Sequential Challenges in Synthesizing Esterel

Cristian Soviani Jia Zeng Stephen A. Edwards

December 2004

Abstract

State assignment is a formidable task. As designs written in a hardware description language such as Esterel inherently carry more high level information that a register transfer level model, such information can be used to guide the encoding process. A question arises if the high level information alone is strong enough to suggest an efficient state assignment, allowing low-level details to be ignored.

This report suggests that with Esterel's flexibility, most optimization potential is *not* within the high-level structure. It appears effective state assignment cannot rely solely on high level information.

Contents

1 Introduction

While the use of hardware description languages such as Verilog and VHDL have become the *de facto* approach to circuit design, the natural trend is toward more abstract HDLs. The key issue is the performance penalty this introduces. As general approaches—at this moment—do not produce satisfactory results, a compromise can be found in domain specific languages, which can minimize this penalty at the price of less generality: such a language can handle a limited range of problems very well.

Esterel is a synchronous language developed by Gérard Berry et al. [2] to handle reactive systems. Its imperative (C-like) semantics makes Esterel a good choice for designing controllers, as they usually consist of several interconnected (more or less sequential) processes.

1.1 Synthesizing Esterel

Given its synchronous semantics (i.e., signal timing is specified exactly with respect to a global clock), an Esterel program is in fact an RTL specification; so the resulting circuit is a FSM in the exact sense.

A syntactic translation of Esterel into a circuit [1, 4] gives a hierarchical structure of inter-connected sub-machines; so the circuit (i.e., product machine) state transition graph is represented implicitly. Even for medium Esterel programs, building the explicit STG is impossible due to its size, so any synthesis technique can not rely directly on classical state minimization, assignment, etc.

The current synthesis technique (as implemented in Esterel v5 and our compiler CEC [5, 3]) starts by a trivial encoding, using the high level structure of the circuit. After this step, a sequence of powerful sequential algorithms is run on the highly redundant flattened network (see "blifopt" below); though much cheaper than synthesizing from the STG, these algorithms are impractical for medium to large circuits, so a compromise has to be carefully chosen.

Network sequential optimization is a mature field; to alleviate the inherently complexity of the problem, many sophisticated techniques are used in present tools. However, sequential optimization is a major bottleneck in logic synthesis; it would be very interesting if we can bypass (some of) these expensive techniques and obtain the same network quality by exploiting the high level information provided by the HDL.

1.2 The Question We Address

Given the complexity of the issue, we proposed to answer a simpler question: starting from an Esterel program, can we generate an initial state assignment such that the resulting network can be successfully optimized using only combinational optimization?

2 Experimental Setup

2.1 Overview

In this this study we used CEC (Columbia Esterel Compiler) and Esterel v5 (the Esterel release by Berry et al.). Both have the ability to generate BLIF, which can be optimized, verified, and benchmarked within the SIS and VIS environments.

In generating a network from an Esterel source, CEC builds an abstract representation of the circuit state as a tree structure, which closely follows the syntactic structure of the source. The output and next state functions are constructed independently, so it is possible to encode the state in various ways while preserving the circuit functionality.

When one hot-encoding is used, CEC generates a circuit almost identical to Esterel v5. In addition to this trivial encoding, CEC can heuristically choose different encodings [5], trying to exploit some available high level information. However, no heuristic was found to consistently improve the circuit performance.

2.2 Blifopt

Esterel v5 does not output an optimized circuit. Instead, a SIS script named "blifopt" is provided, which is a powerful sequential tool, adapted to the v5 output (i.e., one hot encoding). It targets FPGAs, so the result is a k-feasible network; the main goal is performance (i.e., the number of logic levels, as the unit delay model is used). Even if more advanced sequential tools may be available, "blifopt" didactically illustrates the general idea.

The secret of "blifopt" consists in extracting the sequential don't-care information from the network and running a sequence of algorithms which can take advantage of it: "full-simplify" (ala espresso), "equiv_nets", and "remove_latches".

The "remove_latches" step is the most interesting to us, as it does some local re-encoding [6]. Briefly, due to the one hot original encoding, there is a lot of register redundancy. In this case, the algorithm can remove some registers if their function can be easily recomputed using the remaining ones. Some generalizations were proposed, but the trick remains an incremental optimization, instead of a complete re-synthesis (like nova, jedi, etc.).

The result of this algorithm can be far from what could be done by hand. This may be partially due to the combinational tools in the SIS package; it would be interesting to compare the results after using state-of-the-art FPGA synthesis.

2.3 Methodology

For a set of four sample programs, we first ran Esterel v5 followed by blifopt.

Next, we ran CEC using a state encoding we chose by hand. Note that we do not manually optimize the circuit; the transition functions remain unchanged, and the new encoding has to match more or less the high level structure (see Section 2.1).

Given these constraints, we chose the new encoding carefully; no given algorithmic steps were imposed, and several variants were tried, and we report only the best one we could find.

The resulting circuit is optimized by using SIS without computing any sequential don't cares. Finally, the blifopt FPGA mapping subroutine (also 100% combinational) is invoked to produce the 4-feasible network.

3 Experiments

3.1 abcdef.strl

This simple Esterel program is included in almost all Esterel benchmark set. Its complexity can be controlled by varying the number of buttons it describes; our test uses 6 (i.e., abcdef).

Each button (source in Figure 1) runs a four-state FSM; the circuit is build from six such FSMs, running in parallel; note that they are tightly inter-connected by internal signals (the *inputoutput* ones).

As the six FSMs are not independent, not all combinations of states are possible. The sequential analysis shows that—for each FSM—the four states can be "divided" into two classes: "L" and "R," each of which contain two states. The key point is that the six FSMs have to be either all in an "L" state or either all in an "R" state. The original one-hot encoding (Figure 2) does not take advantage of this fact; the red dots show a sample of such an unreachable state (note that some buttons (a,b,c,d,e) are in a "L" class state while f is in an "R" class one).

Our encoding follows from this observation. We use one bit to select between "L" and "R" states, one bit for each FSM to select between the two possible states (inside "L" or "R" classes); and one bit for the boot state (Figure 3). The new encoding has very little sequential redundancy; the few existing sequential don't cares were manually added to the BLIF file.

This is a "lucky" example, as a very efficient encoding could be found manually. Moreover, it fits easily into the existing high level structure.

The results in Table 1 show this new encoding helped. However, the key in finding it was the manual analysis of the whole circuit, including the reachable states pattern; the circuit symmetry made the analysis much easier, but it is questionable if that kind of human reasoning can be automatically reproduced.

	# levels	# LUTs	# latches
v5 + blifopt	5	**114**	25
manual enc	**3**	128	**8**

Table 1: Synthesis results for abcdef.strl

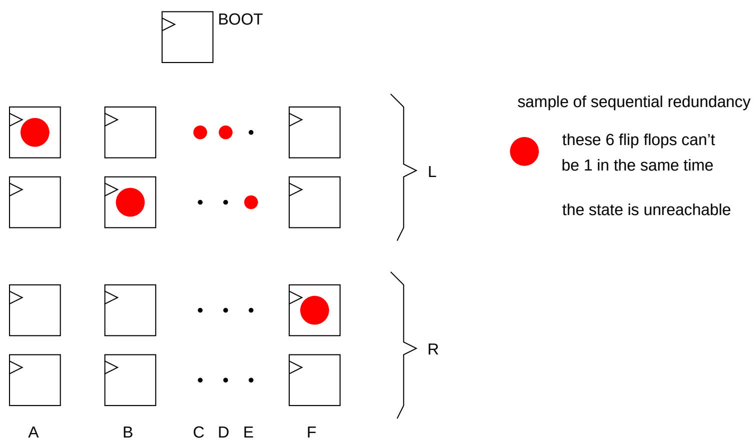


Figure 2: Default one-hot encoding for abcdef.strl, requiring twenty-five latches.

3.2 greycounter.strl

This six-bit grey counter with alarm was designed by one of us (Soviani) while learning Esterel. The design uses carry chains both for counting and alarm generation, so the trivial translation has relatively deep logic; this implies slower operation compared to a state-of-the-art grey counter.

In the source, each counter bit runs a four-state FSM (Figure 4), except the last one, which needs only two states. The alarm sub-circuit uses a two-state FSM for each bit, which stores the alarm pattern.

The alarm FSMs were trivially encoded by single flip-flops (i.e., one bit each). Note that the counter state could be encoded on only six bits, but that generated low performance. So we re-encoded the "bit" FSMs from one hot (four bits) to "grey" (two bits) (Figure 5).

The remaining sequential don't-cares were computed manually and added to the BLIF file. The results (Table 2) look better; like in the previous example, the circuit is highly regular, which made the manual re-encoding much easier.

3.3 memory-controller.strl

This is a very simple version of a memory controller for IBM's on-chip peripheral bus (OPB) used on a Xilinx Spartan-II prototyping board; it was written by a student who tried to copy the functionality of an existing VHDL design. He was not very proficient in Esterel at that point, making the sample even more interesting for our analysis (see Section 4.4).

The circuit is very simple: it is mainly a sequential FSM. However, the designer did not see this from the VHDL code. He analyzed various behaviors of the circuit (i.e., traces) and put them together (see Figure 6). He appeared to cover all the cases, a potential pitfall with this approach, but the sequential redundancy is huge. For example, the two FSMs running in parallel in the code sample can not be active in the same time. Similar constructs are present in the rest of the code.

The straightforward encoding uses seventeen latches. As the controller has actually only twelve states, most of the codes are either unreachable or equivalent. Note that at some point the designer was fully aware of that (see the comments).

Our manual re-encoding tried to exploit that redundancy. Even if some improvement can be noticed (Table 3), the quality is far from the original VHDL

	# levels	# LUTs	# latches
v5 + blifopt	5	66	27
manual enc	**4**	**51**	**17**

Table 2: Synthesis results for greycounter.strl

```
module ONE_BUTTON:
input BUTTON;
input LOCK, UNLOCK;
output BUTTON_PRESELECTED_ON;
output BUTTON_PRESELECTED_OFF;
output BUTTON_LOCKED_ON;
output BUTTON_LOCKED_OFF;
inputoutput PRESELECTED;
inputoutput LOCKED, UNLOCKED;

emit BUTTON_PRESELECTED_OFF;
emit BUTTON_LOCKED_OFF;
loop
  trap BACK_TO_MAIN_LOOP in
    trap PRESELECTED in
      loop
        do
          await BUTTON do
            exit PRESELECTED
          end await
        upto LOCKED;
        await UNLOCKED
      end loop
    end trap;
    loop
      emit PRESELECTED;
      emit BUTTON_PRESELECTED_ON;
      do
        await
          case BUTTON
          case PRESELECTED
        end await;
        emit BUTTON_PRESELECTED_OFF;
        exit BACK_TO_MAIN_LOOP
      watching [LOCK] timeout
        emit BUTTON_PRESELECTED_OFF;
        emit LOCKED;
        emit BUTTON_LOCKED_ON;
        await UNLOCK do
          emit BUTTON_LOCKED_OFF;
          emit UNLOCKED
        end await
      end timeout
    end loop
  end trap
end loop
end module
```

Figure 1: abcdef.strl: Esterel source for a single button. The complete example contains six of these.

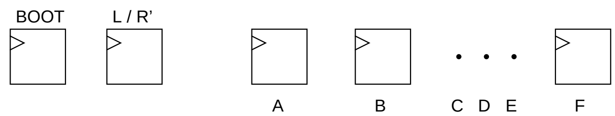
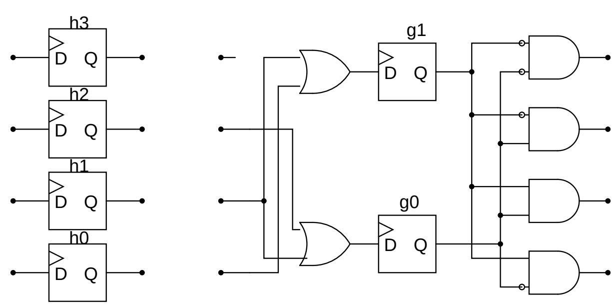


Figure 3: Compact encoding for abcdef.strl, requiring eight latches.

```
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 4: Simplified Esterel code for a counter bit in greycounter.strl.



one-hot				grey	
h3	h2	h1	h0	g1	g0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	1
0	0	0	1	1	0

Figure 5: Bit-grey re-encoding for greycounter.strl.

```
... ... ...
 ||
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: memory-controller.strl: Fragment of Esterel source.

design, which in turn was a lousy encoding of the almost trivial STG.

Unfortunately, it is very hard to detect this kind of sequential redundancy by high-level inspection, as the high-level structure in the above fragment suggests two independent machines. Only a Boolean analysis would detect that they are mutually exclusive and that the last two states in each machine (xfer) are equivalent.

3.4 Tcint.strl

Tcint is one of the few available medium-size Esterel programs designed for hardware implementation. It implements a TurboChannel bus interface.

Briefly, while in a "selection" cycle, the circuit checks which of the eleven available submodules is

	# levels	# LUTs	# latches
v5 + seq	3	**24**	16
manual enc	**2**	31	**13**

Table 3: Synthesis results for memory-controller.strl

selected (mainly looking at the ADDR*, SEL, and WRITE inputs) and activates it accordingly. A new "selection" cycle occurs after the selected module finishes its task. Otherwise, if no module is selected, the cycle is "idle" and the next cycle becomes a valid "selection" one (source in Figure 7).

This process is time-critical as it takes a long time to know whether a particular cycle is a valid "selection" one, and a lot of logical (decoding) operations are done to decide which device is selected. Note that the DMA operations have priority over the regular I/O ones.

The main selection module together with the eleven sub-modules, can be seen as a large sequential FSM, as it is impossible for two of them to be active in the same time. Several auxiliary small FSMs exchange internal signals with this main FSM; some of these are also critical (note for example the signals *DMAWriteAddressReady* and *DMAReadAddressReady*).

Initially, we attempted the alleviate these critical paths in our manual re-encoding.

First, we could see that a lot of states in the main sequential FSM were equivalent (Figure 8); briefly, they are the last states of the various sub-modules; as it can be seen, the "red" states are responsible for detecting a valid "selection" cycle. They were merged together. This we did carefully to avoid creating new critical paths while fighting to optimize the existing ones. Note that in this simple case the transformation is a simple retiming; exploiting equivalent states may be more complicated in the general case.

A lot of local equivalent states were also found in various points in the auxiliary FSMs; they were present due to suboptimal—but valid—Esterel constructs (Figure 9).

Some of the small FSMs were also re-encoded by finding states equivalent to the boot state (Figure 10). In this short sample, equivalent states could be found by inspecting the circuit reachability; as *ConflictOnSEL* is not active in the boot state, the machine is guaranteed to remain in the desired state after the first cycle.

It was also noticed that detecting a valid "selection" state, which can never occur in the boot cycle, had an incoming false path from the boot register.

```
await tick;  % to avoid problems at boot time!
  loop
    await  % First of all, wait for DMA request or SEL
      case immediate [ Fo_HF and DMAWriteAddressReady ] do
                run DMA_WRITE;
      case immediate [ not Fi_HF and DMAReadAddressReady ] 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;
              await tick;
              % send RDY and pHostDrives and wait one cycle
              emit RDY;
              emit pHostDrives;
              await tick
         end trap
    end await
  end loop
```

Figure 7: tcint.strl : Esterel source for the selection cycle

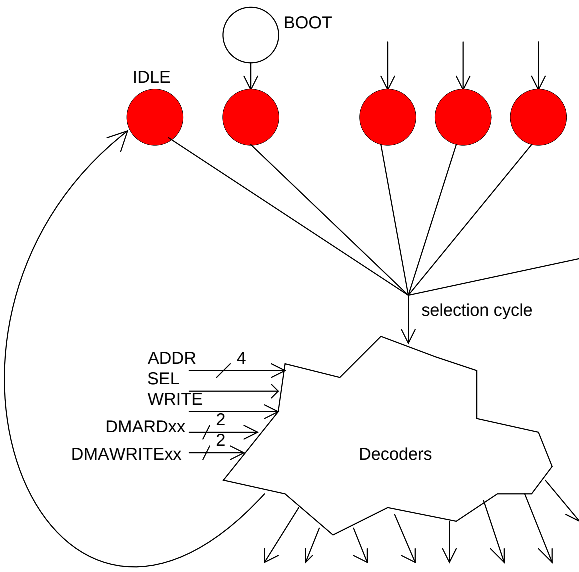


Figure 8: The tcint selection cycle. States with the same color are equivalent.

An explicit don't-care was manually added, and SIS removed the useless signal(s) from the critical paths.

At that point, the critical paths ceased to involve the main selection logic; but the timing problem remained for a small auxiliary Moore FSM (DRIVE, whose source is shown in Figure 11), which was driven by many internal signals generated by the main FSM (*pPamDrives*, *pRomDrives*, *pLcaDrives* and *pHostDrives*).

To begin with, we re-encoded parts of the main FSM to provide the mentioned internal signals earlier. This was easy since the paths were not critical at these points, but the result did not improve substantially.

The key observation was that the "guilty" (DRIVE) FSM runs in perfect lockstep with the main FSM (Figure 12). As the colors suggest, for each state of the main FSM, DRIVE can be in only a certain state.

As a result, many control signals are irrelevant in various states. The observation was not trivial, given the Esterel source that suggested an independent operation between the machines. After manually adding the corresponding don't cares, we achieved the desired three levels of logic almost immediately (Table 4).

This design is mostly responsible for the conclusions in the following section.

Sequential analysis is critical in tcint, as there are many unreachable and equivalent states, false paths, etc. Some of these optimization opportunities are easy to see in the Esterel source (i.e., from purely high-level analysis); but some are not, and—this is the bad news—many of these are critical.

To emphasize the quantity of "tcint" sequential redundancy, note that only 2282 state codes are reachable (v5 plus blifopt optimized output). That is a little bit strange for a fifty-two latch network. However, the real surprise was finding that there are only 231 distinct states after state minimization.

	# levels	# LUTs	# latches
v5 + blifopt	5	**93**	52
manual enc	**3**	118	52

Table 4: Synthesis results for tcint.strl

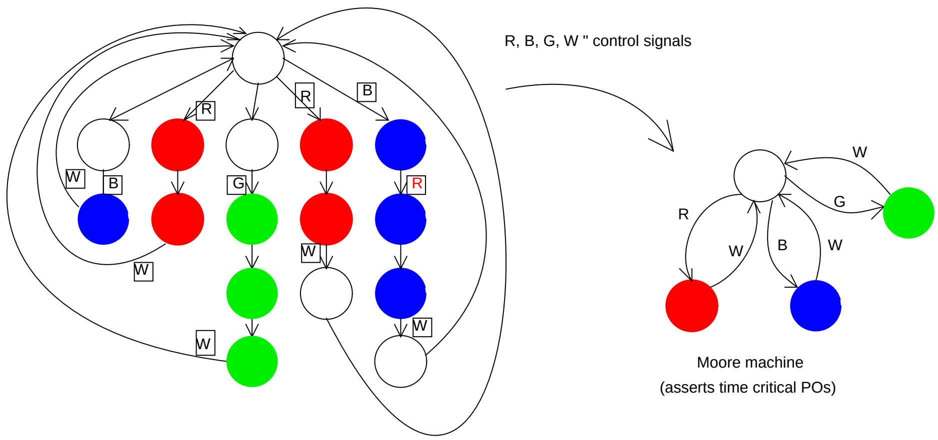


Figure 12: The DRIVE machine running in lockstep w/ the main one

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

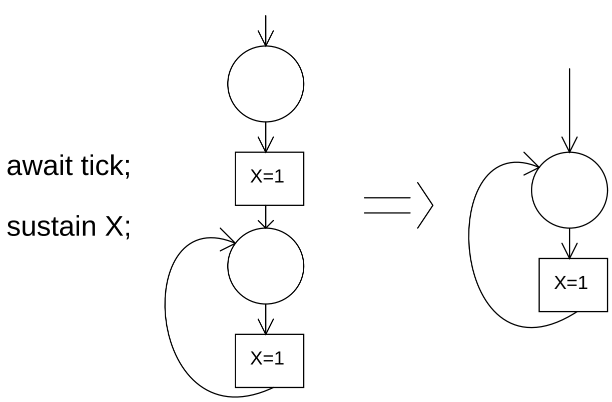


Figure 9: Simple Esterel construct generating equivalent states and the STG for a fraction of the above code

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

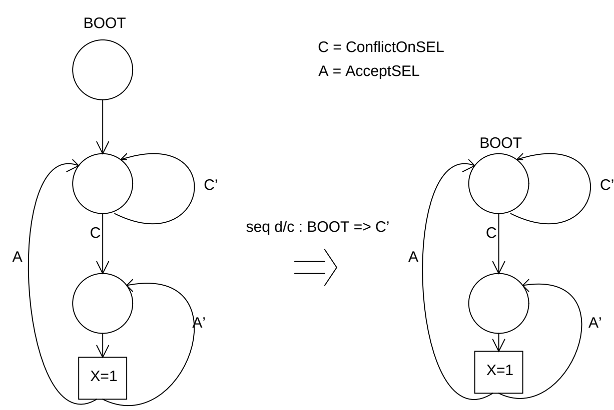


Figure 10: Equivalent boot state found using the sequential don't-care, ConflictOnSEL is never asserted in the BOOT state.

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

Figure 11: tcint.strl : Esterel source (fragment) for DRIVE FSM

4 High-Level Synthesis Issues

4.1 Early samples

Early experiments were done on self-designed samples. As the authors are familiar with VHDL design, and we were aware of the Esterel tool internals, CEC could generate good designs even without sequential optimization. We even occasionally felt the desire to specify an encoding for various FSMs (a common practice in VHDL). As a result, suspected that the tool might be able to find such an encoding itself, based on the high level information.

4.2 Real Samples and Limitations

The "real" Esterel code examples we used differ substantially from the samples we wrote to begin with.

Most people have experience with C or other sequential imperative languages, so they are much more comfortable with sequencing than concurrency. This problem is not particular to Esterel, but Esterel's support of an imperative style tends to encourage that style of problem solving.

This is not intrinsically a bad practice—after all the Esterel basic blocks are sequential constructs—but we did notice a dangerous tendency toward abusing the language's sequential features.

Note that for certain problems, a "nice" Esterel program may not exist, given that Esterel is a DSL and not a general purpose language.

4.3 A Curious Signal in Tcint

As seen above, the tcint main machine runs in lockstep with the DRIVE machine; one internals signal is totally useless and can be ignored. This signal is asserted in the DMAWRITE submodule; a more detailed view is shown in Figure 13.

The red "R" is the guilty signal; the Esterel code generating it is shown in Figure 14; our signal is asserted by the "emit pPamDrives" statement. It comes one cycle after "emit pLcaDrives", so it's useless (when the module starts, DRIVE is in the "white" state). However, it has a clear comment; it should be useful.

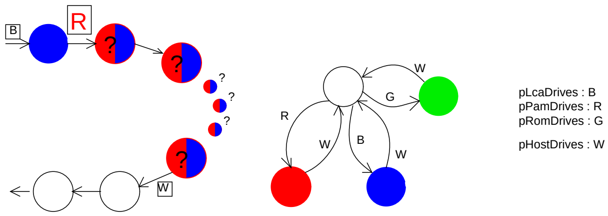


Figure 13: tcint.strl: DMAWRITE : red or blue?

```
module DMA_WRITE:
... ... ...
% prepare Lca drive for next cycle
%  Note: nice comment, not my style
emit pLcaDrives;
await tick;
% setup data path from pam to host
%  Note: this, too
emit pPamDrives;
... ... ...
emit pHostDrives;
... ... ...
```

Figure 14: Fragment of tcint.strl: DMAWRITE

According to Esterel semantics, everything is clear; the signal is ignored, and the states following it are "blue." But according to the comment, maybe the following states should be "red." Or maybe "both" states have to be active, which is obviously not possible using the existing DRIVE FSM.

4.4 VHDL-to-Esterel: A Bad Idea

Going back to the "memory-controller.strl" sample, we noticed that it is not sequentially equivalent to the original VHDL design. At some point, an extra cycle was added, which was sufficient to render the circuit useless.

We rewrote it from scratch, ignoring both the existing sample and the original VHDL code.

This gave better results (Table 5), even without sequential optimization. The author of the original Esterel code had no idea of what the circuit was supposed to do. On the other hand, we knew the desired functionality very precisely, and we trust our Esterel code more than we trust our original VHDL, as is is shorter and more expressive.

Notice that our redesign does not depend much on optimization; even a simple combinational script gives nice results.

4.5 HDL Abstraction

As Esterel is deterministic, any semantically correct code leads to an exact RTL description. So there is theoretically an optimum (good) implementation which is sequentially equivalent to the original. However, the original is not the source code, but the model inside designer's mind.

Each HDL presents its own model of computation. If one formulates the problem in terms of that model, the code is likely both correct and efficient. Otherwise, the code may lead to incorrect circuits even more easily than to inefficient ones. We consider this issue critical.

Reality shows that very popular HDLs (such as Verilog, VHDL, or—in the software world—C) are a scandal from the theoretical point of view; they lack even a deterministic semantics. This is highly regrettable; the fact that they are so widespread, despite of this major drawback, implies that the HDL success depends more on human than mathematical issues. Ignoring the former but concentrating exclusively on the later will not provide viable real-world solutions. We strongly believe that the CAD community has to seriously consider this.

5 Conclusions

5.1 Compiler-flavored Optimizations

Small local optimizations are possible in many cases. Most of them are trivial (and a lot of such techniques are already employed in CEC) but a handful of more advanced ones can be implemented.

Even if such optimizations usually cannot solve the "interesting" problems, they can safely be done in a greedy manner, and—given the large number of opportunities to apply them—the burden on the expensive steps that may follow significantly decreases.

	src(bytes)		# levels	# LUTs	# latches
Third-party Esterel	80	v5 + seq	3	24	16
		auto: no seq	3	52	17
		auto: seq	3	27	15
		manual enc	2	31	13
Our Esterel	36	v5 + seq	2	17	8
		auto: no seq	2	23	9
		auto: seq	2	18	8
		manual enc	2	14	3

Table 5: memory-controller.strl: comparison of two versions

Moreover, it is more likely that the main algorithm will perform better on a "noiseless" input, which exposes the real design challenges.

It is unfair to consider these optimizations as an optional preprocessing step; they have to be performed as aggressively as possible, given that they do not compromise the potential for further optimization.

5.2 Exploiting Sequential DCs

The main conclusion drawn from the samples we analyzed is that a directly translated Esterel specification leads to a circuit with a lot of sequential redundancy. Many state codes are unreachable, many states are equivalent, and critical paths span several hierarchical modules and include false paths.

However, this is not necessarily due to poor design. As Esterel has a different abstraction model than VHDL, it may be unfair to ask the designer to write Esterel with the structure of the resulting network in mind. On the contrary, the idea is to let the designer ignore these details and let him or her focus on describing the circuit in a coherent, natural way.

Optimizing circuits without sequential analysis has little chance of succeeding. We had hoped that important sequential redundancy would be localized, but instead most sequential redundancy and critical paths cross hierarchical boundaries. Because of this, analysis and optimization process has to cross such boundaries, too. Effective state assignment cannot therefore rely solely on high level information.

On the other hand, running brute force sequential optimization on the flattened network is not feasible, and may generate very suboptimal circuits for large samples. General techniques such as circuit partitioning or approximate reachability are present in current tools, but they are based on heuristics. We may not want to discard the high level information; even if it alone is not sufficient to generate a good circuit, it may be a very precious resource to exploit further.

5.3 Future directions

The critical consequence is that Esterel synthesis can not be easily split into two separate steps: high and low level. It would be very attractive to simplify the problem in this way; moreover, a constructive approach for the high-level layer would make the synthesis scale very well.

This may be feasible for large circuits where big modules can be optimized independently. But on the scale of our examples, the designs proved to be more tightly connected than expected.

We expect that on the mentioned medium scale the main computational burden will remain on expensive sequential algorithms, which can be seen as search programs. Given the performance of today's computers and the advances in dealing with specific intractable problems, this kind of algorithms perform pretty well on medium-sized and well defined problems, but they completely lack a "horizon," i.e., a pertinent view of the global problem.

The challenge would be to use the high-level structure to drive these algorithms, feeding them with moderate-sized data and asking for results that are relevant to improving overall circuit performance.

References

[1] Gérard Berry. A hardware implementation of pure Esterel. In *Proceedings of the International Workshop on Formal Methods in VLSI Design*, Miami, Florida, January 1991.

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

[3] Stephen 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.

[4] Klaus Schneider. A verified hardware synthesis for Esterel programs. In *Proceedings of the International IFIP Workshop on Distributed and Parallel Embedded Systems (DIPES)*, Paderborn, Germany, 2000.

[5] Cristian Soviani, Jia Zeng, and Stephen A. Edwards. Improved controller synthesis from esterel. Technical Report CUCS–015–04, Columbia University, Department of Computer Science, New York, NY, 2004.

[6] Horia Toma, Ellen Sentovich, and Gérard 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, November 1996.