Separate Compilation for Synchronous Modules

Jia Zeng Stephen A. Edwards*

Department of Computer Science, Columbia University
New York, New York, USA

Abstract. Synchronous models are useful for designing real-time embedded systems because they provide timing control and deterministic concurrency. However, the semantics of such models usually require an entire system to be compiled at once to analyze the dependencies among modules. The alternative is to write modules that can respond when the values of some of their inputs are unknown, a tedious and error-prone process.
We present a compilation technique that allows a programmer to describe synchronous modules without having to consider undefined inputs. Our algorithm transforms such a description into code that does as much as it can with undefined inputs, allowing modules to be compiled separately and assembled later.
We implemented our technique in a compiler for the Esterel language and present results that show the technique does not impose a substantial overhead.

1 Introduction

The synchronous model of computation [1] has emerged as a successful, practical way to assemble models of concurrent embedded systems because of its deterministic concurrency and its precise control over time. Each process in a synchronous model operates in lock-step with a global clock, and communication between modules is implicitly synchronized to this clock. Provided the processes execute fast enough, processes can precisely control the time (i.e., the clock cycle) when something happens.

In addition to domains including avionics [2] and hardware design [3], the synchronous model has been used for constructing processor simulations [4, 5]. Especially in this latter setting, heterogeneous synchronous models [6], which can assemble and run synchronous components with no knowledge about their contents, is preferable because it allows separate compilation of components (e.g., cache models, branch prediction units) and even allows them to be written in different programming languages.

In the heterogeneous synchronous model [6], a system is assembled from a collection of concurrently-running blocks that communicate through instantaneous "wires" each connected from a single block's output port to one or more input ports on other blocks. That the blocks be able to respond when not all their input wires are defined is the main requirement for being able to run such blocks without knowledge of their contents. Furthermore, a block must be well-behaved when presented with unknown inputs, e.g., if a block decides output o has value v even though input i is undefined, it

* Edwards and his group are supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and from New York State's NYSTAR program.

may not change its mind, e.g., change the output to *w* once *i* becomes defined. But if blocks do obey these rules, such a system can adopt a Ptolemy-like philosophy [7] in which systems can be assembled from black-box components and executed efficiency with precise, deterministic semantics.

Although it is possible to write such well-behaved synchronous blocks in a general-purpose language such as C, it is a tedious and error-prone process. The alternative, which we propose here, is for the programmer to write blocks only taking into account their behavior when all their inputs are applied and have the compiler interpolate the correct behavior of the block when only some of the inputs are applied. While it would be correct to make the blocks strict, i.e., to respond with no information about any output unless all the inputs are defined, but this is not very helpful.

In this paper, we propose an algorithm that does this interpolation on programs written in the synchronous concurrent, imperative language Esterel [8]. Constructs in Esterel only explicitly address the behavior when all inputs are known (i.e., the user cannot control them to respond in a certain way to unknown values), but their semantics are clear when not all inputs are known.

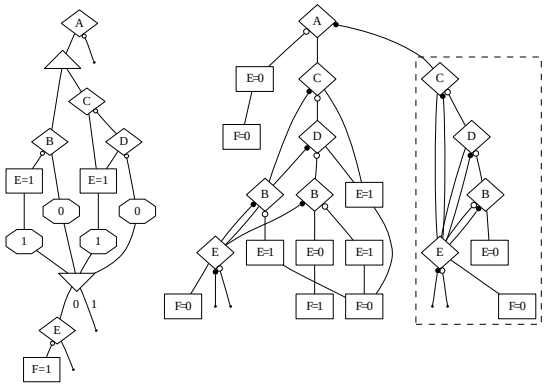
Our work generates code from Esterel that responds to unknown inputs. The enables separate compilation and the assembly of modules written in other languages.

2 Related Work

Digital logic simulators often perform a similar two- to three-valued interpolation. In hardware description languages such as Verilog or VHDL, users often compose systems out of apparently two-valued logic functions such as AND or OR. The simulator, however, interprets them as three-valued functions and performs the simulation in the extended domain. It has long been known, however, that this tends to greatly slow the simulation and attempts have been made to circumvent it where possible (e.g., by detecting when two-valued-only simulation is possible and doing it when possible). Overcoming this speed penalty is a primary goal of our work.

Our intermediate representation bears some resemblance to binary decision diagrams (BDDs—see, e.g., Bryant's survey [9]), but differ enough to make their manipulation very different. Compared to the most common type of BDD, the ROBDD (reduced, ordered BDD), our programs may test variables in different orders and multiple times along a path. Although certain styles of BDDs (e.g., free BDDs) relax this restriction, our formalism is even less like most BDDs because it can communicate within itself, i.e., assign and later test the value of the variable assigned, whereas BDDs typically only make assignment at their leaves. As a result, most BDD algorithms, which are able to assume disciplined variable orderings and a single type of node, are inapplicable for our application. Others, however, have used BDDs to synthesize software [10].

Our algorithm is like a partial evaluation of a three-valued simulator on programs represented as graphs, which resembles many other techniques for generating sequential code from concurrent models [11]. Our algorithm, as a side-effect, orders the nodes under forks and generates a purely sequential program. While this is probably undesirable for certain systems, more clever techniques, such as Zeng et al. [12] could probably be woven into ours to more efficiently generate sequential code.



(a) (b)

Fig. 1. (a) A two-valued GRC. Arcs with bubbles are taken when a variable is 0. (b) Its three-valued projection produced by our algorithm. Arcs with solid bubbles are taken when a variable's value is unknown. Figure 6 shows the construction of the nodes in the dotted region.

3 GRC: Graph Code

We represent the programs we are compiling using a variant of the Graph Code (GRC) format due to Potop-Butucaru [13]. GRC is like a traditional control-flow graph augmented with concurrency and nodes for controlling it. However, loops are prohibited (cross-cycle loops are allowed). The result is a compact, precise way to represent Esterel programs [8], which we compile with our technique, although the same representation could be used for other synchronous, imperative languages.

A GRC program is a rooted directed acyclic graph $G = (N, r, c, V, O, S, t)$ where N is the set of nodes, $r \in N$ is the distinguished root node, $c : N \to (N \cup \{\text{null}\})^*$ is a function that returns the vector of control successors of a node (Successors are ordered, e.g., if $c(n) = (n_1, n_2, n_3)$, then node n can pass control to its first successor n_1, its second successor n_2, or n_3). A null successor represents no successor, used, for example, when there is a *then* branch from a predicate, but no *else* branch.

The finite set V denotes variables. $O \subset V$ are the output variables. $V \setminus O$ are the input variables. S denotes the set of possible states of the program.

Each node has a type given by the function $t : N \to \{$assign-v-to-one, assign-v-to-zero, predicate-on-v, fork, switch, enter, terminate-at-l, sync $\}$. When executed, an assign-v-to-one node sets the variable v to 1 (v is a variable in V). Predicate-on-v tests variable v and sends control to one of its successors; switch is similar but tests program state instead of a variable; enter changes the program state. A fork node sends control to all its successors, which must eventually re-converge at a sync node. All predecessors of a sync must be terminate-at-l nodes, which indicate the exit level of their respective threads. A sync node passes control to the successor whose number corresponds to the highest-numbered terminate node that passed control to it.

The assign-v-to-zero nodes are only added to the graph during our construction. As its name suggests, an assign-v-to-zero node sets the variable v to 0. In two-valued execution, a variable's default value is 0, making such nodes unnecessary. But in the three-valued execution that is the result of our procedure, variables default to the undefined value and therefore require assign-v-to-zero nodes.

Figure 1 depicts such a program graphically. All arcs point downward. The type of each node is indicated by its shape. Assignments are boxes, predicates are diamonds, forks are triangles, terminates are octagons, and syncs are upside-down triangles. The label on a predicate or assignment node indicates the variable tested or set. For predicate nodes, the first (false-valued) arc is indicated with a bubble at its source. The label on a terminate indicates the exit level of the corresponding thread. For sync node, each arc is labeled with a number which matches the exit level. A dashed line denotes a data dependency (as shown in Figure 6a: $6 \rightarrow 10$, $9 \rightarrow 10$, $10 \rightarrow 14$, $15 \rightarrow 16$).

A two-valued execution of a GRC program (which contains no assign-to-zero nodes by definition) starts with an initial program state and an assignment of values to input variables (i.e., either $v = 0$ or $v = 1$ for all $v \in V \setminus O$). Then it derives a subset S of the nodes as follows. S includes the root node; every successor node of each fork, assignment, enter, or terminate node in S; and for every predicate node n in S that refers to variable v, the first (true) successor is in S if v is an input variable with value 1 or the graph includes an assignment-to-one node for v, and the second (false) successor of n otherwise. For a sync node, all of its predecessors' (terminate nodes) exit levels are checked, and S includes the sync's successor under the branch whose label is the same as the highest exit number. The value of each output variable is 1 if the set includes an assignment-v-to-one node to variable v and 0 otherwise.

Consider executing the graph in Figure 1a using the node numbers from Figure 6a and with the assignments A=1, B=1, C=0, and D=1. Node 1 is in S since it is the root, and since A=1, node 2 is also. This adds nodes 3 and 8. Since B=1, node 12 is in S but node 9 and node 11 are not, and since C=0, node 4 is in S, and since D=1, node 6 and 7 are in S but node 5 is not. Since node 7 and node 12 are included, and node 7's exit level (1) is higher than node 12 (0), sync node 13's branch 1 is executed. That excludes node 14 and 15 from S. In the end, $S = \{1,2,3,4,6,7,8,12,13\}$ so E=1 and F=0.

The above procedure requires the value of every input variable to be known when the program starts; we want to relax this. In particular, if we know the values of only certain inputs, we would like to conclude whatever we can about as many outputs as possible provided they are consistent with any future values for the unassigned inputs.

One way to answer this question is to execute the GRC program using three-valued logic, i.e., adding a third value that represents unknown or undefined (we write it $\perp$) to the usual 0s and 1s. This introduces another set of nodes to the simulation procedure: those that might run if additional input is provided later. This is a more complicated procedure that does not reduce to the usual sequential execution behavior of imperative programs, unlike the two-valued simulation of GRC defined above, which can be transformed into sequential code using a fairly inexpensive procedure [12].

4 Our Construction Algorithm

Our main contribution is the algorithm described here that takes a GRC program and constructs a fast sequential program that evaluates the graph in the three-valued domain, i.e., it allows some of the input variables to be undefined. Our algorithm works in four phases (see Figure 2a). Given a GRC program, we add nodes and arcs to represent data dependencies, compute a topological order of this annotated graph, compute informa-

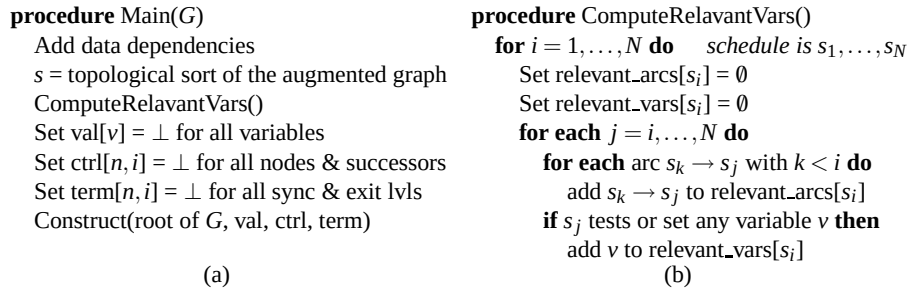


Fig. 2. (a) The Main procedure and (b) ComputeRelevantVars

tion about the subgraph under each node that will tell us what information we can forget during a simulation of the program, and finally construct a sequential program by performing this simulation. We try to keep the size of the generated program under control; we do this by allowing as much reconvergence as possible in the generated code, i.e. by identifying (and reusing) equivalent states during the simulation.

4.1 Adding Data Dependencies

The algorithm starts by adding data dependencies. For each output variable *v*, this process adds an assign-*v*-to-zero node and then adds arcs from each assign-*v*-to-one node to this new node, and arcs from this new node to each predicate-on-*v* node that tests *v*. The result is that there is now a path from each assign-*v*-to-one node for a variable to each node that tests that variable, hence ensuring the topological sort respects data dependencies. Furthermore, it introduces an assign-*v*-to-zero node that will appear in the schedule when it is possible to determine that a particular variable may be zero. Figure 6a shows the effect of applying this procedure on Figure 1a.

4.2 Summarizing Dependency Information

Keeping the size of the generated graph under control is the main trick in our algorithm. Although it would be correct to consider the value of each variable and control arc when considering which subgraphs can be shared during code generation, this would be very inefficient and always produce an exponentially-large tree as a result. Instead, we attempt to model the state of a simulation using as little information as possible because we want to consider a maximum number of states to be identical so code for them can be shared.

Our insight is this: at a particular point in the schedule, we only care about nodes that appear later in the schedule since by definition we must have already executed anything earlier, and only two things matter about them: the variables they test and the state of control arcs that lead from nodes earlier in the schedule to later nodes.

Consider building a subgraph for the nodes starting at 8 in Figure 6a, and assume the node numbers correspond to their position in the schedule. At this point, the simulation will have established values for variables A, C, and D, but we do not directly care about

any of them since code for them has already been generated and we will not test any of them later. However, we do care about whether node 10 will be executed, which can be affected by node 6, and whether node 13 was triggered by its predecessors, since we will be generating code for nodes 10 and 13 (they appear after 8 in the schedule).

As a result, we consider identical any simulation states that differ only on variables A, C, or D. We also consider the control flowing in to nodes 8, 10, and 13.

The ComputeRelavantVars procedure (Figure 2b) builds two sets that exactly capture this notion of which variables and control states we care about during the construction. By stepping through the nodes of the graph in scheduled order, ComputeRelavantVars computes relevant_arcs[s_i], the set of all arcs that go from nodes before s_i in the schedule s to nodes after s_i, and relevant_vars[s_i], the set of all variables that are either tested or set in the nodes after s_i. Note that because s is a topological order, nodes after s_i in the schedule necessarily include the subgraph under s_i.

In Figure 6a, if $s = (1,2,3,4,5,6,7,8,9,10,11)$, ComputeRelavantVars finds relevant_arcs[8] = {2→8, 6→10, 5→13, 7→13}, relevant_vars[6] = $\{B,E,F\}$. Both relevant_vars and relevant_arcs are global and are not modified after ComputeRelavantVars.

4.3 Construct

The Construct procedure (Figure 3) simulates the three-valued behavior of the GRC program and, as a side-effect, constructs our objective: a graph that reproduces its behavior. In addition to the node n that is being synthesized, it takes three arrays: val[v] is the value (0, 1, or $\perp$) of each variable; ctrl$[n,i]$, $i = 0,1,\ldots$ is the state (again, 0, 1, or $\perp$) of each control arc leaving each node; and term$[n,i]$ is the state of each termination level $i = 0...M$ reaching each sync node n (M is the maximum possible exit level reaching n).

Construct begins by checking for an end condition: for the last node in the schedule s, the "node following it" is simply null. It then computes two partial functions (associative arrays): var_state, which contains the value of each relevant variable, i.e., those set or tested by any node that comes after n in the schedule (computed earlier by ComputeRelavantVars); and node_state, which computes the execution state (1=will run, 0=will not run, or $\perp$=might run) of all the relevant nodes, i.e., predecessors of n plus all those with incoming arcs that come before n in the schedule (again, computed earlier by ComputeRelavantVars).

Together, the node itself and the two partial state functions constitute the total state on which the subgraph to be built for n. The procedure then looks to see whether a subgraph with identical state has already been built and returns it if it exists.

Otherwise, the real work starts. First, the node following n in the schedule is identified as m, since it will be recursed on later. The procedure assumes the node n is a flow-through type (e.g., assign-v-to-one or a fork) and sets all its control successors to have the same activation condition as the node itself. These assignments will be modified below when necessary, especially for predicate and switch nodes.

There are two main cases: once the node is known not to run, this information is propagates as far as possible by the PropagateZeros procedure. Nodes that set each such variable to zero are created, assembled into a chain. Finally the subgraph that executes the nodes after n is connected to the end of this chain after a recursive call to Construct.

The other case, when the node might or is known to run (node_state = $\perp$ or 1), is handled quite differently. Dealing with assign-*v*-to-one and enter nodes is simple: if it is known to run, it is simply copied to the new graph. Furthermore, for an assign-*v*-to-one node, the value of *v* is set to 1 so that it will be propagated to later constructions.

Conditional nodes (predicate-on-*v* and switch) are more complicated. To deal with them, the BuildCondition function is called (Figure 4). If the processed node *n* is a predicate-on-*v* and *v*'s value is known, the branches under *n* are set to active and inactive depending on the value.

Otherwise, if the node is a switch or a predicate-on-*v* whose variable *v* is unknown, the algorithm constructs an identical conditional node in the generated program and considers all possibilities: one of the branches—corresponding to a possible condition—is set active, and the others are made inactive (their control state is set to zero). For switch, the possible conditions correspond to each of its successors. For a predicate node, the possible conditions are related to the variable's value, which can be true, false, or unknown when the generated program runs. In the last condition, all branches are set active. For each condition, the variable value is saved appropriately in val array and then Construct is called on the next node in sequence with the new state.

Terminate and sync nodes deal with exit levels and are handled separately. For every sync node, its related threads' exit levels are preserved by the term array. When a terminate-at-*l* node is met at the end of a thread, if it is known to be executed, it sets the term array element of the exit level *l* to be 1 for the corresponding sync; if its control value is $\perp$ and no other thread exited at the same level, the element in the term array is set to $\perp$. The sync node computes the highest possible exit level(s) by looking at the term array, then passes its control value to the corresponding branch. This algorithm simulates the two-valued behavior. BuildSync in Figure 5a simulates sync's behavior.

For all these types, Construct is called on the next node *m* and saves the root of returned subgraph to n''. Switches and predicates are exceptional: they have different new states built to meet all possible conditions, so Construct is called for every condition.

Finally, n' is the new node as the root of the subgraph constructed on *n*. To make it possible to later identify its state, this fact is recorded in BuildNode. n' is returned to the caller, which probably adds an arc leading to it.

We use a few simple helper functions (not shown). Link(*n*,*m*) connects arcs: if *n* is null, it returns *m*; otherwise, a control arc $n \rightarrow m$ is added and *n* is returned. Copy(*n*) creates a new node in the generated program with the same type and variable as node *n*.

4.4 State

The Construct procedure maintains a collection of subgraphs in the generated program, each corresponding to a particular node in the original program and the state that it implicitly assumes the original program was in before reaching the subgraph. Such a state is a triple: $\langle n, \text{var\_state}, \text{node\_state} \rangle$. *n* is the node leading the subgraph constructed, var_state is a partial assignment of values to variables the subgraph cares about, and node_state is an analogous assignment of values to control arcs relevant to the subgraph. Specifically, those that pass into the subgraph from outside: arcs within the subgraph, by definition, will be evaluated as part of the subgraph.

```
1:  function Construct(n, val, ctrl, term)
2:    if n is null then
3:      return null                                          bottom of the program
4:    Clear node_state                                       partial function on nodes
5:    node_state[n] = 1 if n is the root node, ⊥ otherwise
6:    for each arc p --i--> q in relevant_arcs[n] do
7:      node_state[q] = node_state[q] OR ctrl[p,i]
8:    var_state = val                                        partial function on variables
9:    for each v not in relevant_vars[n] do
10:     var_state[v] = DONTCARE
11:   if BuiltNode[⟨n, var_state, node_state⟩] exists then
12:     return BuiltNode[⟨n, var_state, node_state⟩]
13:   m = node following n in s                              on which to recurse
14:   for each successor n_i of n do                         assume flow-through
15:     ctrl[n,n_i] = node_state[n]
16:   if node_state[n] = 0 then                              node known not to run
17:     PropagateZeros(n, node_state, ctrl, val)
18:     Create chain (v_1 = 0) → (v_2 = 0) → ··· → (v_k = 0) for each variable v_i such that
        val[v_i] = 0
19:     Add an arc from (v_k = 0) → Construct(m, val, ctrl,term)
20:     n' = the first node in the chain: "(v_1 = 0)"
21:   else                                                   node_state[n] is ≠ 0
22:     n' = NULL
23:     case n.type of
24:     Assign-v-to-one :
25:       if node_state[n] = 1 and val[v] = ⊥ then
26:         n' = Copy(n)                                     assign-v-to-one that executes
27:         val[v] = 1;
28:     Enter :
29:       if node_state[n] = 1 then
30:         n' = Copy(n)                                     Enter that executes
31:     Terminate-at-l :
32:       c = SyncMap[n]                                     the sync node related with n
33:       term[c][l] = term[c][l] OR node_state[n]
34:     Sync :
35:       BuildSync(n,ctrl,term)
36:     Switch or predicate-on-v :
37:       n' = BuildCondition(n,m,val,ctrl,term)
38:       goto End
39:     n'' = Construct(m, val, ctrl,term)
40:     Link(n',n'')
41:   End: BuiltNode[⟨n,var_state,node_state⟩] = n'
42:   return n'
```

Fig. 3. The Construct Function.

```
1:  function BuildCondition(n, m,val,ctrl,term)
2:    if n is predicate-on-v and val[v] is known then
3:      ctrl[n,val[v]] = node_state[n]                          active branch
4:      ctrl[n, 1−val[v]] = 0                                   inactive branch
5:      n′ = Construct(m, val, ctrl,term)
6:    else                                   switch or predicate with unknown variable
7:      n′ = Copy(n)
8:      for each successor n_i of n do
9:        ctrl[n, i] = node_state[n]                            active branch
10:       for each successor n_j of n other than n_i do
11:         ctrl[n, j] = 0                                      inactive branch
12:         if v is not NULL then                               predicate value is ⊥
13:           val[v] = i
14:       Add an arc n′ → Construct(m, val, ctrl,term)
15:     if n is a predicate then
16:       for each successor n_i of n do
17:         val[v] = ⊥
18:         ctrl[n, i] = node_state[n]                          active branches
19:       Add an arc n′ → Construct(m, val, ctrl,term)
20:   return n′
```

Fig. 4. The BuildCondition Function.

```
function BuildSync(n,ctrl,term)
  unknown_ctrl = false
  findmax = false
  for each i in term[n], max to min do
    if term[n][i] is 0 then
      ctrl[n][i] = 0;
    else
      if findmax is false then
        findmax = true
        if term[n][i] is ⊥ then
          unknown_ctrl = true
      if unknown_ctrl is true then
        ctrl[n][i] = ⊥
      else
        ctrl[n][i] = node_state[n]
      if term[n][i] is 1 then
        break
  return ctrl
```

(a)

```
function PropagateZeros(n, node_state, ctrl, val)
  if n is null then
    return
  node_state′ = node_state
  for each arc t --i--> n do
    node_state′[n] = node_state′[n] OR ctrl[t, i]
  if node_state′[n] is 0 then
    for each successor n_i of n do
      ctrl[n, i] = 0
    if n.type is Assign-v-to-zero then
      val[v] = 0
  m = node following n in s
  PropagateZeros(m, node_state′, ctrl, val)
```

(b)

Fig. 5. (a) The BuildSync Function and (b) the PropagateZeros function

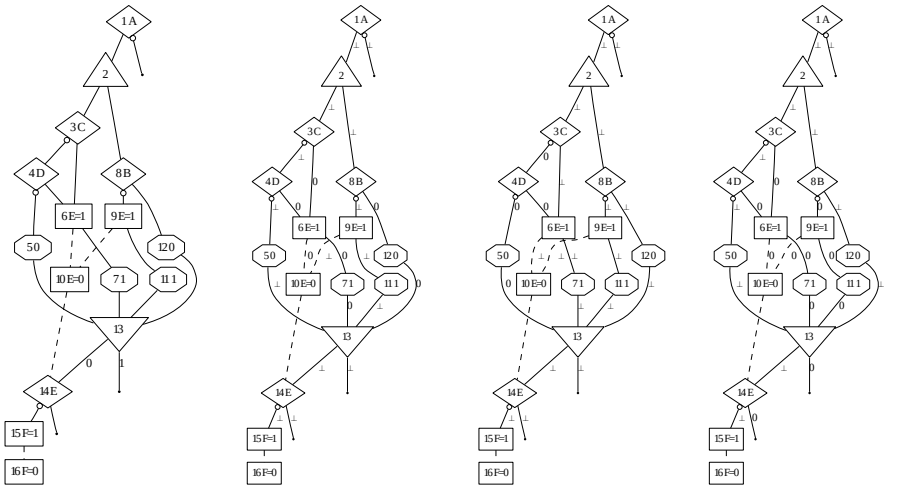


Fig. 6. (a) After adding data dependence nodes and arcs to Figure 1a. (b), (c), (d) Possible simulation states upon reaching node 14. Cases (b) and (c) are equivalent since the relevant variables (E and F) and state of node 14 (the one with incoming arc(s) from outside of the subgraph) are the same. Case (d) is different. Cases (b) and (c) share the node that tests E, whereas case (d) creates the E=0 node in the dashed box in Figure 1b.

4.5 Monotonicity

The code generated by our algorithm is monotonic. When adding data dependencies (Section 4.1), an assign-*v*-to-zero node is linked after all assign-*v*-to-one and before all predicate-on-*v* nodes. This ensures assign-to-one nodes appear first in the topological order, followed by the assign-to-zero node, and finally all predicates that test *v*.

A $v = 0$ assignment is made only when none of the assign-*v*-to-one nodes could or did execute (see Figure 3 line 17-18 and Figure 5b), so the code will never change a variable's value from 1 to 0. It is also impossible for the generated code to change *v*'s value from 0 to 1 because the topological ordering of nodes places assign-to-ones before assign-to-zeros. The *val* array records variables' values throughout the Construct function. So when a predicate-on-*v* node is met (see Figure 3 line 36-38 and Figure 4), val[*v*] is checked first. If *v*'s value is known, the only active branch will be set, and the val[*v*] will not be touched but just passed to later construction (see Figure 4 line 2-5).

4.6 The Example

Figure 6 illustrates some of the algorithm's behavior on Figure 1. A, B, C, and D are input variables; E and F are outputs. Figure 6a was derived from Figure 1 by adding data dependencies. Figure 6b shows the graph after assuming A=⊥, C=0, D=0, and B=0 and arriving at node 14. The label on each arc indicates its value in the ctrl array. Figure 6c is similar, but it assumes A=⊥, C=1 and B=⊥ (predicate-on-D is known not to run in

Example	Lines	Average cycle times		
		Esterel V5	SCFG	3-Valued
comexp	88	1.67s	0.61s	0.80s
iwls3	70	1.04s	0.35s	0.26s
3vsim2	48	0.68s	0.32s	0.46s
multi3	120	1.39s	0.45s	0.47s

Table 1. Experimental Results

this configuration, so D's value is irrelevant). Our algorithm determines that the code generated for these two states is the same and can be shared.

Specifically, at node 14, variables E and F are relevant (and unknown in both Figures 6b and 6c) and the state of node 14 is relevant. In both cases, the state of 14 is $\perp$, which is equal to the ctrl value of incoming arc 13→14.

In these two states, node 10 may still run in the future, so no code is generated to set E to 0, E is therefore also unknown, so it is tested, and F=0 may later be able to run. The code generated for these states is the test of E followed by the assignment of F to 0 in the dashed region of Figure 1b. Paths from the test of C (i.e., when C is 0—Figure 6b) and the test of B (i.e., when B is $\perp$—Figure 6c) converge on this subgraph because the algorithm has identified these states as equivalent.

By contrast, assuming A=$\perp$, C=0, D=0 and B=1 gives the state in Figure 6d. Here it is known that node 10 (assign 0 to E) will run because none of its predecessors will (this is reversed from the usual rule because such nodes are specially designed to detect when a variable is set to 0). This leads to different code of the other two cases, i.e., the assignment of 0 to E attached to the true branch under the test of B in Figure 1b.

5 Experimental Results and Conclusions

We compared the speed of the code generated by our algorithm to that from the Esterel V5 compiler, which translates the Esterel program into a logic circuit and generates code to simulate it, and to the code generated by the algorithm described by Zeng et al. [12], which generates sequential code by adding guard variables. To obtain the average cycle times in Table 1, we ran the generated C code from all three compilers (compiled with gcc `-O3`) for 10 million cycles on a 2.5 GHz Pentium 4 running Linux.

Table 1 shows our results. While the theoretical complexity of our algorithm is exponential, the experiments we ran show it appears to not be an issue in practice.

The code generated by the other two compilers (V5 and SCFG) only perform two-valued computation. Because our compiler adds code for three-valued computation, it generates slower code. However, the experimental results suggest that the slow-down is fairly mild and in some cases, our compiler actually generates faster code. We suspect it is because our compiler uses a different technique to sequentialize the concurrent code.

Together, these experiments suggest that our algorithm is practical, at least for modest-sized programs. There are certainly additional opportunities for optimization. In particular, we intend to integrate this technique with our earlier technique for producing efficient sequential code from (concurrent) program dependence graphs [12].

Although our algorithm was originally designed to generate monotonic three-valued programs from two-valued ones to work with the heterogeneous synchronous model of

computation, it may have other applications. The general idea of partially simulating networks and recording the results as a branching program resembles some approaches for generating efficient simulators for gate-level circuit descriptions [14, 15]. While these approaches insisted on a BDD-like representation, our technique suggests the possibility of selectively "forgetting" inputs, which should give an interesting trade-off between efficiency and code size.

References

1. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91** (2003) 64–83 Invited.
2. Berry, G., Bouali, A., Fornari, X., Ledinot, E., Nassor, E., De Simone, R.: Esterel: A formal method applied to avionic software development. Science of Computer Programming **36** (2000) 5–25
3. Arditi, L., Bouali, A., Boufaied, H., Clave, G., Hadj-Chaib, M., Leblanc, L., de Simone, R.: Using Esterel and formal methods to increase the confidence in the functional validation of a commercial DSP. In: Proceedings of the ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS), Trento, Italy (1999)
4. Vachharajani, M., Vachharajani, N., August, D.I.: The Liberty structural specification language: A high-level modeling language for component reuse. In: Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI). (2004)
5. Penry, D.A., August, D.I.: Optimizations for a simulator construction system supporting reusable components. In: Proceedings of the 40th Design Automation Conference, Anaheim, California (2003) 926–931
6. Edwards, S.A., Lee, E.A.: The semantics and execution of a synchronous block-diagram language. Science of Computer Programming **48** (2003) 21–42
7. Buck, J.T., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: A mixed-paradigm simulation/prototyping platform in C++. In: Proceedings of the C++ At Work Conference, Santa Clara, California (1991)
8. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152
9. Bryant, R.E.: Binary decision diagrams and beyond: Enabling technologies for formal verification. In: Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD), San Jose, California (1995) 236–243
10. Chiodo, M., Giusto, P., Jurecska, A., Lavagno, L., Hsieh, H., Suzuki, K., Sangiovanni-Vincentelli, A., Sentovich, E.: Synthesis of software programs for embedded control applications. In: Proceedings of the 32nd Design Automation Conference, San Francisco, California, Association for Computing Machinery (1995) 587–592
11. Edwards, S.A.: Compiling concurrent languages for sequential processors. ACM Transactions on Design Automation of Electronic Systems **8** (2003) 141–187
12. Zeng, J., Soviani, C., Edwards, S.A.: Generating fast code from concurrent program dependence graphs. In: Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES), Washington, DC (2004) 175–181
13. Potop-Butucaru, D.: Optimizations for faster execution of Esterel programs. In: Proceedings of Memocode, Mont St. Michel, France (2003) 227–236
14. Murgai, R., Hirose, F., Fujita, M.: Logic synthesis for a single large look-up table. In: International Workshop on Logic Synthesis. (1995) 6–11–6–19
15. Ashar, P., Malik, S.: Fast functional simulation using branching programs. In: Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD), San Jose, California (1995) 408–412