods were presented, and it was claimed that these transformations would probably increase the number of states of the control.

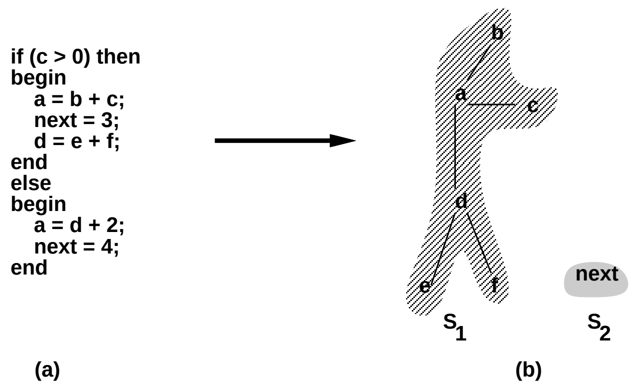
We will first define a reduced dependency graph below, whose structure will allow us to determine which variables should be moved to the control-flow.

Let $\mathcal{D}_f$ be the set of dataflows of a specification.

Definition 3.1 *A reduced dependency graph is the undirected graph $G_r = (V_r, E_r)$, where V_r is the set of non-constant variables, and an edge between two variables u and v exists if u depends on v or if v depends on u in at least one of the dataflows of $\mathcal{D}_f$.*

In this definition, a reduced dependency graph collapses all the dependencies occurring in the different dataflow graphs, thus disregarding the dependency of the dataflows with respect to the control-flow. Recall that a variable in a dataflow graph can generate events to the control-flow; thus, the reduced dependency graph can be easily annotated with the variables that are used to generate events to the control-flow.

Because of the nature of specifications in programming languages, not all of the vertices in a reduced dependency graph will be connected, i.e., in general, there will be some variables u and v for which no path will exist between u and v. Let $S = \{S_1, \dots, S_n\}$ be a partition of the set of vertices V_r such that vertices u and v belong to the same partition if they are connected in G_r.



```
if (c > 0) then
begin
   a = b + c;
   next = 3;
   d = e + f;
end
else
begin
   a = d + 2;
   next = 4;
end
```

Figure 3: *(a) Specification and (b) Reduced dependency graph*

Example 2 In Figure 3 we present a specification and its reduced dependency graph. The dataflow blocks corresponding to the then and else clauses of the *if* partitions the variables of the specification into two sets, $S_1 = \{a, b, c, d, e, f\}$ and $S_2 = \{next\}$. Note that if we considered the *then* clause of the *if* construct alone, variables $\{a, b, c\}$ would be disconnected from variables $\{d, e, f\}$, because the edge between variables a and d can be obtained only in the dataflow of the else clause. □

What happens when one of the blocks S_i of a partition S is connected to the control-flow, but not connected to remaining part of the dataflow? If this block of variables were moved to the control-flow, the number of edges crossing the control-flow and dataflow boundaries (given by the relationships between operations and possible control flows in the specification) would be reduced, thus giving a better dataflow/control-flow partitioning. By reducing this interaction between dataflow and control-flow, we would make the system more predictable.

Although in theory we could move all of the dataflow into the control-flow, or vice-versa, in practice this becomes infeasible for two reasons. First, the techniques for analyzing and synthesizing dataflows and control-flows are different, and as a result, optimization techniques would be applied in the wrong places. Second, indiscriminately making everything a control-flow may potentially cause an exponential blow-up in the number of states. Thus, any move from dataflow to control-flow and vice-versa must be performed with caution. For a limited set of operations which uses constant operands, variables can be moved into the control-flow without a large penalty to the complexity of the control-flow. We call such variables control-flow variables, and their corresponding variable blocks (S_i) control-flow blocks.

Let S be a partition on the vertices of G_r, a reduced dependency graph, and let S_i be a block of S such that no vertex $v \in S_i$ corresponds to an I/O port of the specification. Then, we can say that S_i is useless or it is a control-flow block.

The basic idea relies on the fact that S_i is disconnected from the remaining part of the control-flow. Thus, if S_i is not connected to the control-flow, it will be useless, since all the values assigned to its variables will not be used anywhere. On the other hand, if this block of variables is connected to the control-flow, then it will be a control-flow block.

In the sequel we denote by $\sigma = \{v, c_1, \ldots, c_m\}$ a generic connected component of V_r when c_i are Boolean variables. We also denote by $\mathbf{R} = \{=, \neq, <, >, \leq, \geq\}$ the set of relational operations and by ρ a generic element of $\mathbf{R}$. We also denote by γ a constant. The following corollary is used in our extension to control-flow expressions.

Corollary 3.1 *Let v and c be variables of the connected component σ. Let also f be either an identity function, an increment or a decrement, and let $c_j \leftarrow \rho(v_1, \ldots, v_n)$ and $v \leftarrow f(v)$ be the only operations of the specification defined over c_j and v. Then, either S_i is a control-flow block or it is useless.*

It remains to be seen that such transformations are useful by showing that these types of specifications occur in real designs. It is not hard to see that the variable *state* from Figure 2 satisfies the conditions of Corollary 3.1. We present in Figure 4 (a) the different dataflows for the description of Figure 2, and in Figure 4 (b) the reduced dependency graph for these dataflows. Other variables that often occur in the specifications of control-dominated specifications are counters, for example.

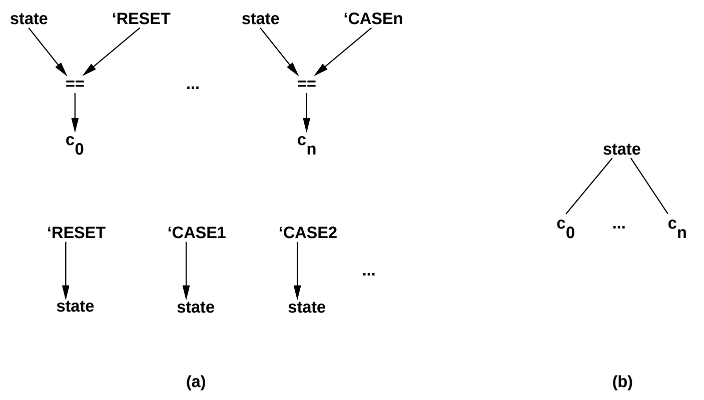


Figure 4: *(a) Dataflow graphs for program-state machine and (b) reduced dependency graph*

The observations shown in this section lead to the definition of register variables and register operations into the control-flow. Note that we only incorporated assignment to constants, increments and decrements. We could have incorporated more of the dataflow operations into the control-flow. However, introducing more variables into the control-flow could easily increase complexity in the internal representation of the control-flow.

Note also that every register variable $v \in V_r$ is finite, since the corresponding specification has finite memory in the number of variables. As a result, every operation performed on the register variable v will be computed over the range $\{0, \ldots, |v| - 1\}$, where $|v|$ is the number of possible values for the register.

4 Building Finite-State Machines from the Extended Control-Flow

Exception handling and register variables make the control-flow much harder to be analyzed and synthesized. In order to cope with this additional complexity, we have developed a Binary Decision Diagram [2] based tool that encapsulates all the design decisions and design choices from the control-flow and scheduling information from the dataflow. Due to the lack of space, we will only present the procedure used to obtain the finite-state machine from the control-flow representation. This finite-state machine is represented symbolically as a transition relation [21]. The reader should refer to [10] for a throughout explanation of the subject.

```
(out,states,disable) cfe2cffsm(in,kill,p) {
  kill = kill ∨ p.disable();
  switch (p.type()) {
  case OPERATION:
     out = create_new_register();
     out.guard() = in ∧ kill';
     $p.operation()$ = $p.operation()$ ∨ out;
     return (out,out);
  case DISABLE:
     p.block().disable() = p.block().disable() ∨ in;
     return (0,0);
  case SEQUENTIAL:
     states = empty_set();
     foreach (pi sequential) {
       (out[i],states[i]) = cfe2cffsm(in,kill,pi);
       in = out[i] ∨ pi.disable();
       states.union(states[i]);
     }
     return (out[n] ∨ p.disable(),states);
  case PARALLEL:
     states = empty_set();
     or = false;
     foreach (pi parallel) {
       (out[i],states[i]) = cfe2cffsm(in,kill,pi);
       out[i] = wait_all_branches(p,out[i]);
       out = out ∧ out[i];
       states.union(states[i]);
     }
     return (out,states);
  case ALTERNATIVE:
     states = empty_set();
     or = false;
     foreach (case ci:  pi in an alternative construct) {
       (out[i],states[i]) = cfe2cffsm(in ∧ ci,kill,pi);
       out = out ∨ out[i] ∨ pi.disable();
       states.union(states[i]);
     }
     return (out,states);
  case INFINITE:
     net = create_new_net(in);
     (out,states) = cfe2cffsm(net,kill,pi);
     net = net ∨ out;
     return (0,states);
  case LOOP:
     net = create_new_net(in ∧ ci);
     (out,states) = cfe2cffsm(net,kill,pi);
     net = net ∨ out ∧ ci;
     return ((in ∨ out) ∧ ci',states);
  case BASIC BLOCK:
     states = empty_set();
     for(i=1; i≤ALAP; i++) {
       out = create_new_register();
       out.guard = in ∧ kill' ∧$F_i$;
       in = out;
       states.union(out);
     }
     return (out,states);
  }
}
```

Figure 5: *Procedure for the translation of a Control-Flow into a Finite-State Machine*

Upon receiving the execution guards `in` and `kill`, procedure `cfe2cffsm` depicted in Figure 5 signals the execution for a control-flow block `p`, by computing symbolically the FSM for `p`, and returning the condition (`out`) representing when the FSM exits the execution of `p` and a collection of the states created in block `p`. We assume that each block contains a disable [1] mechanism (`p.disable()`) that collects all the guards of a disable command to that block. The mechanism `p.disable()` tells us when the block is forced to exit due to an exception handling condition.

In basic blocks, we generate a state for each possible execution time of the basic block, given by its as soon as possible and as late as possible schedules. Then, for each operation i that can be scheduled in the basic block, we create a Boolean decision variable x_{ij} whose value will be determined during synthesis. Whenever x_{ij} is 1, action i is scheduled to be executed in state j. We assume here single cycle actions, although this procedure could be easily extended to include multi-cycle actions [11]. A transition from a state j to state $j+1$ in the FSM of a basic block can occur only if some action can be scheduled after j, which is captured by $F_j = \bigvee_{k>j} x_{ik}$.

[1]the disable mechanism is used here to denote an exception handling mechanism like the exception mechanism used in Verilog HDL or StateCharts

5 Experimental Results

In this section, we present how efficient this technique is in handling different styles of coding along by allowing exception handling mechanisms and register variables.

Unless otherwise stated, all execution times reported in this section will be for an Silicon Graphics INDY 4400 at 200 Mhz with 64 Mbytes of RAM.

5.1 Protocol Synthesis in an Ethernet Coprocessor

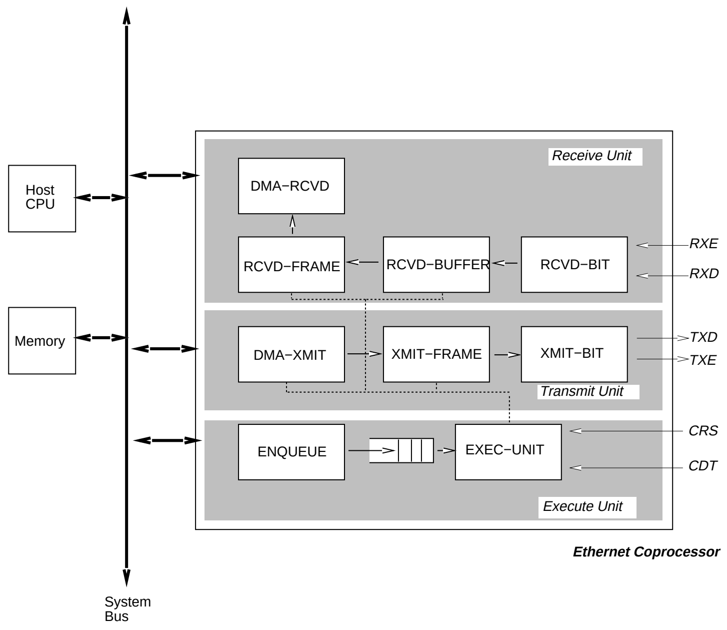


Figure 6: *Ethernet controller block diagram*

The block diagram of an ethernet coprocessor is shown in Figure 6. This coprocessor contains three units: an execution unit, a reception unit and a transmission unit. These three units are modeled by thirteen concurrent processes. We will present here how two different code styles of the process *xmit_frame* lead to the same control-unit implementation.

Process *xmit_frame* interacts with two other processes, *dma_xmit* and *xmit_bit*. It was specified as a program state machine written in Verilog HDL, as shown in Figure 7 [9].

This process works as follows. Upon receiving a byte from process *xmit_frame*, *xmit_bit* sends the corresponding bit streams over the line `TXD`. Thus, *xmit_bit* must receive each byte eight cycles apart, which constraints the rate at which the bytes are transmitted from *xmit_frame*.

As most protocols behave, there is a few number of states in which exception handling mechanisms must be ensured. As a result, we can also specify process *xmit_frame* with an exception handling mechanism, i.e., the *disable* command of Verilog HDL. In the former implementation, because we have to abort the transmission of a frame if `CCT` becomes true, we implemented the program state machine with a *while* loop which pools signal `CCT`, and a *case* statement on variable *state*, which determines the next state of the program state machine to be executed. Note that this state variable is not part of dataflow and it should be incorporated into the control-unit for *xmit_frame*.

Since the execution of the *while* loop should be aborted if `CCT` becomes true, we re-implemented the specification for the program state machine of process *xmit_frame*. In this new specification, we execute a sequential code that traverse the different program states of Figure 7, and we execute a watchdog in parallel with the new sequential code. If condition `CCT` becomes true, then the watchdog will disable the execution

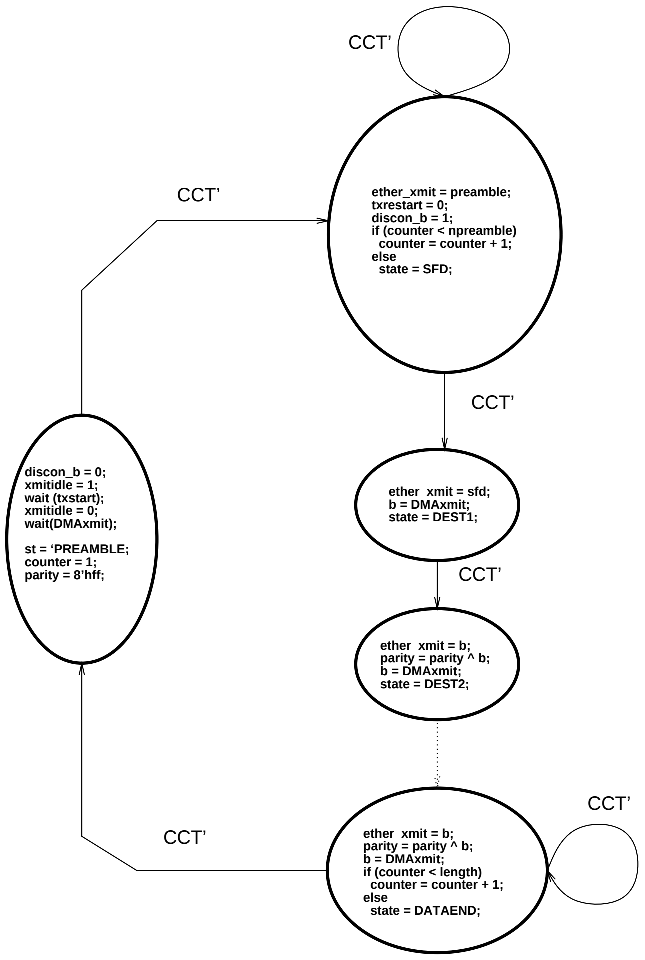


Figure 7: *Program state machine for process xmit_frame*

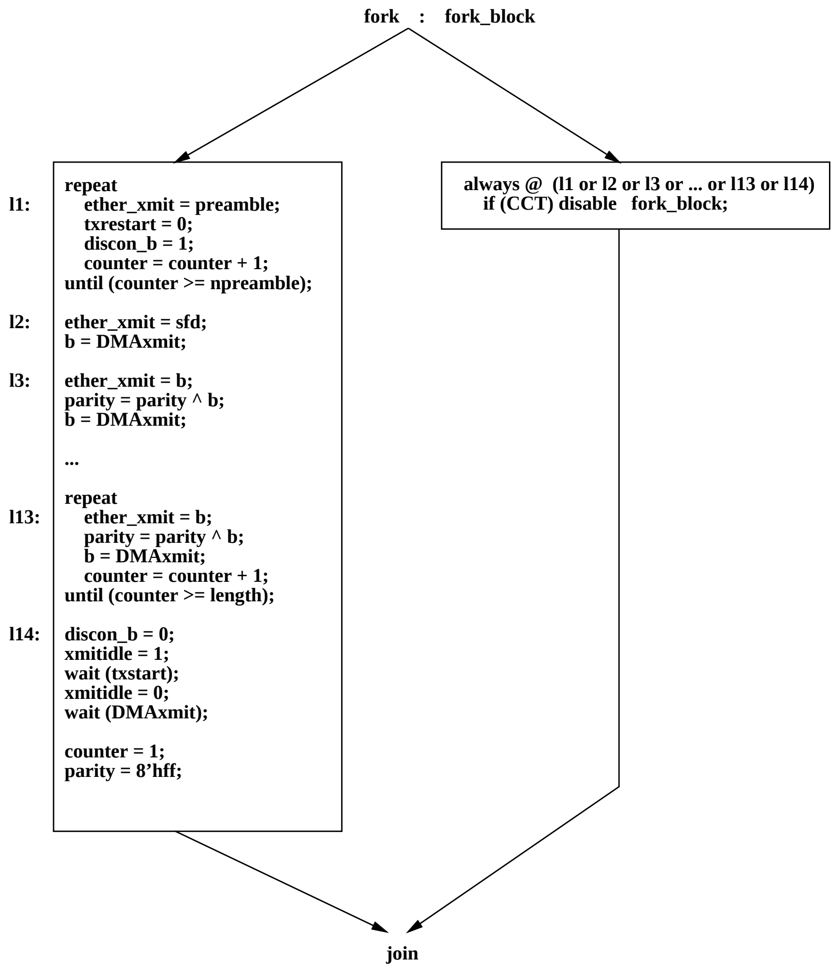
of the concurrent block containing both the sequential code and the watchdog. A graphical representation of the specification can be seen in Figure 8.

	# States		Trans. Rel.	Time
xmit-frame (except.)	178	90	3022	4.3s
xmit-frame	178	90	24439	32.21s

Table 1: *Results for the control-unit generation of xmit_frame*

Table 1 presents the results for the generation of the control-unit for *xmit_frame* from its control-flow model. The first column shows the number of states of *xmit-frame* before scheduling the operations. The second column shows the number of states after state minimization. The third column shows the size of the transition relation in terms of BDD nodes. The fourth column shows the execution time taken to synthesize the control-unit. Note that by having a finite-state representation of the behavior of the system in two different specifications, we were able to obtain two comparable implementations with the same number of states.

In Table 1, note the difference between the sizes of the transition relation of both implementations. Although the complexity of the control-flow in the program state machine case is larger than the complexity of



```
      repeat
l1:     ether_xmit = preamble;
        txrestart = 0;
        discon_b = 1;
        counter = counter + 1;
      until (counter >= npreamble);

l2:   ether_xmit = sfd;
      b = DMAxmit;

l3:   ether_xmit = b;
      parity = parity ^ b;
      b = DMAxmit;

      ...

      repeat
l13:    ether_xmit = b;
        parity = parity ^ b;
        b = DMAxmit;
        counter = counter + 1;
      until (counter >= length);

l14:  discon_b = 0;
      xmitidle = 1;
      wait (txstart);
      xmitidle = 0;
      wait (DMAxmit);

      counter = 1;
      parity = 8'hff;
```

```
always @ (l1 or l2 or l3 or ... or l13 or l14)
   if (CCT) disable   fork_block;
```

Figure 8: *Implementation of program state machine with exception handling*

the specification using the disable construct, it would still not account for this large difference. Another reason for this discrepancy is due to the variable ordering chosen for the BDD variables. Binary Decision Diagrams are very sensitive to variable ordering and a bad choice for variable ordering can result in exponentially large BDDs. When computing the transition relation, we placed the conditionals and register variables on the top, and we grouped the Boolean variables belonging to basic blocks together.

	# States		Trans. Rel.	Time
xmit-frame (except.)	178	90	3022	4.35s
xmit-frame	178	90	14149	402.64s

Table 2: *Control-unit generation with dynamic variable ordering*

The reader should recall that the program state machine implementation of *xmit_frame* has a state variable that was incorporated into the control-unit of *xmit_frame*. This variable interacts with all basic blocks representing the states of the program state machine, as it can be seen in Figure 7. As a result, no good variable ordering can be found for this variable with respect to the ordering of variables created for each basic block.

In order to smooth out the effects of a bad variable ordering for the state variable, we ran both specifications on our program with the BDD using dynamic variable ordering [17]. The results are reported in Table 2.

6 Conclusions

We considered in this paper the effect of introducing register variables into the control-flow of a specification in order to smooth out the effects of coding style of the specifications. At the higher levels of abstraction, these effects tend to cause a large variance in the final implementation of control-dominated specifications.

We also presented a technique to obtain a control-unit implementation from the control-flow based on Binary Decision Diagrams. This technique was necessary in order to reduce the increased complexity of dealing with extended control-flow.

As future work, we are currently considering the application of these techniques to reduce the activity of the circuit, and thus reducing the power consumption.

References

[1] G. Berry and G. Gonthier. *The Esterel Synchronous Programming Language: Design, Semantics, Implementation.* Ecole Nationale Supérieure des Mines de Paris and Institut National de Recherche en Informatique et Automatique.

[2] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, pages 293–318, September 1992.

[3] R. Camposano, R. A. Bergamaschi, C. E. Haynes, M. Payer, and S. M. Wu. The ibm high-level synthesis system. In R. Camposano and Wayne Wolf, editors, *High-Level VLSI Synthesis*, pages 79–104. Kluwer Academic Publishers, June 1991.

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

[5] M. Davio, J.-P. Deschamps, and A. Thayse. *Digital Systems with Algorithm Implementation.* John Wiley & Sons, 1983.

[6] G. DeMicheli, D. C. Ku, F. Mailhot, and T. Truong. The olympus synthesis system for digital design. *IEEE Design and Test Magazine*, pages 37–53, October 1990.

[7] D. Drusinsky and D. Harel. Statecharts as an abstract model for digital control-units. Technical Report CS86-12, Weizmann Institute of Science, 1986.

[8] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[9] Benchmarks of the 1992 high-level synthesis workshop.

[10] C. N. Coelho Jr. and G. De Micheli. Modeling and synthesis of synchronous system-level specifications. (to appear in current issues in electronic modeling) 001, UFMG, 1994.

[11] C. N. Coelho Jr. and G. De Micheli. *Analysis and Synthesis of Concurrent Digital Circuits Using Control-Flow Expressions.* PhD thesis, Stanford University, February 1996.

[12] K. Keutzer. Three competing design methodologies for asics: Architectural synthesis, logic synthesis, and module generation. In *Proceedings of the Design Automation Conference*, pages 308–313, June 1989.

[13] D. Knapp, T. Ly, D. MacMillen, and R. Miller. Behavioral synthesis methodology for hdl-based specification and validation. In *Proceedings of the Design Automation Conference*, pages 286–291, June 1995.

[14] D. C. Ku and G. DeMicheli. Hardwarec - a language for hardware design (version 1.0). CSL Technical Report CSL-TR-88-362, Stanford, August 1988.

[15] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1989.

[16] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw Hill, 1994.

[17] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Workshop on Logic Synthesis*, Lake Tahoe, CA, April 1993.

[18] A. Seawright. *Grammar-Based Specification and Synthesis for Synchronous Digital Hardware Design.* PhD thesis, UC Santa Barbara, 1994.

[19] E. Stabler. Microprogram transformations. *IEEE Transactions on Computers*, c-19:908–916, 1970.

[20] D. E. Thomas and P. R. Moorby. *The Verilog hardware description language.* Kluwer Academic Publishers, 1991.

[21] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, Santa Clara, November 1990.

[22] W. Wolf, A. Takach, C. Huang, and R. Manno. The Princeton university behavioral synthesis system. In *Proceedings of the 29^{th} Design Automation Conference*, pages 182–187, June 1992.