Challenges in Synthesizing Fast Control-Dominated Circuits

Cristian Soviani Stephen A. Edwards*
Department of Computer Science
Columbia University, New York

Abstract

Presenting designers with higher-level specification languages is one sure way to improve productivity, but the more abstract the language, the higher the compiler's optimization burden.

We consider generating efficient controller circuits from descriptions written in Esterel. To understand the demands of scalable optimization algorithms, we manually matched the results from sequential synthesis algorithms that produce good circuits but are costly or impossible to run on large designs.

We hoped the high-level structure of Esterel would suggest inexpensive, effective optimizations, but our results are mixed. In the five examples we considered, many optimizations clearly could be automated cheaply, but we needed more global information to match the quality of the existing automatic techniques. This suggests an effective solution would have to combine both local and (potentially costly) global techniques.

1 Introduction

Increasing the productivity of digital designers remains a perpetual challenge. One tried-and-true approach is to provide them with a more abstract hardware description language and let synthesis and optimization handle the time-consuming details. However, despite many valiant attempts at alternatives, the industry remains stuck at the register transfer level (RTL).

We consider the problem of synthesizing algorithms with complex control behavior expressed in high-level languages that are otherwise very time-consuming to specify and verify at the RT level. Thus we are considering a high-level synthesis problem, but not the classical one that has focused primarily on arithmetic-heavy signal processing algorithms (see, e.g., De Micheli [6]). More specifically, we attempt to answer what sort of knowledge is needed about a program written in the Esterel synchronous language [3] for it to be synthesized into an efficient circuit. Unfortunately, for the highest performance it appears necessary to have fairly detailed information about what states the system can actually enter, a global, emergent property that is usually quite costly to compute. This differs from our earlier experience with software synthesis from Esterel [9], in which it turned out that the syntactic structure of the high-level specification provided enough information to greatly speed the generated code compared to that generated directly from the netlist.

*soviani, sedwards@cs.columbia.edu Edwards and his group is supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and from New York State's NYSTAR program. http://www.cs.columbia.edu/~sedwards

The most interesting aspect of Esterel, for the purposes of this study, is its ability to specify concurrently-running, hierarchical FSMs. This can lead to very complex sequential behavior (a consequence of the way concurrency can lead to state explosion), and can be very succinct. Unfortunately, it is also easy to express designs for which the usual (syntax-directed) translation produces inefficient circuits. Although partially the responsibility of the designer, it would be onerous to expect the designer to always choose optimal idioms.

When we began this work, we conceived of it as a state encoding problem, i.e., by manually selecting a good state encoding, could we emulate the effects of aggressive sequential synthesis (i.e., calculating reachable states, using them to derive sequential don't-cares, and doing combinational logic optimization along with some simple state re-encoding)? This was a first step; if successful, our next step would be to devise an algorithm for automatically selecting such an encoding. As we expected, it was possible to produce good circuits purely by choosing a good encoding: the logic optimization algorithms were able to clean up the mess made by manually adding the appropriate re-encoding and decoding circuitry and ultimately produce circuits as least as good as those from (expensive) sequential optimization. However, after considering the encodings we chose, it became clear that their choice was often guided by fairly detailed knowledge of the reachable state sets, and that not having this information would almost certainly lead to less efficient circuits.

But the news is not entirely bleak. While manually encoding the five examples, we found many cases of local redundancy, i.e., that could be found through inexpensive algorithms that did not have to consider inter-machine interactions.

In the last section, we conclude that we need a synthesis technique consisting of local, peephole optimizations followed by judiciously-applied global optimization. In the following sections, we discuss how we conducted the experiments that lead us to this conclusions, the details of the circuits we optimized, and the structures we found in the process.

2 Related Work

Multi-level combinational logic optimization [5] was the key that enabled the synthesis of efficient circuits from high-level descriptions, and of course the obvious next step was to extend it to synthesizing sequential circuits [16]. However, despite many years of research on sequential circuit synthesis and high-level synthesis [6], the industry remains stuck using register-transfer level specifications and multi-level combinational logic optimization.

Sequential optimization is much more challenging than its combinational counterpart. One difficulty is that it contains combinational optimization as a subroutine, so successful algorithms, such as De Micheli et al.'s KISS [7] need to consider how the combinational part of the generated circuit will be optimized. De Micheli's KISS and Villa et al.'s NOVA [19] target two-level implementations; Devadas et al.'s MUSTANG [8] and Lin et al.'s JEDI [12] target multi-level implementations. Bergamaschi et al. [1] show how such algorithms can be incorporated into datapath-centric high-level synthesis.

These algorithms also suffer from a severe scaling problem because they require an explicit representation of states and transitions. While this usually is practical for a single FSM, it often is not for FSMs running concurrently, which together can have exponentially many more states than the originals.

Other approaches have tried to side-step the scalability issue by working directly on the circuit structure. Leiserson and Saxe's retiming [11] scales very well, but only addresses timing and, by design, does not change the structure of the logic. Retiming and resynthesis [13, 16] exposes more opportunities for logic optimization using retiming, but is largely a heuristic and is a somewhat awkward way to re-encode a state machine.

To date, Touati and Berry [18] and Sentovich, Toma, and Berry [17] have come closest to addressing the problem we wish to solve. They propose an algorithm that uses information about the reachable state set to merge otherwise redundant latches in circuits generated from Esterel programs. This works particularly well with the sparse encoding generated by the standard synthesis algorithm [2], but is rather brute-force since it requires a complete symbolic state-space traversal.

Our goal, therefore, is to achieve results akin to Sentovich et al. without having to resort to performing complete state reachability. What follows is a discussion of how we were able to do this manually and the things we found along the way.

3 Methodology

For five examples, ranging from a simple six-bit Gray code counter to a fairly sophisticated bus controller, we compared the quality of two circuits. We synthesized the first circuit by sending the unoptimized output of the Esterel V5 compiler to the *blifopt* script in SIS [15, 16], which performs aggressive sequential optimization by symbolically computing the reachable state set and using it to derive sequential don't-cares that are used for combinational logic optimization. For the second circuit, we used CEC, the Columbia Esterel Compiler, to produce a circuit essentially the same as that from V5, then manually massaged the generated BLIF file by removing the original set of flip-flops, added our own, adding encoding and decoding logic, and passed this to SIS for purely combinational optimization. Finally, to validate our encoding, we ran VIS [4] to verify that the circuits were sequentially identical.

Both Esterel compilers use Berry's standard structural translation [2] of Esterel into hardware, which encodes independent FSMs using a one-hot-like code; separately-running FSMs are given independent state registers. Such a technique generates fairly fast circuits, but usually with substantial sequential redundancy that we found could slow the circuit.

example	lines of code	synthesis method	levels of logic	look-up tables	latches
graycounter	91	V5 + blifopt	5	66	27
		manual	4	51	17
abcdef	142	V5 + blifopt	5	114	25
		manual	3	128	8
mem-ctrl	80	V5 + blifopt	3	24	16
		CEC + comb	3	52	17
		CEC + blifopt	3	27	15
		manual	2	31	13
		Original VHDL	2	17	11
mem-ctrl2	36	V5 + blifopt	2	17	8
		CEC + comb	2	23	9
		CEC + blifopt	2	18	8
		manual	2	14	3
		JEDI + comb	2	14	3
tcint	689	V5 + blifopt	5	93	52
		manual	3	118	52

Table 1: Synthesis results for the examples.

3.1 Sequential Optimization: Blifopt

Our automatic path takes a circuit from the Esterel V5 compiler, feeds it to SIS, and runs the *blifopt* script, developed by Sentovich et al. [17]. The main power of *blifopt* comes from extracting the sequential don't-care information from the network by computing the circuit's set of reachable states symbolically (using BDDs) and running a sequence of algorithms that take advantage of it: *full-simplify*, which optimizes nodes using the Espresso algorithm, *equiv-nets*, and *remove-latches*. Each algorithm treats unreachable states as don't-care conditions on the inputs to the combinational part of the circuit, which is well-known to work well at simplifying the logic.

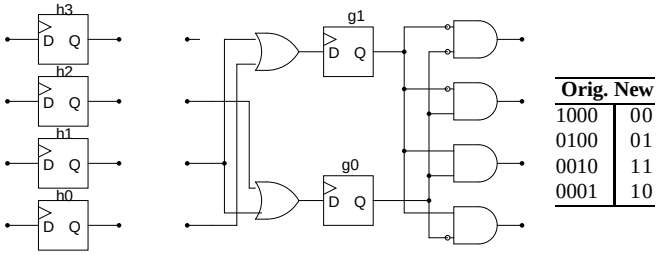
Computing and using sequential don't-cares is very powerful, but the *remove-latches* adds yet another dimension of flexibility by performing local state re-encoding using the algorithm of Sentovich, Toma, and Berry [17]. Briefly, due to the one hot original encoding in the Esterel circuit, there is a lot of register redundancy; the algorithm can remove certain registers if their function can be easily recomputed using the remaining ones. Generalizations have been proposed, but the algorithm remains an incremental optimization.

3.2 Manual Encoding

In our manual flow, we ran the Columbia Esterel Compiler (CEC), removed the latches from the generated BLIF file, and added our own latches and encoding/decoding logic and sequential don't-care information. In particular, we did not modify any of the combinational logic generated by the compiler. As a result, our hand-massaged circuit is initially worse than one that uses the standard encoding, but we then ran the same combinational optimization step we used in our automatic flow (i.e., skipping the *remove-latches* operation).

```
module Bit:
input CLK;
output B, CY;
loop
  await CLK;
  abort sustain B when CLK;
  emit CY;
  abort sustain B when CLK;
  await CLK;
  emit CY
end loop
end module
```

Figure 1: Esterel code for one of the bits in the Gray counter.



Orig.	New
1000	00
0100	01
0010	11
0001	10

(a) (b) (c)

Figure 2: For graycounter, (a) original state storage for each bit's FSM, (b) manually-designed encoding/decoding circuitry, and (c) its truth table.

4 Experiments

For each of the five circuits we analyzed, we found a better state encoding that lead to faster circuits with fewer registers (see Table 1), beating what *blifopt* was able to find using its reachable state analysis and combinational optimization. This is promising: it suggests there could exist an algorithm that could noticeably improve the quality of circuits just by choosing a suitable state encoding.

However, to reach these levels of quality, we needed fairly deep insight into each of these circuits to choose a better state encoding. In the sections that follow, we describe each of these circuits and how we chose our good encoding.

4.1 Graycounter

This example is a simple six-bit Gray-code counter with an alarm. The Esterel code is written in a ripple-carry style (each bit looks like Figure 1), so the default translation produces fairly deep logic. Each one-bit machine has four states, and each alarm machine is a two-state FSM.

Although the counter's state could be encoded using six bits, we found instead that re-encoding each four-state FSM using a pair of bits was superior. Figure 2 illustrates the transformation. We also re-encoded each two-bit alarm FSM with a single flip-flop.

We manually computed all remaining sequential don't-cares and added them to the BLIF file. The resulting circuit was smaller, faster, and used fewer flip-flops than what *blifopt* was able to obtain (Table 1). Here, the manual re-encoding was easy and could be done locally because the circuit was regular.

```
module ONE_BUTTON:
input BUTTON, LOCK;
output SELECTED_ON, SELECTED_OFF;
output LOCKED_ON,  LOCKED_OFF;
inputoutput SELECTED, LOCKED, UNLOCKED;

emit SELECTED_OFF; emit LOCKED_OFF;
loop
  trap WAIT_FOR_SELECTION in

    trap NOW_SELECTED in  % Wait to be selected
      loop
        abort
          await BUTTON do % S1
            exit NOW_SELECTED % We were pressed
          end await
        when LOCKED;      % We have been locked out
        await UNLOCKED    % S2: Wait to be released
    end end;

    loop                  % Selected or locked
      emit SELECTED_ON;   % We were selected
      emit SELECTED;      % Disable other buttons
      abort
        await             % S3
          case BUTTON     % User disabled us or
          case SELECTED   % other button pressed
        end await;
        emit SELECTED_OFF; % We lost the selection
        exit WAIT_FOR_SELECTION
      when LOCK;

      emit LOCKED_ON;     % We are now locked and
      emit SELECTED_OFF;  % no longer selected.
      emit LOCKED;        % Lock out others and
      await LOCK;         % S4: wait for unlock
      emit LOCKED_OFF;    % Announce it and
      emit UNLOCKED       % release other buttons.
end end end end
end module
```

Figure 3: Esterel source for one button of the abcdef example. The BUTTON input toggles its selection when the system is unlocked, and LOCK locks the most-recently selected button until LOCK is pressed again.

4.2 Abcdef

The inputs for this model, a simple user interface, are six buttons labeled A through F and one labeled LOCK. Figure 3 is the code for one button. Each is in one of four states: waiting to be selected (S1), unselected and waiting to be unlocked (S2), selected and waiting to be locked or unselected (S3), and selected and waiting to be unlocked (S4). The overall system is composed of six such machines running concurrently and communicating through the SELECTED, LOCKED, and UNLOCKED signals.

By design, the six machines are not independent: most combinations of states are unreachable. Specifically, SELECTED, sent when a button is pressed and the system is not in the "locked" state, causes all the other machines to become unselected. In fact, there are three types of states the system may be in: all the machines in S1 (nothing selected); one machine in S3 (selected), the others in S1; and one machine in S4 (locked), the others in S2.

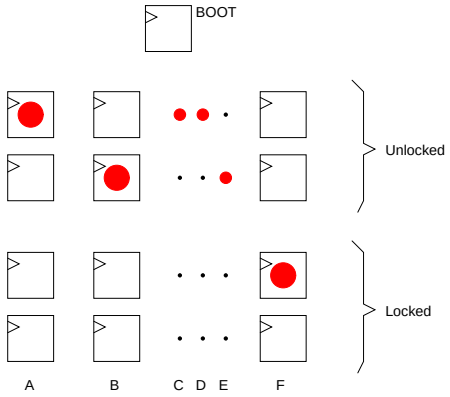


Figure 4: Original 25-latch encoding for the abcdef example, shown in an unreachable state. Each button can be in one of four states, but together, all buttons must either be in one of the unlocked states or in one of the locked states. Here, only F is in the locked state.

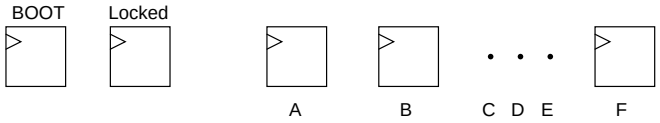


Figure 5: Our 8-latch encoding for abcdef example. One latch indicates whether all buttons are locked or unlocked, and six indicate which button is selected.

Instead of the default one-hot encoding, which assigns four flip-flops to each button (Figure 4), we chose to use one flip-flop to record whether any of the buttons are in the locked state, and one flip-flop for each button to indicate whether it is selected or locked (Figure 5). There is still some sequential redundancy in our encoding (in particular, no two of A–F can be set simultaneously); we manually added these sequential don't-cares to the BLIF file. Our encoding produced a much faster circuit (Table 1) with one-third as many registers.

This design illustrates an emergent property—that at most one button may be selected or locked simultaneously—that is not obvious from the source code. Why the designer chose this particular style to express the system is unclear. Although it has the advantage of scaling (adding a button amounts to adding another button process), it seems overly clever since omitting, say, a single *emit* would violate the desired mutual-exclusion property. Furthermore, knowing this property was crucial in improving the quality of the circuit, and discovering it would probably be challenging.

4.3 Memory controller

This arbitrates between a processor and a video system trying to gain access to a shared memory and generates the control signals for the RAM. It was written by an inexperienced Esterel coder who translated it from a VHDL implementation.

The circuit is approximately a simple FSM, but this is not obvious from looking at the original VHDL, which was written in a pure dataflow style using one-hot encoding. By contrast, the Esterel is written as a collection of concurrent processes,

```
||
present [not rnw and onecycle] then
   % now in xfer state
   pause;
   emit xfer;
   emit rres;
end present
||
present [not rnw and not onecycle] then
   % now in w state
   await [not vreq];
   % now in xfer state
   pause;
   emit xfer;
   emit rres;
end present
||
```

Figure 6: Fragment of the memory controller source. The two concurrent threads are actually mutually exclusive.

each apparently trying to reproduce a particular trace of the original machine. This leads to substantial sequential redundancy because it turns out that most of these traces cannot happen simultaneously (again, this is not clear from the VHDL, which was coded knowing environmental constraints).

The initial encoding encoded twelve states with seventeen latches. Naturally, most codes were unreachable or equivalent. At some point, the designer became aware of this, as evidenced by the comments, but did not change the code to reduce the number of unreachable states.

Our manual re-encoding attempted to exploit this redundancy, which led to some improvement (Table 1), but because the structure of the initial circuit was so very different than the original VHDL implementation, we were not able to devise encoding/decoding logic to bring it to the level of the VHDL original. We suspect that with an automatic tool (e.g., that would establish sequential equivalence) we probably could get it to that level, but this would be computationally expensive and not very illuminating.

In this example, the high-level structure of the Esterel code was outright misleading. Rather than containing a clever emergent property like the abcdef example, it was poorly coded and its correct behavior was almost an accidental side-effect brought on by copying a working design.

4.4 Memory Controller Version Two

In fact, the first Esterel version of the memory controller had a bug: the Esterel coder had inadvertently inserted an extra cycle in one of the traces. So we rewrote the example from scratch, making sure to match the behavior of the original (simpler) VHDL implementation exactly and with an eye toward an efficient translation.

Even after only combinational optimization, the fixed memory controller was superior to all but the original VHDL implementation (Table 1). In fact, the effects of sequential optimization were somewhat limited on this example. More impressive were manual encoding and complete resynthesis using Lin's JEDI [12], which was possible because the controller is small.

```
pause;  % to avoid problems at boot time!
loop
  await  % DMA request or SEL
    case immediate [Fo_HF and DMAWrAddrRdy] do
      run DMA_WRITE
    case immediate [not Fi_HF and DMARdAddrRdy] do
      run DMA_READ

    case immediate SEL do  % SEL : decode opcode
      emit TagFlag;
      trap ReadSharedEnd, WriteSharedEnd in
        present [SEL and WRITE and not ADB24 and
                         ADB23 and not ADB22] then
          run WPOM
        else present [SEL and not WRITE and
            not ADB24 and ADB23 and not ADB22] then
          run RPOM; exit ReadSharedEnd
        else present [SEL and WRITE and ADB24] then
          run WPAM
        else present [SEL and not WRITE and
                                      ADB24] then
          run RPAM; exit ReadSharedEnd
        else present [SEL and WRITE and not ADB24
                         and ADB23 and ADB22] then
          run WFIFO
        else present [SEL and not WRITE and
                 not ADB24 and ADB23 and ADB22] then
          run RFIFO; exit ReadSharedEnd
        else present [SEL and not WRITE and not
            ADB24 and not ADB23 and not ADB22] then
          run RROM; exit ReadSharedEnd
        else present [SEL and WRITE and not ADB24
                       and not ADB23 and ADB22] then
          run WLCA
        else present [SEL and not WRITE and
            not ADB24 and not ADB23 and ADB22] then
          run RLCA; exit ReadSharedEnd
        else
          halt
        end end end end end end end end end

      handle ReadSharedEnd do
        % drive final data word on next cycle
        emit pDriveTBC;
        pause;
        % send RDY and pHostDrives, wait one cycle
        emit RDY;
        emit pHostDrives;
        pause
      end trap
  end await
end loop
```

Figure 7: Fragment of the tcint example: the selection cycle

4.5 Tcint

Tcint is a TurboChannel bus controller implemented in 680 lines of Esterel. Its main loop (Figure 7) begins by checking ADDR, SEL, and WRITE to determine whether a cycle is DMA read or write, or an operation directed to one of eleven submodules (each controls a peripheral—such as a FIFO—that is outside the model), and then starts a sub-machine as appropriate. When these sub-machines terminate, the main loop restarts. The decisions made in the main loop—a form of chip select—are costly and time-critical.

The main selection module together with the eleven submodules behaves as a large sequential FSM since it is impossible for two sub-modules to be active simultaneously. However, several auxiliary FSMs run concurrently and exchange signals with the main FSM. The timing of some of these signals (e.g., *DMAWrAddrRdy* and *DMARdAddrRdy*) is also critical.

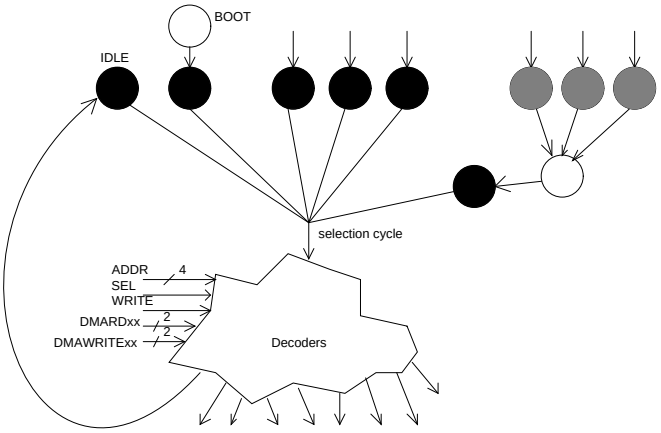


Figure 8: The selection cycle in tcint: states with the same color are equivalent. “Decoders” represents the combinational next-state logic.

```
trap AckReceived in
  await tick;
  sustain TCRegOutCkDis
||
  await immediate ACK;
  exit AckReceived
end trap;
```

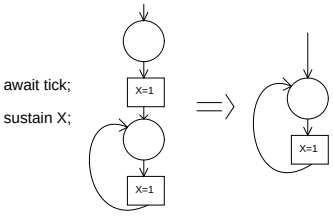


Figure 9: Esterel fragment generating equivalent states and its state transition graph.

We first chose an encoding that alleviated the critical paths. We observed many of the states in the main FSM were equivalent, mostly the last states of the various sub-modules. This was unfortunate since most of these states appear the cycle before a costly selection cycle, thereby making it necessary to compute the logical OR of all these states in addition to decoding the various address lines during a critical selection cycle (Figure 8). We merged these states carefully, avoiding creating any new critical paths while optimizing the existing ones. In this case, the transformation was effectively a retiming.

We also found many equivalent states in auxiliary FSMs. These derived from valid, but suboptimal, Esterel constructs. Figure 9 illustrates such a case.

We also reencoded certain small FSMs with states that were equivalent to the initial state (Figure 10). Here we considered a more global reachability question. Since *ConflictOnSEL* is not active in the boot state, the machine is guaranteed to remain in the desired state after the first cycle.

We also found that detecting a valid selection state, which can never occur in the initial state, had a false path coming from the register used to represent the initial state. We added this don’t-care explicitly and SIS cheerfully removed the useless signals from the critical paths.

At this point, the critical paths ceased to involve the main selection logic, but timing problems remained for an auxiliary Moore FSM (DRIVE: Figure 11), which was driven by many signals generated by the main FSM (*pPamDrives*, *pRomDrives*, *pLcaDrives* and *pHostDrives*).

```
loop
  await ConflictOnSEL;
  do
    every immediate SEL do
      emit RejectSEL
    end
  watching AcceptSEL
end loop
```

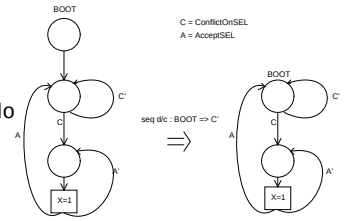


Figure 10: State merging using sequential don't-cares: *ConflictOnSEL* is never asserted in the initial state.

```
loop
  trap EndOfHostDrives in
    await tick;
    [  sustain TCRegInOE
    || sustain ExtBufDir
    || sustain ExtBufOE ]
  ||
    await [pPamDrives or pRomDrives or pLcaDrives];
    exit EndOfHostDrives
  end;
  present pPamDrives then
    trap EndOfPamDrives in
      await tick;
      sustain ExtBufOE
    ||
      await pHostDrives do exit EndOfPamDrives end
    end
  else present pRomDrives then
    trap EndOfRomDrives in
      await tick;
      sustain RomOE
    ||
      await pHostDrives do exit EndOfRomDrives end
    end
  else present pLcaDrives then
    trap EndOfLcaDrives in
      await tick;
      sustain LcaOE
    ||
      await pHostDrives do exit EndOfLcaDrives end
    end
  end end end
end
```

Figure 11: Fragment of tcint: the DRIVE FSM, which turns out to run in lockstep with the main FSM.

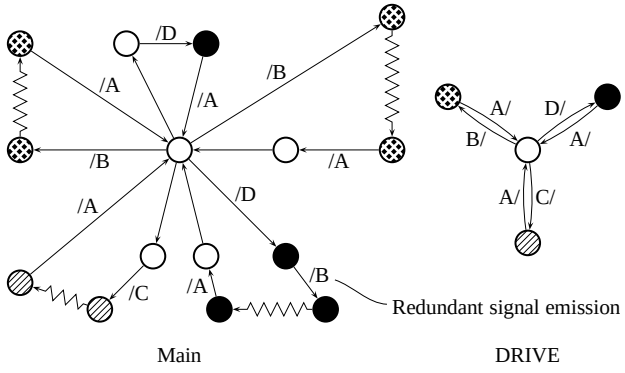


Figure 12: An abstraction of the Main and DRIVE machines in tcint. Coded as two independent machines, they actually operate in lockstep. Four control signals ensure that both machines are both in a white state, both in a black state, etc. Wavy lines represent multiple intermediate states and transitions.

```
% during first cycle :
% * start sustaining pWREQ: on the next cycle, we
%   shall have WREQ and DMA address ready
% * prepare Lca drive for next cycle

emit pLcaDrives;
await tick;
% setup data path from pam to host
emit pPamDrives;
% ...
emit pHostDrives;
```

Figure 13: Excerpt from the DMA_WRITE machine in Tcint.

We initially reencoded part of the main FSM to provide the mentioned internal signals earlier. This was easy since the machine was not critical at those points, but this was not enough.

The key observation was that the DRIVE FSM ran in lockstep with the main FSM. For each state of the main FSM, DRIVE can be in only a certain state. Figure 12 shows an abstraction of the problem in which the colors of the various states depict the constraints.

Many control signals are irrelevant in certain states. This was not obvious: the Esterel source suggested the machines operated independently. After manually adding don't-cares for these cases, we finally obtained three levels of logic.

Sequential analysis turned out to be crucial in tcint, as there were many unreachable and equivalent states and false paths. Some of them were obvious from the Esterel source, but others were more subtle, emergent properties that were critical.

To emphasize how much sequential redundancy there is in tcint, only 2282 states are reachable, which is somewhat surprising for a circuit with fifty-two latches. The more remarkable result is that only 231 of these are actually unique. (Since this example was small enough, we were able to use SIS's *stg_extract* command to enumerate the states and the stamina tool to minimize them.)

Since the main FSM in tcint actually runs in lockstep with *DRIVE*, one signal emission is useless and can be removed. This signal is asserted in the DMAWRITE submodule.

The "B" is the guilty signal; (Figure 13 shows the relevant code); our signal is asserted by the "emit pPamDrives" statement. It comes one cycle after "emit pLcaDrives," so it is useless (when the module starts, DRIVE is in the "white" state).

5 Conclusions

We began this work by asking what information was necessary to choose good state encodings for control-dominated circuits expressed in the Esterel language. Our initial hypothesis [10], based on our previous success with generating software from Esterel using high-level information [9], was that considering the high-level structure of the Esterel program would be enough and that we would not have to resort to performing an exact, global state reachability computation.

Our results suggest we were half right: many of the examples had substantial local redundancy that could be detected and corrected easily, i.e., without considering interactions between concurrently-running machines. However, to reach an acceptable quality, we had to resort to more global analysis.

Overall, this suggests a two-pronged approach to the problem of optimizing complex sequential controllers: aggressive, exhaustive local optimizations followed by a more global analysis and possibly even resynthesis. Seawright and Meyer [14] propose a technique like this, but their starting point, a regular-expression-like language, appears to have even higher levels of sequential redundancy than is typical in Esterel programs.

An interesting question is how much responsibility should be borne by the coder. Clearly, some ways of expressing behavior lead to more efficient circuits than others, but at the same time, it is often more elegant to write the less efficient version. For example, consider these sequentially equivalent fragments:

```
pause;                    loop
sustain A       vs.         pause;
                            emit A
                          end loop
```

We found the more succinct version (left) in Tcint, but it produces a less efficient circuit since it produces two (equivalent) states whereas the right version only gives one. However, the behavior of the left fragment is clearer and thus faster to write, debug, and modify; it is what we would like to write. So in this case, it should be the responsibility of the compiler to find the more efficient implementation for the left fragment.

However, how far should the compiler's reencoding responsibility extend? Of the five examples we considered, *abcdef* required the most aggressive analysis because, by design, the system had a emergent property of mutual exclusion between machines, something that requires inter-machine analysis to discover. Now, this example had the property that the number of states in the resulting machine grew very slowly with the size of the example (i.e., linearly instead of exponentially), so it would be feasible to have used explicit state enumeration to discover this property. However, such a technique will fail in general because of state explosion, so we need a clever heuristic to tell us where and when to perform explicit state enumeration. Seawright and Meyer [14] used a simple threshold, but would a similar technique work for Esterel?

In the end, our experiments suggest that a mixture of algorithms will be the solution to generating efficient state encodings for complex controllers. Devising such algorithms, which we have started to do, is the next step.

References

[1] R. A. Bergamaschi, D. Lobo, and A. Kuehlmann. Control optimization in high-level synthesis using behavioral don't cares. In *Proceedings of the 29th Design Automation Conference*, pages 657–661, Anaheim, California, June 1992.

[2] G. Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 339:87–103, Apr. 1992. Issue 1652, Mechanized Reasoning and Hardware Design.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, Nov. 1992.

[4] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[5] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, Feb. 1990.

[6] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[7] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-4(3):269–285, jul 1985.

[8] S. Devadas, H.-K. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines targeting multilevel logic implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(12):1290–1300, Dec. 1988.

[9] S. A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2):169–183, Feb. 2002.

[10] S. A. Edwards. High-level synthesis from the synchronous language Esterel. In *Proceedings of the International Workshop on Logic Synthesis (IWLS)*, New Orleans, Louisiana, June 2002.

[11] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[12] B. Lin and A. R. Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *Proceedings of IFIP International Conference on VLSI*, pages 187–196, Munich, West Germany, Aug. 1989. Elsevier.

[13] S. Malik, E. M. Sentovich, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):74–84, Jan. 1991.

[14] A. Seawright and W. Meyer. Partitioning and optimizing controllers synthesized from hierarchical high-level descriptions. In *Proceedings of the 35th Design Automation Conference*, pages 770–775, San Francisco, California, June 1998.

[15] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[16] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 328–333, Cambridge, Massachusetts, Oct. 1992.

[17] E. M. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 428–435, San Jose, California, Nov. 1996.

[18] H. Touati and G. Berry. Optimized controller synthesis using Esterel. In *Proceedings of the International Workshop on Logic Synthesis (IWLS)*, Tahoe City, California, May 1993.

[19] T. Villa and A. Sangiovanni-Vincentelli. NOVA: state assignment of finite state machines for optimal two-level logic implementations. In *Proceedings of the 26th Design Automation Conference*, pages 327–332, Las Vegas, Nevada, June 1989.