# Improved Controller Synthesis from Esterel

Cristian Soviani Jia Zeng Stephen A. Edwards*
Department of Computer Science, Columbia University
1214 Amsterdam Avenue, New York, New York, 10027
{soviani, jia, sedwards}@cs.columbia.edu

## Abstract

*We present a new procedure for automatically synthesizing controllers from high-level Esterel specifications. Unlike existing* RTL *synthesis approaches, this approach frees the designer from tedious bit-level state encoding and certain types of inter-machine communication.*

*Experimental results suggest that even with a fairly primitive state assignment heuristic, our compiler consistently produces smaller, slightly faster circuits that the existing Esterel compiler. We mainly attribute this to a different style of distributing state bits throughout the circuit.*

*Initial results are encouraging, but some hand-optimized encodings suggest room for a better state assignment algorithm. We are confident that such improvements will make our technique even more practical.*

## 1 Introduction

Designing state machine controllers remains one of the more challenging aspects of digital circuit design. Although such controllers tend to contain fewer gates than datapaths or memory, the irregularity and concurrency of controllers makes their design and verification time-consuming.

Currently, controllers are usually written in register-transfer-level VHDL or Verilog and synthesized automatically. While such RTL synthesis frees the designer from detailed logic design, he or she is still responsible for encoding, communication, and composition.

Like the move from schematic capture systems to RTL design, we need to raise the level of abstraction in designing controllers. This paper describes a way to do this through an efficient compiler for a higher-level language well-suited for describing complex synchronous digital controllers.

We present novel techniques for compiling Berry's Esterel language [2] into synchronous digital circuits. Esterel was not originally intended for hardware specification, but its synchronous, concurrent, deterministic semantics plus high-level control constructs make it adept at describing control-intensive synchronous digital logic circuits.

While resembling the problem of generating circuits from Verilog or VHDL RTL descriptions, generating circuits from Esterel differs in two main ways. First, in addition to "combinational" statements such as conditionals and assignments, which execute within a single clock cycle, Esterel supports implicit state machines through the "sequential" *pause* statement, which delays for a cycle. Thus, Esterel program counters hold their state between clock cycles and are in this sense more like behavioral descriptions. Unlike RTL, therefore, the synthesis system is responsible for state encoding. Second, Esterel supports high-level control constructs such as concurrent composition, preemption, and exceptions. Both aspects make Esterel a challenging language to translate into circuitry, but also enable more aggressive optimizations because the compiler is able to gain a better understanding of the program's behavior.

Berry first described the translation of Esterel into circuitry in 1992 [1] and little has changed since. Touati, Berry, Toma, and Sentovich [12, 11, 9] improved it by reduce the number of latches produced by Berry's mechanical translation procedure. They compute the reachable state set implicitly using BDDs, then use this knowledge to remove sequential redundancies. This improves circuits because the group-hot encoding used by Berry's synthesis procedure is fairly inefficient. Unfortunately, computing the reachable state becomes prohibitively expensive for large examples.

Potop-Butucaru's [6, 7] optimizations, while intended for software, apparently also improve circuit generation, but we are unaware of any published results confirming this.

In this paper, we present a new circuit synthesis procedure for Esterel that improves upon the current technique in three novel ways. First, a new state encoding technique is employed. Second, the "scaffolding" logic around the state machines is synthesized in a different way. And finally, a class of combinational redundancies is removed cheaply by synthesizing the circuit from a control-dependence graph, not a control-flow graph. One of the authors (Edwards) presented some of these ideas earlier [5].

*Edwards and his group are supported by an NSF career award, a grant from Intel corporation, and an award from the SRC.

## 2 An Esterel Example

Figure 1a is a simple Esterel program that will illustrate our compiler's operation. It consists of a pair of state machines with a global reset input R and four one-bit outputs, A, B, C, and D. The first machine, enclosed in the *loop* statement, makes signals A and B true in alternate cycles. The combinational *emit* statement sets its signal true in the cycle in which it executes; the sequential *pause* statement delays for a cycle before passing control to the statement following it.

The second machine (starting at *emit C*) makes signal C true in the first cycle, D true in the third, and then terminates. The double vertical bars between the two machines makes them start and run concurrently in lockstep.

The *every* statement surrounding the two machines restarts them every time the R signal is present. *Every* provides strong preemption, i.e., in a cycle in which R is true, the machines are restarted before they have a chance to do whatever they would otherwise do if R were false.

Figure 2 shows a representative timing diagram for the program. In the last cycle, D would have been true had R not reset the machines.

## 3 Intermediate Representations

CEC employs two intermediate representations, the first generated from a traditional abstract syntax tree. The first IR is a variant of Potop-Butucaru's GRC format [6, 7], which consists of a selection tree (Figure 1b) that represents the hierarchical state structure of the program, and an acyclic control-flow graph (Figure 1c) that represents the behavior of the program within a cycle. Ultimately, the control-flow graph represents the combinational portion of the program; the selection tree the sequential portion.

The second IR transforms GRC's control-flow graph into a control-dependence graph (Figure 1d) that exposes additional concurrency. In circuitry, the CFG-to-CDG transformation corresponds to removing redundant gates.

From the abstract syntax tree, the compiler builds the selection tree in Figure 1b and the control-flow graph in Figure 1c using the recursive procedure described by Potop [6, 7]. In Figure 1b, the double diamonds—*switch* nodes—represent exclusive states, the triangles—*fork* nodes—concurrent states, and the others simple states. The two children under *switch* node 5 correspond to the two states of the first machine: branch 0 emits A, and branch 1 emits B. The three children under *switch* node 8 are the three states of the second machine: branch 2 waits for a cycle after C was emitted, branch 1 emits D, and branch 0 corresponds to the machine being terminated. And the three children under *switch* node 0 represent the initialization of the machine, its behavior in the first cycle, and its behavior in later cycles.

The GRC control-flow graph is a flowchart with some additions. It is concurrent: when a triangle-shaped *fork* node is executed, it sends control to all of its successors. Once

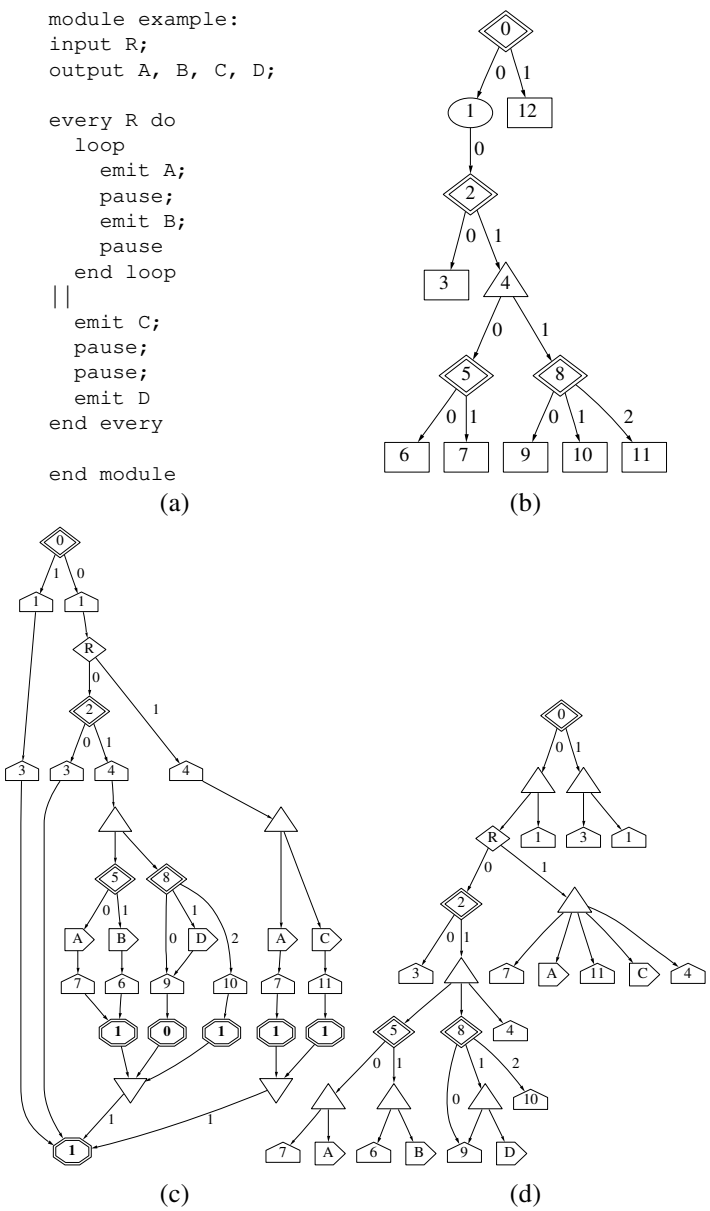


**Figure 1. (a) A simple Esterel program, (b) its selection tree, (c) its control-flow graph, and (d) its (simplified) control-dependence graph.**

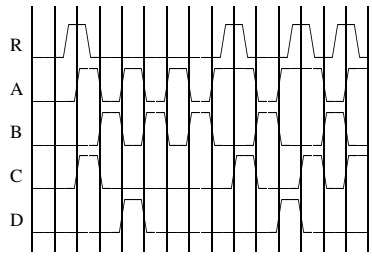


**Figure 2. A timing diagram for Figure 1a**

spawned, concurrent threads of control are synchronized by *sync* nodes (the downward-pointing triangles).

Although not in this example, *sync* nodes generally have multiple successors and handle terminating threads and concurrently-thrown exceptions. When control in all incoming threads reaches *sync* node, the node passes control to its successor labeled with the highest exit level produced by its threads. This example only uses levels 0 ("thread terminated") and 1 ("thread paused"); exceptions are encoded using higher exit levels.

The state of the program between cycles can be thought of as residing in the *switch* nodes in the selection tree. Each remembers which of its children was last *enter*ed, and when control reaches a *switch* node in the control-flow graph, it passes control to one of its children depending on this state. For example, when the house-shaped *enter* 1 node in Figure 1c is executed, it sets the state of *switch* 0 in Figure 1b to 0 because node 1 is child 0 of node 0. After this, when control reaches *switch* 0 in Figure 1c, the node sends control down the arc labeled 0.

Consider what happens when the program starts executing. By convention, *switch* 0 is initially set to 1, so control initially flows down the arc labeled 1 from *switch* 0 to a fork, which sends control to the *enter* 1 and *enter* 3 nodes, setting *switch* 0 to 0 and *switch* 2 to 0.

In the next cycle, control flows down the arc labeled 0 under *switch* 0 to the test of R (the diamond). If R is absent, control flows through *switch* 2 to *enter* 3 and the program stays in the same state. Otherwise, control flows through *enter* 4 setting the state of *switch* 2 to 1, hits a fork, and executes *emit* C and *emit* A before hitting *enter* 7, setting the state of *switch* 5 to 1, and enter 11, setting *switch* 8 to 2.

While it would be practical to generate a circuit directly from the control-flow graph of Figure 1c (Berry's V5 compiler does essentially this [1]), to further improve the generated circuit our compiler transforms the control-flow graph into the control-dependence graph of Figure 1d using the efficient algorithm of Cytron et al. [3]. This transformation exposes additional parallelism in the graph by adding fork nodes and grouping control-equivalent nodes under them. For example, the *emit* A and *enter* 7 nodes in Figure 1c are always executed together. The control-dependence algorithm recognizes this and always pairs these two nodes under a single *fork* nodes in Figure 1d.

Some redundant nodes were removed in the translation from Figure 1c to Figure 1d. The *sync* nodes only have a single successor and therefore do not do anything. This means the *terminate* nodes (the double octagons) are also unneeded and were removed.

Our GRC representation differs from Potop's [6, 7] in one key way: handling of the terminated state of a thread. This occurs when a thread in a group has terminated while the others continue running. Potop has an additional type of node in the control-flow graph: *exit*, which is the opposite of an enter, i.e., it marks its thread as not operating; it marks a *switch* as passing control to none of its children. We feel, and experiments bear this out, that the terminated state of a thread should be treated like any other state to minimize the number of special cases the state machine synthesis procedures must address and potentially simplify the state encoding problem. In fact, experiments suggest that instead the initial state of a machine should be treated specially.

Other other changes to the GRC representation are cosmetic. Potop draws *switch* nodes in the control-flow graph as triangles, not diamonds (we prefer the diamond because it is a decision point). His parallel synchronizer nodes distinguish their inputs; ours have equivalent inputs and distinguish exit levels using *terminate* nodes instead.

## 4 Generating Circuits

Much as the GRC format consists of the sequential selection tree and the combinational control-flow graph, our compiler synthesizes circuits in two parts: sequential state machines with encoding and decoding logic, and "scaffolding" logic that implements the control flow graph and handles inter-machine communication. Separating the two simplifies the task of analyzing the impact of state encoding.

Each exclusive node in the selection tree becomes a local state machine; the high-level structure of the specification is preserved. The output of each machine is a set of one-hot encoded *chk* signals, one per state. This is the most natural interface for the logic at the *switch* nodes in the control-flow graph, the only observers of the machines' outputs.

Each machine has a *goto* input for each of its states plus a *hold* input that instructs the machine to hold its state. The machine assumes the *goto* and *hold* inputs are mutually exclusive, i.e., at most one is active in any cycle. To ensure this, the circuit outside the machine often includes arbitration logic. Making it the responsibility of the machine's environment is natural because the appropriate priorities come from structure of the control-flow graph.

The machines' *hold* inputs implement Esterel's *suspend* instruction, which can temporarily halt a group of state machines. The *hold* inputs of the machines within the scope of a *suspend* instruction (i.e., the exclusive nodes under the *suspend* node in the selection tree) are driven with the activation net of a *suspend* node in the control-flow graph. Like the *goto* inputs, external arbitration logic ensures *hold* and *goto* inputs are never active simultaneously.

For each machine, our compiler builds two-level encoding/decoding logic given either the default or a user-supplied encoding. The default encoding is one-hot for machines with greater than three states; for machines with three or fewer states, one state is encoded as all 0's. Surprisingly, this simple variation on one-hot produces a significant improvement in the quality of the generated circuits.

Our encoding scheme differs from the traditional V5 encoding. In V5, each active state of a concurrently-running thread is given a one-hot code, but the terminated state of a thread, which may appear when other threads within the same group continue to run, is encoded with all zeros. Detecting this state can require a wide OR gate. By contrast, our compiler treats this "terminated" state as yet another state if it may occur, possibly saving the wide OR gate at the expense of an additional latch. In any case, our flexible encoding technique allows the terminated state to be encoded as in V5, or in whatever other style proves suitable.

Our technique builds state machines throughout the selection tree. By contrast, V5 only holds state at the leaves. While the V5 encoding is sufficient (the state of any machine can be computed by logically ORing the state of its children), it can lead to inefficiencies. Consider the signal that indicates an abort condition should be tested. In V5, this is computed by ORing together all the flip-flops "under" the abort test. In our technique, this is usually a *chk* signal from a machine above the abort statement in the selection tree. This generally eliminates levels of logic and may make computing the effect of abort statements faster. In some sense, ours is a retiming of the V5 scheme, but is more flexible because it also supports different encodings.

Thew circuity between state machines is nearly a one-to-one translation of the control-flow graph. Each arc in the CFG becomes a net that is true if the corresponding arc executes in the current cycle. The circuit driving each net is usually trivial. An *emit* ORs its activation net with all the others that emit the same signal, a conditional test generates a pair of AND gates driven by the test's predicate and its complement, and a fork node becomes simply fanout.

The translation for the sync node is more complicated: a non-trivial one becomes a priority encoder that computes the maximum exit level for the group of threads.

Nodes that interact with state machines are more complicated. *Switch* nodes AND their activation net with the *chk* nets from their machine. Nodes for *enter* and *suspend* either drive machines' *goto* or *hold* inputs directly, or may feed into a arbitration circuit that ensures the *goto* nets remain mutually exclusive. Such logic is often necessary because of Esterel's exception-handling constructs and its strange reincarnation phenomenon that makes some statements execute twice or more every cycle.

CEC generated Figure 3 from the control-dependence graph in Figure 1d. The inputs are the R signal and *chk* signals from the four state machines. The outputs are signals A, B, C, and D plus the *goto* outputs for the state machines. There are no *hold* outputs because the example has no *suspend* statements.

The leftmost pair of AND gates in Figure 3 implement the test for R under branch 0 of *switch* 0, and the leftmost OR implement the two *enter* 1 nodes. At the second level, the

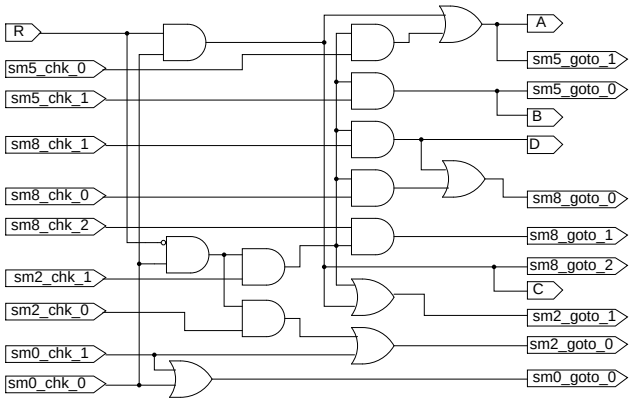


**Figure 3. The scaffolding circuit generated for the example in Figure 1.**

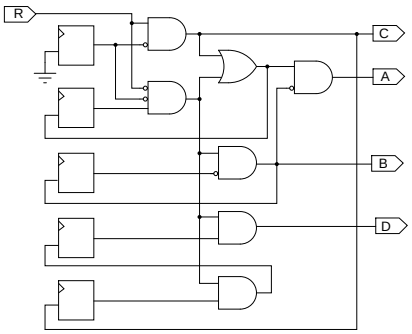


**Figure 4. The final SIS-optimized circuit.**

AND gates implement *switch* 2. The AND gates at the third level implement *switch*es 5 and 6 and the OR gates implement the four *enter* 3 and 4 nodes. The topmost OR gate on the right implements the two *emit* A nodes, and the second OR combines the two incoming control arcs on *enter* 9.

Figure 4 shows the circuit after adding state machines and optimizing with SIS. From the top, the five latches in Figure 4 encode *switch* 0, *switch* 2, *switch* 5, and the second and first branches of *switch* 8.

## 5 Counters

Counting events is easy in Esterel. For example, the statement *await 3 S* waits for the three cycles in which signal S is true. Both V5 and our compiler counts such events with explicit counter state machines. Although counting could be treated as just another source of states (e.g., by dismantling such an *await* statement into three), treating them as counters simplifies counter-specific optimizations.

CEC synthesizes simple binary down-counters to count events. Each has a *start* input that loads an initial value, a *enable* input that causes the counter to counter, and an *zero* output that indicates when the count has expired.

**Table 1. Experimental Results**

| Example | SIS | | | | | | | | | Xilinx | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Literals | | | Latches | | | Levels | | | Slices | | | Period (ns) | | |
| | V5 | CEC | hand | V5 | CEC | hand | V5 | CEC | hand | V5 | CEC | hand | V5 | CEC | hand |
| Figure 1a | 23 | 15 | 15 | 6 (0) | 5 | 5 | 4 | 3 | 3 | 7 | 4 | 4 | 4.7 | 4.6 | 4.4 |
| dacexample | 41 | 23 | 22 | 7 (0) | 5 | 5 | 5 | 3 | 3 | 10 | 5 | 5 | 6.2 | 6.0 | 5.5 |
| jacky1 | 39 | 22 | 20 | 5 (0) | 4 | 4 | 4 | 3 | 3 | 6 | 5 | 4 | 5.4 | 6.1 | 5.0 |
| runner | 218 | 145 | 144 | 30 (24) | 20 | 20 | 11 | 10 | 10 | 56 | 36 | 35 | 10.6 | 8.4 | 8.1 |
| greycounter | 240 | 173 | 142 | 34 (6) | 18 | 15 | 11 | 13 | 9 | 40 | 34 | 17 | 12.4 | 13.4 | 8.9 |
| scheduler | 519 | 380 | | 74 (52) | 55 | | 8 | 8 | | 80 | 66 | | 11.3 | 8.9 | |
| servos | 407 | 287 | | 60 (16) | 47 | | 10 | 10 | | 105 | 66 | | 16.7 | 13.4 | |
| abcd | 167 | 165 | | 17 (0) | 13 | | 7 | 8 | | 43 | 43 | | 12.8 | 12.5 | |
| tcint | 508 | 414 | | 95 (14) | 60 | | 17 | 9 | | 115 | 81 | | 10.8 | 10.9 | |

Reported by SIS: **Literals**: number of literals after optimization by "script.rugged." **Latches**: number of latches (number due to counters). **Levels**: levels of logic (complex gates). Reported by Xilinx tools: **Slices**: Number of slices (area). **Period**: Minimum clock period in nanoseconds.

The V5 compiler synthesizes faster, more complex counters with two latches per bit. Although this may produce faster unoptimized circuits, the increased circuit complexity appears to make it harder for the logic optimization step.

Currently, CEC does technology-independent synthesis, but Esterel's counters illustrate the need for being more technology-aware. Especially on the Xilinx architecture, which has highly optimized circuitry for arithmetic carries, binary counters can be implemented very efficiently. But other counter styles may be superior in other technologies.

## 6 Experimental Results

Table 1 shows our experimental results. We compare the output of CEC running normally with the output of Berry et al.'s V5 compiler and a circuit generated with CEC using hand-optimized state machine encodings. The information in columns titled "SIS" are statistics from running the SIS logic optimizer [8]. Data in columns titled "Xilinx" were collected after running the Xilinx ISE tools.

We used SIS's combinational optimization script *script.rugged*. Although targeted only at area optimization, it is standard and scales well. SIS can also do sequential optimization, but it requires the ability to calculate the reachable state set, which rapidly becomes impractical for larger examples. Our goal is to synthesize arbitrarily large controllers, so we chose not to use other optimizations.

For the Xilinx results, we transformed the optimized circuit from SIS into structural Verilog using a simple script and passed it to the Xilinx ISE tools (i.e., xst, ngdbuild, map, par, trce) with all optimization flags set to maximum and a Spartan 2s50e-ft265-6 target. We registered the inputs and outputs for reasonable I/O timings. The number of slices and minimum clock period come from a placed and routed design, so we consider them realistic. A slice is a pair of four-input lookup tables driving two flip-flops.

We ran our compiler on controllers ranging from the small example in Figure 1a to a bus controller (tcint). Dacexample is the simple arbiter from Edwards [4]. Jacky1 is a pattern matcher due to Potop-Butucaru [6]. Runner is Berry's example from the Esterel tutorial. Greycounter is a four-bit grey-code counter with an alarm. Scheduler models a round-robin arbiter responding to requests to a shared resource. Servos is a controller for a trio of stepper motors that rotates them into a home position then to a given position, all the while watching for faults. Abcd is a four-button user interface that locks out other buttons while one is active pressed. Tcint is a controller for the turbochannel bus.

In general, CEC generates circuits with fewer latches for two reasons. Its counters use half as many latches as those in V5, but this only affects certain examples. The numbers in parentheses in Table 1 are the number of latches in the V5 circuit due to counters. So the reduction in latches in the scheduler, runner, and servos examples are due exclusively to counters, but this is not true for the others.

Our encoding of machines with three or fewer states causes most of the different in latch counts. This causes the differences in greycounter and tcint, which has many two-state machines. Applying Potop-Butucaru's optimizations to the V5 synthesis procedure may produce similar improvements, but we know of no published comparison.

CEC generates smaller machines that run at comparable speeds. The area improves approximately 20% on many examples. We attribute this to the more compact state encoding and the control dependence analysis, which removes certain redundant gates more effectively than SIS.

Examples with more extensive preemption constructs (e.g., servos, scheduler, and runner) run faster when synthesized with CEC. We attribute this to our distributed style of state assignment, i.e., because we synthesize latches at

every *switch* node in the selection tree, not just those at the leaves as V5 does.

For the smaller examples, we hand-optimized the state encoding, largely by trial-and-error, to see how much better we could make the circuits. Not surprisingly, the biggest advantage came in the greycounter example, which became significantly smaller and faster; the others improved less dramatically. We attribute this to the fairly regular character of the greycounter example.

We included the hand-optimized results partially to show the power of CEC's user-supplied encoding mechanism, but also to suggest the possibility of improvement in the current heuristics for state assignment.

## 7 Conclusions and Future Work

We presented a novel procedure for synthesizing controller circuits from Esterel programs. Experiments suggest our technique gives reduced circuit size with a slight speed advantage over the procedure in Berry et al.'s Esterel V5 compiler. Our procedure uses the GRC intermediate representation suggested by Potop-Butucaru [6, 7] and a redundancy-removing control dependence transformation. Together, these give smaller, more distributed controllers with less communication.

Our speed results are less impressive than those for area, but they no worse than the existing V5 technique. We will concentrate on circuit speed in the future.

One drawback of the recursive GRC generation procedure described by Potop [7] is its excessive duplication of structures (e.g., there are two identical emissions of A in Figure 1c that could be merged). While Esterel's reincarnation semantics often do require structures to be duplicated, and the SIS logic optimizer is often able to identify and collapse such redundancy, the existing translation makes many needless duplications that should be avoided. Tardieu and de Simone [10] have devised some simple, powerful static analysis for identifying when duplication is necessary. We will incorporate this algorithm in a future version of the compiler to reduce the size of the generated circuit.

Much work remains to be done. We plan to explore more technology-dependent optimizations, such as different counter architectures for different technologies. Also, we currently only synthesize the pure subset of Esterel. While this is the most natural one for controllers, it should be possible to add limited datapath synthesis ability to CEC. Finally, we plan to explore more powerful state assignment algorithms that take into account the structure of the scaffolding logic. Because we know the environment in which these machines will execute, ours will be significantly different than existing state assignment algorithms.


## Acknowledgements

Intel's Mike Kishinevsky has long demanded better state encoding. Xilinx's Satnam Singh gave us real chips and development tools. Dumitru Potop-Butucaru gave us his GRC format. Gérard Berry has always helped.


## References


[1] 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.

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

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[4] S. A. Edwards. Compiling Esterel into sequential code. In *Proceedings of the 37th Design Automation Conference*, pages 322–327, Los Angeles, California, June 2000. Association for Computing Machinery.

[5] 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.

[6] D. Potop-Butucaru. *Optimizing for Faster Simulation of Esterel Programs*. PhD thesis, INRIA, Sophia-Antipolis, France, Aug. 2002.

[7] D. Potop-Butucaru. Optimizations for faster execution of Esterel programs. In *Proceedings of Memocode*, pages 227–236, Mont St. Michel, France, June 2003.

[8] 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.

[9] E. M. Sentovich, H. Toma, and G. Berry. Efficient latch optimization using exclusive sets. In *Proceedings of the 34th Design Automation Conference*, pages 8–11, Anaheim, California, June 1997.

[10] O. Tardieu and R. de Simone. Instantaneous termination in pure esterel. In *Proceedings of the 10th Annual Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 91–108, San Diego, California, June 2003.

[11] H. Toma, E. Sentovich, 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.

[12] 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.