An Esterel Compiler for Large Control-Dominated Systems

Stephen A. Edwards, *Member, IEEE*

***Abstract*—Embedded hard real-time software systems often need fine-grained parallelism and precise control of timing, things typical real-time operating systems do not provide. The Esterel language has both, but compiling large Esterel programs has been challenging, producing either needlessly slow or large code. This paper presents the first Esterel compiler able to compile large Esterel programs into fast, small code. By choosing a concurrent control-flow graph (CCFG) as its intermediate representation, it preserves many of the control constructs to produce code that can be 100 times faster and half the size than code from other compilers with similar capacity. The primary contribution is an algorithm that generates efficient sequential code from a CCFG. While developed specifically for compiling Esterel, the algorithm could be used to compile other synchronous languages with fine-grained parallelism.**

***Index Terms*—Code generation, compilers, concurrency, embedded systems, Esterel, reactive, real-time language, synchronous.**

I. Introduction

MANY applications in reactive real-time embedded systems are most naturally described as concurrent systems, yet many are implemented using sequential languages like C or assembly on sequential processors. A real-time operating system (RTOS) capable of scheduling the execution of multiple independent sequential processes is a common way of providing concurrency to such languages, but the behavior of such an RTOS can be unpredictable, making it difficult to guarantee precise system timing.

The synchronous approach [1] provides precise timing control by operating a system in lockstep with a global periodic clock signal. Although timing within a particular clock cycle is essentially uncontrolled, a system has exact control over the clock cycle in which each event occurs.

The synchronous approach is natural in hardware, where global clocks often drive sequential elements, but is used less frequently in software. A few synchronous languages for software have been proposed (Esterel [2] and Lustre [3]), but they have proven challenging to compile.

Implementing a concurrent synchronous language such as Esterel using concurrency supplied by an operating system would be very inefficient because of the large number of threads in a typical Esterel program (thousands in large programs) and the need for synchronization within each cycle. Since each thread needs to run at least once every cycle, and possibly much more depending on communication, context switching cost could dominate a single-processor implementation. A multiple-processor implementation, such as the one proposed by Caspi *et al.* [4], trades some context switching overhead for communication and synchronization overhead.

```
module Example:

input  S, I;
output O;

signal R, A in
  every S do
    await I;
    weak abort
      sustain R
    when immediate A;
    emit O
  ||
    loop
      pause; pause;
      present R then emit A end;
    end
  end
end

end module
```

Fig. 1. A simple Esterel module modeling a shared resource. The first thread generates requests (R) in response to external requests (I), and the second thread responds to them (A) in alternate cycles. The S input resets both threads.

This paper describes EC, a new compiler for Esterel that can produce small, fast code for large programs, avoiding the shortcomings of earlier compilers, and making large synchronous specifications practical. An executable produced by EC can run 100 times faster than one from another compiler able to handle large programs and can be exponentially smaller than code from the compiler that produces the fastest known code (EC's code is about half as fast).

This paper is structured around the small Esterel program in Fig. 1. In Section II, we introduce the Esterel language and explain the behavior of the example. Section III is a review of existing Esterel compilers. Section IV describes EC's input format—the intermediate representation IC—and the example's manifestation in it (Fig. 3).

The bulk of the paper describes the new compiler. Section V begins with an overview of the three compilation steps: constructing a concurrent control-flow graph (CCFG) from Esterel, scheduling the CCFG, and finally synthesizing a sequential control-flow graph (SCFG) from the concurrent one. Section VI describes the recursive unrolling algorithm that translates an In-

Manuscript received February 29, 2000; revised May 23, 2001. This paper was recommended by Associate Editor R. Gupta.

The author was with Synopsys, Inc., Mountain View, CA USA. He is now with the Department of Computer Science, Columbia University, New York, NY 10027 USA (e-mail: sedwards@cs.columbia.edu).

Publisher Item Identifier S 0278-0070(02)01046-1.

termediate Code (IC) graph into an equivalent CCFG. The main issues here are adhering to IC's complicated semantics and correctly unrolling the graph when certain instructions may execute more than once in a cycle. Section VII describes how a SCFG is synthesized from the concurrent one using a simulation procedure. After scheduling the nodes in the graph, the procedure steps through each one, copying it to the sequential graph and inserting code that simulates the effects of a context switch when it encounters a node from a different thread. Finally, Section VIII shows how to generate attractive C code from the SCFG. Fig. 17 shows the C code EC generates for the example.

Section IX describes experiments that compare the quality of the code EC generates with that from other Esterel compilers. The paper concludes with suggestions about how to extend the work.

II. The Esterel Language

Intended for specifying reactive real-time systems, Esterel [2] has the control constructs of an imperative language like C but includes concurrency, preemption, and a synchronous model of time like that used in synchronous digital circuits. In each clock cycle, the program resumes running its concurrent threads, reads its inputs, computes its reaction, and suspends until the next cycle.

An Esterel program communicates through signals that are either present or absent in each cycle. In each cycle, each signal is absent unless an *emit* statement for the signal runs. Conditional *present* statements test signals and perform different actions depending on the presence of a signal. The presence of an emitted signal is seen immediately in the cycle it was emitted and does not hold its value in later cycles.

In Esterel, a statement that tests the value of a signal in a clock cycle blocks until the presence or absence of the signal is established in that clock cycle. Put another way, any statement that emits a signal must run before any statement that reads it. This constrains the order in which statements running in concurrent threads may execute within a cycle and can lead to deadlock. For example, code that attempts to test a signal before emitting it is erroneous: the data dependency contradicts the control dependency.

EC uses a simpleminded structural check to detect deadlock conditions and reject programs that contain them, but this can reject useful programs. Unfortunately, the alternative, used in the V5 compiler described in Section III, requires exploring the state space of the program (currently only practical with symbolic methods that are still quite expensive) and resynthesizing the program. EC's greatest shortcoming is its inability to do this analysis and accept a larger class of programs; we discuss how this problem might be addressed at the end of the paper.

A. An Example

Fig. 1 shows a simple Esterel program with two concurrent threads. Meant to model an arbiter for a shared resource, the first thread passes requests from the environment to the second thread, which responds to requests. The first thread waits for an I signal before holding the R signal present until the A signal

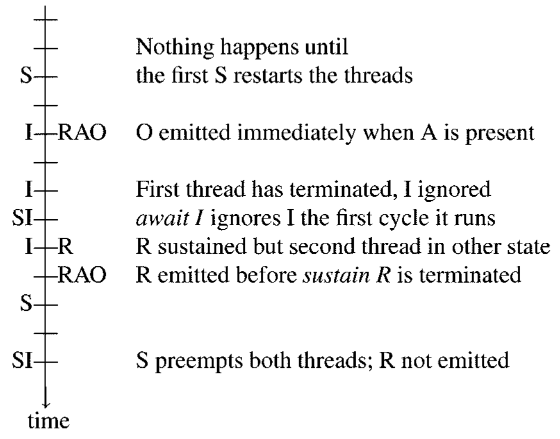

S
I
RAO
I
SI
I
R
RAO
S
SI
time
Nothing happens until
the first S restarts the threads
O emitted immediately when A is present
First thread has terminated, I ignored
await I ignores I the first cycle it runs
R sustained but second thread in other state
R emitted before *sustain R* is terminated
S preempts both threads; R not emitted

Fig. 2. A timing diagram for an execution of the Esterel program in Fig. 1. Inputs are listed on the left; the signals generated by the program in response are listed on the right. Each horizontal tick denotes a clock cycle.

arrives, at which point it emits the O signal and terminates. The second thread emits R in response to A in alternate cycles. Fig. 2 illustrates this behavior with a depiction of the program's response for a particular input sequence.

The body of the first thread starts with an *await* statement that waits for the next cycle in which its signal is present. *Await* always waits one cycle before it can terminate, so this thread does nothing in the cycle S when it first appears. Following the *await* is a *weak abort*. When this runs (i.e., in the cycle in which signal I is present), it immediately starts its body (in this case, a *sustain* statement that makes signal R present in every cycle until the statement is terminated) and watches for the A signal. Because this is a weak *abort*, the *sustain* runs in the cycle when A is present, but terminates. Moreover, since the predicate is "immediate A," the A signal is checked in the first cycle the *abort* statement runs (*abort* normally starts checking the next cycle, just like *await*).

The second thread is an infinite loop. Each *pause* statement delays a cycle when control reaches it. This happens in the first and second cycle in which the loop runs. In the third, signal R is checked, and its presence causes A to be emitted. After this, the loop is immediately restarted and the first *pause* is executed again.

The two threads are enclosed in an *every-do* loop that can preempt the two threads. The body of the statement (the pair of threads) is restarted in the cycle in which the signal S is present before the body has a chance to resume.

There is a subtle and often critical difference between the strong preemption of the *every-do*, which checks its predicate before its body resumes, and the weak preemption of *weak abort-when*, which checks its predicate after its body resumes. We used weak abortion to preempt *sustain R* because strong abortion would have caused a deadlock: emitting R can cause A to be emitted. Had this been strong preemption, the presence of A would have prevented R from being emitted—a contradiction. Strong preemption around the two threads does not cause deadlock. It makes the program ignore I in cycles when S is present.

III. Related Work

Three other techniques for compiling Esterel are based on automata, logic gates, and event graphs.

The earliest compilers, such as Berry *et al.*'s V3 [2], built a single automaton for an Esterel program through exhaustive simulation. Each state of this automaton corresponds to set of control points from which the Esterel program will resume in the next cycle. The code for each state is a tree containing actions (such as emitting signals), conditionals that test external signals, and leaves that set the next state.

Automata compilers produce very fast code, but it can be exponentially larger than the source program since they generate separate code for each possible state of the program. Two approaches have been proposed to reduce this code size. The Polis group's automata compiler [5], [6] uses a binary decision diagram to identify code that can be shared between states. Castelluccia *et al.* [7] also share subtrees to reduce code size, but also attempt to improve code speed by inlining called functions, swapping *then* and *else* branches to improve branch prediction, moving away infrequently executed code to improve cache performance, and reordering tests to put the common cases first. While both techniques can significantly improve the quality of generated code, they are still limited to small programs where it is practical to enumerate the states.

EC-generated code is slower than automata code because it has more overhead (due to internal communication and context switches), but EC code can be exponentially smaller because code is duplicated far less. As a result, EC is superior for all but the smallest Esterel programs.

The second class of compilation technique, exemplified by Berry *et al.*'s V5 compiler, translates Esterel into a netlist of Boolean logic gates and then generates a levelized compiled-code simulator for it. Capacity is the main advantage of this approach. Unlike automata compilers, each Esterel source instruction is translated into a small group of instructions in the executable; nothing is duplicated. The disadvantage is slower code; because the generated code is forced to perform computation for idle portions of the program, it can run hundreds of times slower than an automaton implementation of the same program.

Compared to gate-based compilers, EC generates code that is slightly more compact, but because it does not need to do work for idle portions of the program, the code can run hundreds of times faster. The key advantage of gate-based compilers over EC is their ability to analyze and compile programs that appear to have cyclic dependencies. However, this is a costly procedure that requires symbolic state-space exploration and circuit resynthesis [8].

The third approach, pioneered by a group from France Telecom [9], [10], treats Esterel as having discrete-event semantics and generates a compiled event-driven simulator. For each segment of code between a *pause* or signal test, their compiler generates a small C function dispatched by a hard-coded scheduler. Each such function typically produces side effects and may schedule other functions to execute later in the same cycle or in the next cycle.

While this approach avoids doing work for most idle sections of code (improving performance over gate-based compilation), its scheduler does not take advantage of mutual exclusion among parts of the program (e.g., between branches of a conditional). EC exploits this, instead using program counters affecting multiway branches. EC produces faster code as a result.

The overall flow of EC—building a concurrent intermediate representation, scheduling it, and generating sequential code—was inspired by Lin's compiler [11], which compiles a concurrent variant of C. Lin translates his language into Petri nets that work well for Lin's rendezvous-style communication, but are awkward for Esterel's synchronous style. Lin schedules and then simulates these Petri nets to generate very fast automata-style code. Unfortunately, this technique can cause an exponential explosion in code size, even with an optimal schedule.

Later, Zhu and Lin [12] proposed an algorithm that avoids the exponential increase in code size by allowing each process to suspend and resume at interprocess communication points (i.e., where it might have to wait for another process to handshake). The result is a collection of processes implemented as coroutines invoked in round-robin order.

EC improves upon Lin's work by generating more compact code that is less affected by poor quality schedules. Lin's compiler would benefit from using EC's sequential code generator. EC also improves upon Zhu and Lin's work because Esterel's semantics allow EC to statically schedule the execution of concurrent processes: Zhu and Lin resort to a round-robin scheme that may waste time deciding what to execute next.

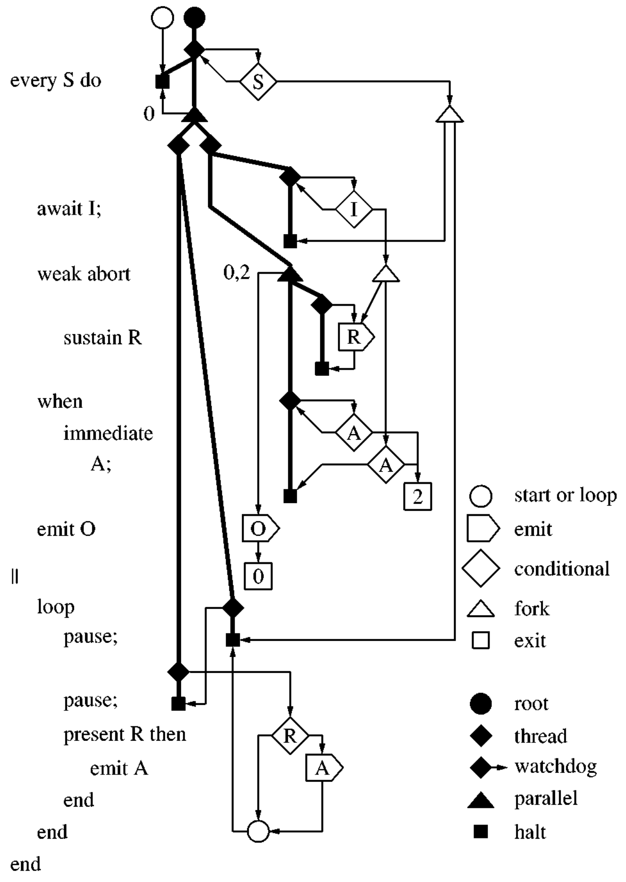
IV. The IC Format

This section describes the input to EC, the IC format. EC reads this instead of Esterel source because it is a more convenient starting point; IC contains high-level information, yet it is easier to manipulate and can be assumed correct. We use the front end of Berry's group's compilers to translate Esterel source into IC.

IC consists of a fairly traditional control-flow graph dangling from a reconstruction tree (Fig. 3). The reconstruction tree (dark nodes and arcs) coordinates exceptions, preemption, and concurrency by dictating how the program resumes ("reconstructs its state") in each cycle. Gonthier developed IC as part of his thesis [13] and it has continued to evolve with the Esterel language.

In a cycle, an Esterel program executes in three phases. In the first phase, the program tries to resume where it paused in the last cycle after checking strong preemption conditions such as *every S*. After this, normal statements such as *emit* and *present* run until control reaches a statement such as *pause*. In the final phase, termination and exceptions are checked and handled. IC models these three phases by sending control down the reconstruction tree toward leaves that were reached at the end of the last cycle, through nodes in the control-flow graph, and finally back up the tree.

Before the first cycle, control flows from the start node to initialize the program. In the example, this immediately sends control to the halt associated with the *every S* statement [Fig. 4(a)].



```
every S do
  await I;
  weak abort
    sustain R
  when
    immediate
      A;
  emit O
||
  loop
    pause;
    pause;
    present R then
      emit A
    end
  end
end
```

Fig. 3. The IC graph for the program in Fig. 1. Each node is drawn to the right of its instruction where possible. The thin lines and outlined nodes are a control-flow graph with concurrency. The thick lines and solid nodes form the reconstruction tree, responsible for restarting the program at the beginning of each cycle.

When control reaches a halt, it starts walking up the reconstruction tree toward the root, marking its path at each reconstruction instruction to prepare the program to resume where it halted. When control reaches the root, the program is finished for the cycle.

At the beginning of each cycle, control starts at the root node and walks down the reconstruction tree along the path taken up the tree at the end of the last cycle. When control reaches a watchdog statement it takes a detour, usually to check a preemption signal. For the example in the first cycle, control starts at the root then flows to the first watchdog and to the test for S. If S is absent, control is sent back to the watchdog, which sends control to the halt and back up the reconstruction tree [Fig. 4(b)]. If S is present, control flows to the fork [Fig. 4(c)].

When control reaches a watchdog from a sequential node [e.g., when S is absent in Fig. 4(b) and (d)], the watchdog sends control to the child along the path taken up the reconstruction tree at the end of the last cycle. So if control followed the path in Fig. 4(b) in the last cycle, the path to the halt was marked and control will flow like Fig. 4(b) in the next cycle. If instead control reached nodes beneath the parallel in the last cycle, the path leading to the parallel would have been marked and control will follow the path in Fig. 4(d) in the next cycle.

Fork and parallel nodes start and resume threads. If S is present, control flows through the topmost watchdog, through

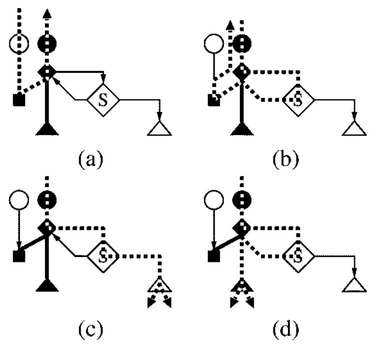


Fig. 4. Flow of control around IC nodes implementing the *every S* statement. (a) Initialization: control begins at the start node, flows to the halt and stops. (b) S is absent and the halt was active last cycle (i.e., the body of the *every* is not running): the watchdog sends control to the halt. (c) S is present: control flows to the fork. (d) S is absent and the parallel was active last cycle (i.e., the *every* is running): the watchdog sends control to the parallel.

the conditional, and to the fork [Fig. 4(c)], thus implementing the *every S do* instruction. Control splits when it reaches the fork, starting two threads and sending control to two halts (one just after *await I*, the other after the first *pause*). If S is absent in the next cycle, control flows to the parallel [Fig. 4(d)] and splits toward both thread nodes, resuming the two threads.

In addition to restarting threads, a parallel node handles thread termination and exceptions by checking the exit levels of the threads beneath it when control is passed to it from below. When a thread is done for the cycle, it can terminate by running an exit at level 0, pause by reaching a halt, corresponding to level 1, and throw an exception by running an exit at level 2 or higher. An exit node sends control directly to the parallel for its thread; a halt sends control back up the reconstruction tree to the closest parallel. Once all of a parallel's threads have returned control to it, the parallel sends control either back up the reconstruction tree or to an exception handler, depending on the highest exit level of all its threads. Using the highest exit level means the parallel only terminates if all its threads have terminated, and exceptions take precedence over paused or terminated threads.

To illustrate exit level behavior, consider the case when the *sustain R* statement is terminated. The node that emits R sends control to the halt immediately beneath it, indicating an exit at level 1 (i.e., halt), and control flows up to the second parallel. Meanwhile, this causes the test for R to succeed and causes A to be emitted, sending control to the second-to-last halt (an exit at level 1) and up the reconstruction tree to the first parallel. The presence of A is noted and causes the exit at level 2 to be executed. The exit level of the threads at the second parallel is therefore 2, so the parallel sends control to the node that emits O and exits at level 0. Both threads under the second parallel are now terminated because the right thread under the top parallel has terminated. The topmost parallel has an exit level of 1 since the other thread halted, so control flows back up the reconstruction tree to the root and the program is done for the cycle.

V. Overview of the New Compiler

The EC compiler produces better code for sequential processors by choosing a more appropriate intermediate representation

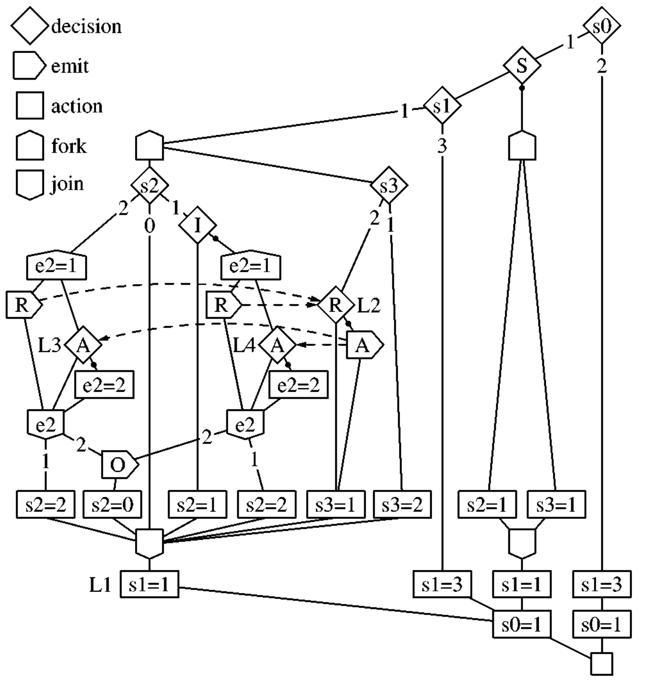


Fig. 5. The CCFG EC generates for the program in Fig. 1. Dashed lines represent data dependencies. Variables s0, s1, s2, and s3 store state between cycles; e2 holds the exit level of the group of threads. Initially, s0 = 2 and all other variables are uninitialized.

than other compilers. The representation—a CCFG—is semantically closer to the Esterel source and the final generated code than the automata, netlists, or event graphs used by other Esterel compilers. The result is code that looks more like a direct implementation of the Esterel program instead of a simulation of its behavior as a circuit or a discrete-event system.

EC interprets the control flow of the IC format much like the gate compilers, so the generated code is of comparable (usually linear) size and avoids the exponential increase of the automata compilers. However, unlike the gate compilers, EC is able to avoid wasting time performing computations in inactive portions of the program because it preserves control flow. The result is code about the same size as that from a gate-based compiler that can run as much as 100 times faster.

The CCFG EC uses as an intermediate representation can be translated into software almost as easily as the branching programs of the automata compilers and has concurrent semantics so it can be generated from IC using an algorithm almost exactly like that in the gate-based compiler. Removing concurrency is the one challenge. We present an efficient algorithm for this in Section VII.

EC translates the IC graph in Fig. 3 into the CCFG in Fig. 5 using the algorithm in Fig. 7. This mainly transforms the reconstruction tree into semantically simpler conditional, fork, and join nodes, but also compiles away much of the walk up the reconstruction tree. Rather than storing information about the path taken up the reconstruction tree at each reconstruction node, this information is stored in one variable per thread (s0, s1, s2, and s3) encoded using the algorithm described in Section VI-D.

A CCFG has software-like semantics, but its concurrency must be removed. Removing concurrency is complicated mainly

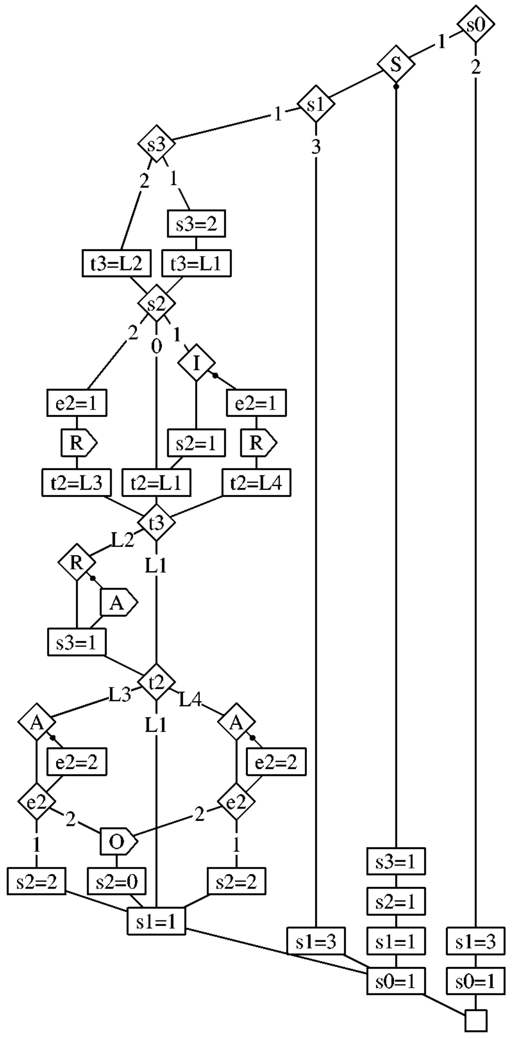


Fig. 6. The SCFG EC generates for the program in Fig. 1. Three context switches—tests and assignments of t2 and t3—were introduced to interleave the execution of the threads.

by Esterel's ability to communicate between threads in the same cycle. In a program such as Fig. 1, the execution of the two threads must be interleaved, i.e., the second thread must run after the first thread runs *emit R* but before the first thread runs *emit O*. The dashed lines in Fig. 5 show these dependencies.

EC interleaves concurrently running threads by inserting code that simulates a context switch. Instead of using a costly operating-system-like mechanism to save and restore the processor's state, each thread simply writes its control state to a variable (a single constant) and resumes with a multiway branch. In effect, the C compiler becomes responsible for saving and restoring context (register contents and the program counter) and can do it more efficiently since it knows which variables are live. In Fig. 6 (the SCFG EC generated from the CCFG in Fig. 5), EC has inserted three such context switches, which write and test variables t2 and t3.

EC can only handle programs where the instructions can run in the same order in all states, much like the acyclic circuits generated by the gate-based compilers. This is a fundamental limitation since EC is based on static scheduling. Although each instruction may or may not execute in each cycle, the statements appear in a particular order in the generated code and can only be executed in that order. Thus, EC is unable to compile all valid Esterel programs, but the class of programs it can compile is broad, interesting, and includes all the large Esterel programs we know of.

VI. Translating IC Into a Concurrent Control-Flow Graph

EC starts by translating an Esterel program expressed as an IC graph (Fig. 3) into a CCFG such as Fig. 5. This replaces preemption conditions with simple conditionals, inserts multiway conditionals to resume threads at the beginning of each cycle, and replicates code that is executed twice or more in the same cycle (Section VI-B explains this "reincarnation" problem in detail). The result is a representation that is close to the natural representation of code on a sequential processor (i.e., a flowchart), but concurrent. Section VII explains how to remove concurrency.

A CCFG contains action, conditional, fork, and join nodes, each with an expression. When control reaches a node, the node's expression is evaluated and control flows along one or more arcs leaving the node. An action node has a single outgoing arc and its expression is usually an assignment. Control leaves a conditional node along the arc whose integer label matches the value of the expression. These become *if* or *switch* statements in C.

Fork and join nodes start and collect groups of parallel threads. Control flows out all arcs leaving a fork, starting a group of threads that will rendezvous at a matching join node before continuing. Fork and join nodes may nest, but control may not pass between threads. Specifically, all paths from a particular fork meet for the first time at the matching join. The CCFGs built by the translation algorithm described below always have this structure.

A. The Translation Algorithm

Fig. 7 presents the recursive algorithm for converting an IC graph into a CCFG, an adaptation of the algorithm Berry developed for synthesizing gates from Esterel. It consists of two recursive functions with side effects, seqNode and recNode, that visit IC nodes and build the CCFG on the way. The algorithm is split into two routines mainly to distinguish the two ways control can reach a watchdog node. The arguments to the two routines are n, the IC node being copied, l, the reincarnation level (explained in the next section), t, the thread of node n, and j, the join node for the current group of threads.

The two functions perform a modified depth-first traversal of the IC graph, adding and copying nodes to the CCFG as they go. The seqNode function begins with a test to see whether the node has been visited before; recNode needs no such test since the reconstruction nodes form a tree.

Exit and halt nodes have no successors, so they terminate the recursion. Each becomes a pair of nodes, one that sets its thread's state (encoding these variables is described in Section VI-D) and one that sets the exit level for the parallel that spawned the threads. The last of these two nodes branches to the join for the current group of threads.

```
function seqNode($n, l, t, j$)
  if node $n$ already exists at level $l$ then
    return sequential node for $n$ at level $l$
  case $n$ of
  Exit, Halt :
    $n'$ = new assignment($t$.state = $\cdots$)
    $n''$ = new assignment($t$.parallel.exitLevel = $\cdots$)
    add arcs $n' \to n'' \to j$
  Conditional, Emit :
    $n'$ = copy of $n$
    for all successors $s$ of $n$ do
      add arc $n' \to$ seqNode($s, l, t, j$)
  Watchdog :
    $n'$ = new conditional($t$.state)
    for all reconstruction children $s$ of $n$ do
      add arc $n' \to$ recNode($s, l, t, j$)
  Fork :
    $n'$ = new fork($n$.parallel.exitLevel = $\cdots$)
    $j'$ = new join($n$.parallel.exitLevel)
    for all successors $s$ of $n$ do
      $t'$ = thread of successor $s$
      add arc $n' \to$ seqNode($s, l, t', j'$)
    for all reachable exception successors $s$ of $n$.parallel do
      add arc $j' \to$ seqNode($s, l, t, j$)
  return $n'$

function recNode($n, l, t, j$)
  case $n$ of
  Halt :
    return seqNode($n, l, t, j$)
  Watchdog :
    return seqNode($n$.successor, $l, t, j$)
  Parallel :
    $n'$ = new fork($n$.exitLevel = $\cdots$)
    $j'$ = new join($n$.exitLevel)
    for all threads $t'$ beneath $n$ do
      $c'$ = new conditional($t'$.state)
      add arcs $n' \to c' \to j'$
      for all reconstruction children $s$ in thread $t'$ do
        add arc $c' \to$ seqNode($s, l+1, t', j'$)
    for all reachable exception successors $s$ of $n$ do
      add arc $j' \to$ seqNode($s, l, t, j$)
    return $n'$
```

Fig. 7. How to translate an IC graph into a CCFG (after Berry). The recursive procedures seqNode and recNode visit and copy IC nodes by following sequential and reconstruction arcs, respectively. n: IC node being visited, l: reincarnation level (see Section VI-B), t: thread in which n resides, j: CCFG join node for the thread.

The basic recursive step happens for conditional and emit nodes. These are simply copied to the CCFG and arcs added to copies of their successors.

There are two ways to reach a watchdog. From a sequential node such as a conditional or emit, a watchdog becomes a conditional that checks its thread's state and branches to one of its reconstruction children. This is the rule for a Watchdog node in seqNode. From a reconstruction node, a watchdog sends control to its single sequential successor. This is the rule in recNode.

Not surprisingly, fork and parallel nodes are the most complicated. In both cases, they synthesize new threads after adding fork and join nodes that reset and test the exit level associated with the parallel. (Exit and halt nodes set this level, as described above.) Fork is easier to understand: for each of its successors,

```
loop trap T2 in
    pause;
    exit T2
||
    loop trap T3 in
        pause;
        exit T3
    ||
        loop
            emit S;
            pause
        end loop
    end trap end loop
end trap end loop
```

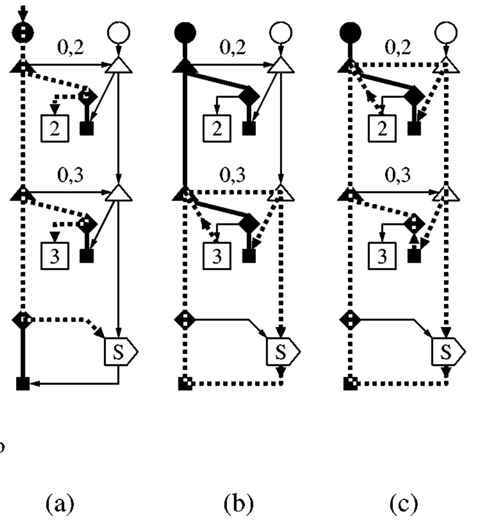


Fig. 8. Reincarnation. *Emit S* executes three times: (a) once at the beginning of the cycle when the two parallels are restarted (level 2), (b) once when *exit T3* terminates and restarts the inner loop (level 1), and (c) once when *exit T2* executes and is executed and causes the outer loop to restart (level 0).

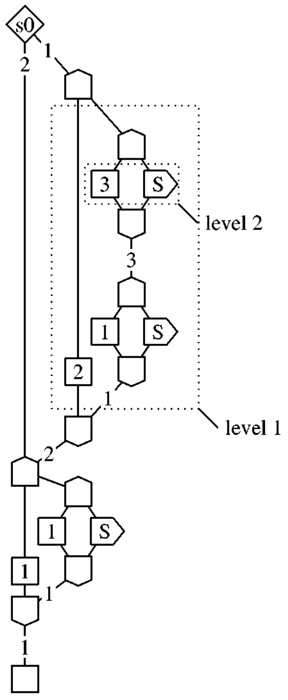


Fig. 9. The CCFG generated for the program with reincarnation in Fig. 8.

it synthesizes the nodes in that thread, instructing them to connect to the just-created join node for the group. After this, it adds arcs from the join node to every reachable exception handling routine on the matching parallel.

The rule for the Parallel is slightly more complicated. For each of its threads, it generates a conditional that checks the state of the thread and either branches to one of the thread node's children or directly to the join if the state is zero (i.e., when the thread is no longer running but one of the other threads under the same parallel is).

B. Unrolling to Remove Reincarnation

A simple depth-first traversal of the IC graph can produce cyclic CCFGs even though Esterel prohibits single-cycle infinite loops. The sequentializing algorithm we present in Section VII requires an acyclic CCFG, so the CCFG synthesis algorithm must remove these cycles by duplicating nodes, visiting them more than once. Berry first developed this unrolling technique to remove cycles from the output of his gate-based compiler.

Even if an Esterel statement is enclosed in a loop, it generally cannot execute twice in a cycle because Esterel forbids loops whose bodies can execute entirely within one cycle. However, if a concurrently running thread throws an exception, the loop can be restarted in the same cycle and the first statement in the loop can run again.

Fig. 8 shows a program with reincarnation. *Emit S* runs three times because it is the first statement in a loop that is terminated and instantly restarted by exceptions. *Emit S* runs first when it is restarted through the reconstruction tree [Fig. 8(a)]. Meanwhile, *exit T3* runs and restarts the inner loop, sending control back to *emit S* through the arc labeled 0,3 [Fig. 8(b)]. Similarly, *exit T2* also runs, terminating and restarting the outermost loop and running *emit S* for a third and final time [Fig. 8(c)]. This behavior is unambiguous because the effects of a trap are always felt after all parallel threads have finished for the cycle.

The CCFG synthesis procedure in Fig. 7 distinguishes the three invocations of *emit S* by maintaining the parallel nesting level l. In general, all the nodes in a thread are at the same level. Nodes in a thread reached from a parallel are at one level greater, but nodes reached through a fork are at the same level.

In Fig. 8, *emit S* can be reached at levels 0 [through the two forks, Fig. 8(c)], 1 [through the top parallel and lower fork, Fig. 8(b)], and 2 [through the two parallels, Fig. 8(c)]. The CCFG synthesis algorithm considers these three separate visits and makes three copies of *emit S*, one per reincarnation, as shown in Fig. 9.

The *emit R* in Fig. 3 can be reached at levels 1 (through the first parallel and the second fork) and 2 (through the two parallels). Since it cannot be reached at level 0 (there is no path to it that does not pass through at least one parallel), only two copies of *emit R* appear in Fig. 5.

C. Computing Reachable Exit Levels

The rules for synthesizing fork and parallel nodes in Fig. 7 involve an iteration over all reachable exception successors of a parallel node. To avoid cycles in the CCFG, it is crucial to distinguish between the exceptions that can be taken by the code in a thread reachable from a parallel and that reachable through the fork. For example, in Fig. 8 only exit level 1 can be reached from the lower fork. This is important since if exit level 3 was erroneously considered reachable, the join node synthesized for the fork would branch back to the fork and create a loop. EC calculates the reachable exits from each fork and parallel using a conservative algorithm due to Berry that considers all paths from a conditional reachable. The information is used both to limit which exception handlers are synthesized from each join and for a variety of other optimizations.

The recursive algorithm for computing exit levels, shown in Fig. 10, builds two sets for each thread, the set of exit levels that can be reached from the fork, i.e., when the thread is first started

```
function seqlevels(node $n$)
  case $n$ of
  Exit :
    return exit level of $n$
  Halt :
    return 1
  Action :
    return seqlevels(successor of $n$)
  Conditional :
    return seqlevels(true successor of $n$) $\cup$
           seqlevels(false successor of $n$)
  Watchdog :
    return reclevels(child $c_1$) $\cup \cdots \cup$ reclevels(child $c_k$)
  Fork :
    for all successors $s_1, \ldots, s_k$ of $n$ do
      $l_k$ = seqlevels($s_k$)
    return parexits(parallel of $n$, $l_1, \ldots, l_k$)

function reclevels(node $n$)
  case $n$ of
  Halt :
    return 1
  Watchdog :
    return seqlevels(successor of $n$)
  Parallel :
    for all reconstruction children $c_1, \ldots, c_k$ of $n$ do
      $l_k$ = reclevels($c_k$)
    return parexits($n, l_1, \ldots, l_k$)

function parexits(parallel node $p$, levels $l_1, \ldots, l_k$)
  $m = \max$(minimum level in $l_1, \ldots,$ minimum level in $l_k$)
  $\{e_1, \ldots, e_j\} = \bigcup_i \{\max(m, x) : x \in l_k\}$
  return seqlevels(handler from $p$ for $e_1$) $\cup \cdots$
         $\cup$ seqlevels(handler from $p$ for $e_j$)
```

Fig. 10. The algorithm for computing reachable exit levels in the IC graph. The two recursive functions seqlevels and reclevels return the exit levels reachable from a node when entered through a sequential arc and arc in the reconstruction tree, respectively. Parexits computes the exit levels reachable at a parallel. The implementation in EC of these functions memoizes two values of the functions, one for the levels visible from a fork, the other from a parallel.

and the set that can be reached when the thread is restarted through the parallel.

The seqlevels and reclevels functions in Fig. 10 recursively compute the exit levels visible from a node reached through a sequential arc and the reconstruction tree, respectively. The rule for conditional considers both true and false branches and simply combines the two sets. The rule for watchdog combines the levels reachable from each of its children in the reconstruction tree.

Not surprisingly, the rules for parallel and fork are the most complicated. Both compute the exit levels that each of their threads can reach. They then pass this information to the parexits function, which looks at the exit levels of each thread to find the exit levels that can be taken by the parallel once the threads are finished for the cycle, and finally combines the levels that can arise from handling any of these exceptions.

The rules in parexit assume that any combination of the exit levels from each thread may appear. However, since the highest exit level always prevails at a thread, parexit first computes the minimum exit level of each thread and uses the highest such value to place a lower bound on the lowest exit level that will be considered. For example, if one thread can exit at levels 0, 2, and 3, and another can exit at levels 1 or 2, the pair can only exit at levels 1, 2, or 3, since if the first thread exits at level 0, the second always takes priority since it exits at levels 1 or 2. This situation occurs frequently since threads often contain infinite loops and thus never terminate at level 0.

D. State Assignment

At the beginning of each cycle, control flows down the reconstruction tree along the path taken up the tree at the end of the last cycle. When control reaches it, each reconstruction node must decide to which of its children it should send control. The state assignment problem amounts to choosing a representation that makes these decisions easy.

The V3 compiler keeps a variable at each node that stores the index of the child that will receive control when the node is next executed. This makes the decisions very easy but tends to waste memory because most nodes have only a few children to distinguish. Furthermore, executing a halt instruction may require setting many variables.

This section presents an encoding that requires less memory and time. The key observation is that an Esterel program can be divided into threads, sections of the reconstruction tree through which at most one path is taken. These are blocks of code separated by | | statements and amount to areas where there is at most a single point of control. In the reconstruction tree, a thread is a subtree rooted at a thread node or the root whose leaves are halts or other parallels.

Instead of one variable per watchdog, EC represents the state of each thread using a single variable. This saves memory, since fewer variables are needed, and time, since a halt simply assigns one constant to one variable instead of walking up the reconstruction tree, but the code at a watchdog is slightly more complicated because it must shift and mask the state before testing it. Fig. 11 illustrates the technique. The code for a path is formed by concatenating the sequence of labels along the branches. The least significant bits are labels at the root. For example, the halt labeled 10110 in Fig. 11 takes its value by being along the path labeled 10, 01, and finally 1. Most branches are labeled 0, 1, 2, etc. encoded in binary using just enough bits to distinguish them. Branches from the root are slightly different; since the all-zero state is used to represent when the thread has terminated, the labels at the root of the thread start at 1.

Such an encoding allows the code at each watchdog to decide which branch to take by a shift and a mask (drawn to the right of the nodes in Fig. 11). The shift is unnecessary at the first decision point within the thread, and the mask is unnecessary at the last. In practice, many threads contain very simple reconstruction subtrees and these optimizations often greatly simplify the code.

This encoding is very compact when the number of branches at each node in the reconstruction tree is a power of two. Codes are wasted, for example, when a node has nine branches beneath it, since this requires four bits and nearly half of the possible codes are unused.

A single thread with many levels of nested watchdogs could overflow the single integers used to represent the state of a single thread. Using more bytes to represent a thread's state is the obvious solution, but we did not implement this because none of

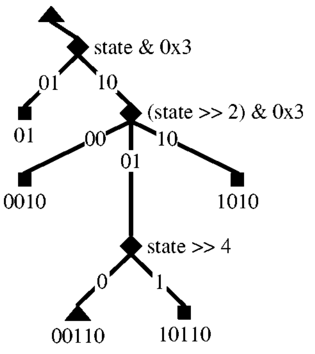


Fig. 11. Encoding states within a thread (a subtree of the reconstruction tree). The encoding of each state (written below each halt or parallel) is formed by concatenating the labels along the path from the top parallel. The expression evaluated at each node is written to its right.

the benchmarks had this problem. In handwritten code, most state trees (e.g., Fig. 11) are wide (i.e., contain many pauses), not tall (i.e., do not contain many sequences of nested preemption conditions). The state encoding is trivial for small examples such as Fig. 1.

VII. Removing Concurrency

After EC translates the IC graph representation of an Esterel program into a CCFG using the algorithm in the last section, it generates a SCFG by statically analyzing the concurrent behavior of the program, interleaving the code for each thread and inserting code that simulates context switches. The result is a SCFG that can easily be translated into a C function using the algorithm in Section VIII. Code generated this way runs faster than V5's gate code because it is a better fit to a processor's natural control behavior. The code from V3 runs faster because it avoids the overhead of context switching and internal communication, which it exhaustively analyzes when the program is compiled. But this comes at the expense of extensive code duplication. In effect, the code generated by V3 uses a distinct value of the program counter for each possible combination of program counters and internal signal states in the Esterel program.

The sequentializing procedure operates in three steps. First, it adds data dependency arcs to represent the constraint that each write of a signal must precede each read (the dashed lines in Fig. 5). Second, it schedules the nodes in the CCFG by placing them in a topological order that respects both the existing control dependencies and the newly added data dependencies. Finally, it copies each CCFG node in scheduled order to a SCFG, along the way inserting nodes that save and restore control state where the schedule implies a context switch. (For example, the nodes that write and test variables t2 and t3 in Fig. 6.)

A. Scheduling

A schedule is an order of all the instructions in the program such that each runs when its predecessors have run and its data is ready. EC uses a schedule to determine the order instructions in concurrent threads will run and when one thread must be suspended and another resumed, i.e., when to context switch.

TABLE I
Examples Used in Experiments

Example	IC Nodes	Lines	States	Threads
Berry's runner	50	55	7	7
Reflex game [17]	111	74	8	11
Combination lock	154	101	33	11
GSM [10]	220	432	19	33
Turbochannel bus	412	687	287	85
Wristwatch [18]	465	1088	42	87
Comm. protocol [19]	589		>10000	40
Video generator	892	948	152	138
Task sequencer [20]	1318		>700	247
Shock absorber [21]	3485	2243		135
Chorus [10]	4053	1565		563
Avionics fuel	4260	4594		944
Avionics display [20]	9305			3330

The second column lists the number of IC nodes required to represent the program. The third lists the number of Esterel source lines including comments and whitespace. The fourth lists the number of states the V3 compiler identified, where it was able to, and the fifth lists the number of independent threads, not all of which can run simultaneously.

TABLE II
Number of IC, CCFG, and SCFG Nodes in the Examples

IC	CCFG Nodes		SCFG Nodes		Total
50	57	14%	65	14%	30%
111	126	14	141	12	27
154	135	−11	204	51	32
220	245	11	279	14	27
412	456	11	573	26	39
465	604	30	870	44	87
589	610	4	737	21	25
892	1053	18	1265	20	42
1318	1771	34	2620	48	99
3485	3831	10	4179	9	20
4053	4492	11	6512	45	61
4260	6204	46	8677	40	104
9305	16505	77	27188	65	192

The percentage increase in each is a rough measure of code size and speed. For example, the avionics display example started with 9305 IC nodes. These became 16505 CCFG nodes after unrolling using the algorithm in Figure 7, a 77% increase. The sequentialization algorithm in Figure 13 produced 27188 SCFG nodes from these, a 65% increase over the CCFG, and a 192% increase over the original number of IC nodes.

Any topological order of the CCFG augmented with data arcs is a correct schedule, but certain orders are better than others because they require fewer context switches. For example, a better choice of schedule in Fig. 6 would have eliminated the assignments and test of t3; the nodes s3 and $s3 = 2$ appeared too early. Unfortunately, finding a schedule with a minimum number of context switches is NP-complete, but a bad schedule does not significantly slow or bloat the generated code. Theoretically, the slowdown or bloat may be quadratic, but in practice EC produces acceptable code using a simple depth-first scheduler. Table II shows the sequentializing procedure fed with a simple scheduler increases the number of nodes at most 65% for the largest 16 500-node example.

The optimum scheduling problem is NP-complete because it could be used to solve the minimum feedback vertex set problem. To solve the minimum feedback vertex set problem for a particular directed graph, create a program with one

```
loop
  present n1 then
    present a1 then emit n1a end;
    present a2 then emit n1b end
  ||
    emit a3; emit a4; emit a5
  end present;
  pause
end loop
```

(a)

```
if (n1) {                if (n1) {
  a3 = 1;                  a3 = 1;
  a4 = 1;                  a4 = 1;
  a5 = 1;                  a5 = 1;
  if (a1) n1a=1;         }
  if (a2) n1b=1;         /* ... */
}                        if (n1) {
                           if (a1) n1a=1;
                           if (a2) n1b=1;
                         }
```

(b) (c)

Fig. 12. Construction to demonstrate the NP-completeness of the minimal scheduling problem. (a) For each node in a directed graph, construct a thread like this one for a node $\mathrm{n1}$ with incoming arcs $\mathrm{a1}$ and $\mathrm{a2}$ and outgoing arcs $\mathrm{a3}$, $\mathrm{a4}$, and $\mathrm{a5}$. The signal $\mathrm{n1}$ is an input, $\mathrm{n1a}$, and $\mathrm{n1b}$ are outputs. (b) If this node is not part of a cycle, the code for the two threads can be grouped under a single conditional. (c) If code for the node must be split because of a cyclic path in the graph that passes through the node, the test for $\mathrm{n1}$ must be duplicated.

thread per node, each thread having a single conditional invoking two threads beneath it [an $O(V+E)$ operation], such as in Fig. 12(a). The first of these threads depends on signals corresponding to incoming arcs in the given node; the second emits signals corresponding to outgoing arcs. Instructions corresponding to a node that does not participate in any cycle can be scheduled together to generate code such as in Fig. 12(b), the code for at least one node in each cycle must be split as in Fig. 12(c). This requires duplicating a test, so the code size grows with the number of splits. Splitting a node is analogous to removing it from the graph, so asking if there is a schedule that generates code under a certain size is equivalent to asking if the graph can be made acyclic by removing fewer than a certain number of nodes—the feedback vertex set problem.

EC assumes a static schedule exists, i.e., there is a unique order in which all statements can execute in every cycle. This is true for many programs, but Esterel programs can have data-dependent orders. These may appear when resources are shared and used in different orders in different cycles, or when the system is inherently cyclic, such as a cyclic-ring arbiter. EC is unable to compile such programs.

Automata compilers permit statements to execute in different orders in each state, since they generate separate code for each. Gate-based compilers have a harder time: they must completely explore the state space of the system before concluding that the program is valid. As a side-effect, they determine the function of the system in all of these states and resynthesize the cyclic portions of the netlist. This is the machinery developed by Shiple *et al.* [8]. We discuss the possibility of using this technique in EC in Section X.

B. Building the Graph

Fig. 13 shows the algorithm for synthesizing a SCFG from a CCFG, the most important algorithm in EC and this paper's key contribution. The algorithm walks through the nodes in the CCFG in scheduled order, copying each (line 2) and attaching its incoming arcs at each step (line 24). Normally this just copies the graph, but additional nodes are inserted that save and restore control state when control must switch between concurrently running threads.

Fig. 14 depicts a few steps of the algorithm while it is building Fig. 6. In effect, the algorithm sweeps a line through the concurrent graph, shifting each node across the frontier (the dotted line) and attaching its incoming arcs according to the arcs that cross the frontier. Maintaining these arcs—predecessors (indicated with dashed lines)—is the algorithm's main concern.

A node keeps track of two types of node that may eventually branch to it. The most common is a "normal predecessor," a copy of a CCFG node. Line 21 maintains these predecessors, indicated by dashed arrows in Fig. 14(a). The other type of predecessor is a "restart predecessor," a multiway branch created when a thread is resumed. Line 34 assigns these. Node $\mathrm{t3}$ in Fig. 14(e) is a restart predecessor, indicated by a dotted arrow.

The distinction between normal and restart predecessors comes when a thread is suspended, in line 44. Only normal predecessors are saved. Since a restart predecessor indicates the possibility of something restarting, it is wasteful to save something that never started.

The algorithm simply copies the CCFG when synthesizing a sequence of nodes in the same thread (where context switching is unnecessary). The main loop copies each node (line 2), connects arcs from its predecessors (line 24), and prepares to connect the arcs to its successors (line 21). The resume and suspend procedures do nothing.

Fig. 14(a) and (b) illustrates this common case, showing how a node is synthesized when no context switch is needed. Fig. 14(a) shows $\mathrm{s1}$'s single normal predecessor S, created earlier in line 21 when node S was copied. In Fig. 14(b), line 2 has copied $\mathrm{s1}$ to the sequential graph (moved above the dotted line), line 24 has added an arc from $\mathrm{s1}$'s predecessor S to the new copy of $\mathrm{s1}$, and line 21 has added the new sequential node $\mathrm{s1}$ as a normal predecessor of both $\mathrm{s3}$ and $\mathrm{s1} = 3$.

Synthesizing context switches is more complicated. When one thread is running and a node from another thread must run (usually when the first thread communicates to the second), the control state of the running thread is saved and the control state of the suspended thread is recovered. The suspend and resume procedures accomplish this by adding nodes that save and restore control state. They are recursive because threads are often nested in Esterel.

Fig. 14(c)–(e) shows how this works. In Fig. 14(c), the two join nodes labeled $\mathrm{e2}$ and an unlabeled join are in the thread to be suspended (line 20 placed them there when their predecessors were copied). In Fig. 14(d), the suspend procedure has run; line 43 has added nodes $\mathrm{t2} = \mathrm{L3}$, $\mathrm{t2} = \mathrm{L4}$, and $\mathrm{t2} = \mathrm{L1}$ to save the state of the thread, line 44 has added arcs from the two R nodes, $\mathrm{s2} = 1$, and $\mathrm{s2}$ to the new state-saving nodes, and line 45 has added each of the save state nodes as normal predecessors

```
1:  for all nodes $n$ in scheduled order do
2:    copy $n$ to new node $n'$ in the SCFG
3:    $t$ = thread of $n$
4:    resume($t$)
5:    if $n$ is a join then
6:      $f = n$'s corresponding fork
7:      suspend($f$)
8:      connect($f, n'$)
9:      remove $f$ from thread $t$
10:   else
11:     connect($n, n'$)
12:     remove $n$ from thread $t$
13:   if $n$ is a fork then
14:     mark fork $n$ as Runnable
15:     add $n'$ as a normal predecessor of $n$
16:     add $n$ to thread $t$
17:     add each successor $s$ of fork $n$ to the thread of $s$
18:   else
19:     for all successors $s$ of $n$ do
20:       add $s$ to thread $t$
21:       add $n'$ as a normal predecessor of $s$
22:       if $s$ is a fork then mark $s$ as Runnable

23: procedure connect($n, n'$)
24:   add arc $s \rightarrow n'$ from each normal predecessor $s$ of $n$
25:   if $n$ has a restart predecessor $r$ then add arc $r \rightarrow n'$
26:   clear $n$'s predecessors

27: procedure resume($t$)
28:   $f$ = fork of $t$
29:   if $f$ is not the top then resume(thread of $f$)
30:   if $f$ is Running but not running $t$ then suspend($f$)
31:   if $f$ is Runnable then
32:     $r$ = new restart node for $t$
33:     connect($f, r$)
34:     make $r$ the restart predecessor of each node $n$ in $t$
35:     mark $f$ as Running
36:     set $t$ as the running thread of $f$

37: procedure suspend($f$)
38:   if $f$ is Running then
39:     $t$ = running thread of $f$
40:     for all nodes $n$ in $t$ do
41:       if $n$ is a fork then suspend($n$)
42:       if $n$ has normal predecessors then
43:         $q$ = new save state node
44:         add arc $p \rightarrow q$ from each normal predecessor $p$ of $n$
45:         make $q$ a normal predecessor of $f$
46:       clear $n$'s predecessors
47:     if any node in $t$ had a restart predecessor $r$ then
48:       make $r$ a predecessor of $f$
49:     mark the running thread of $f$ as none
50:     mark $f$ as Runnable
```

Fig. 13. How to convert a CCFG into a sequential one. The main loop steps through the CCFG in scheduled order, copying each node and calling connect() to attach arcs from nodes that could run it next. The suspend procedure suspends a running thread under a fork, and resume prepares it to continue running.

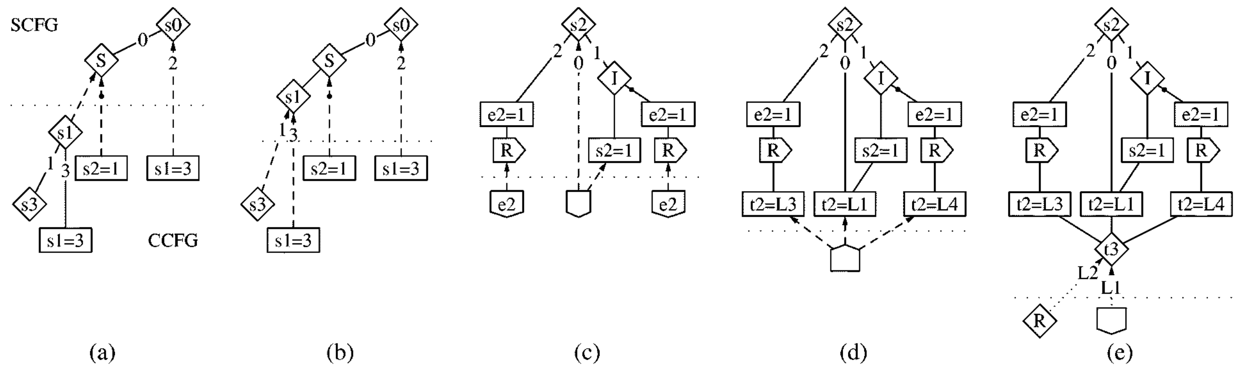


Fig. 14. Synthesizing SCFG nodes. The dotted line separates the SCFG (above) from the uncopied part of the CCFG. In (a) and (b), node s1 is synthesized. In (a), s1 has S as its sole predecessor (dashed arrow). In (b), s1 has been copied to the SCFG, an arc has been added from S to s1 in the SCFG, and nodes s3 and s1 = 3 now have the copy of s1 as their predecessors. (c), (d), and (e) depict the synthesis of a context switch. In (d), the three uncopied nodes in (c) have been run as nodes that save their thread's state and the fork has gained them as predecessors. In (e), the fork node has been run as a conditional that tests the state of the resumed thread, and nodes in that thread have gained it as a predecessor.

of the unlabeled fork that is the parent of this thread (f in the suspend procedure).

Fig. 14(e) shows the state after the resume procedure has run. Line 32 has created node t3, which tests the state of the thread being resumed, line 33 has called connect, causing line 21 to connect arcs from t2 = L3, t2 = L4, and t2 = L5 to t3 (these nodes were the normal predecessors of the fork), and line 34 has added t3 as the restart predecessor of the node that tests R and the unlabeled join; the two nodes that were about to run before the thread was suspended.

Currently, EC uses a trivial encoding for the state of threads suspended within a cycle (this is different from the careful encoding used to save state between cycles described in Section VI-D). Each CCFG node is encoded with the unique small-integer identifier assigned when the node is created. While this is hardly the most efficient encoding, it matters little. Most resume branches are two-way, which become comparisons to a constant. A more efficient encoding might consider the sequence of nodes that are suspended and resumed for a particular thread and try to reuse codes to make the encodings at each resume dense, perhaps using a graph-coloring algorithm.

VIII. Generating Well-Structured C Code

This section describes how to generate attractive, human-readable C code from the SCFG produced by the algorithms in Section VII. Generating correct C code is easy (the SCFG representation was chosen to make this trivial), but making it easy-to-read is harder. We did this to aid users of the compiler when they wish to debug their programs, although it may also make it easier for the C compiler to optimize the result.

The SCFGs generated by the algorithm in the last section are simple. Each node has an expression that is evaluated when the node runs. If the node has two or more successors, the result of the expression is used to choose among them. A code generation algorithm could simply generate the code for each node and use *goto* statements to branch to the appropriate location. The Polis compiler [5], [6] does this and the results, while correct, are inscrutable, although this is partly due to the unstructured control graphs generated from BDD's.

Structuring control-flow graphs has been studied in a few contexts. Baker [14] proposed an algorithm for finding high-level control constructs (e.g., if–then–else, while loops) in *goto*-riddled FORTRAN programs. Cifuentes [15] used a similar procedure to reconstruct high-level information in control-flow graphs generated by decompiling executables. Since loops are the main concern in both works, and the SCFG our compiler generates have no loops, the algorithms are not a good fit for the problem here.

The algorithm in EC generates sequences of statements for sequences of action nodes in the control-flow graph and conditionals—*if–then–else* for two-way branches; *switch* statements for three or more—for nodes with more than one outgoing arc. The challenge is deciding which nodes to include in the body of a conditional and which to place outside.

We choose the immediate postdominator of a conditional node as the first node outside the scope of the body of the conditional. A node d postdominates node n if every path from n to the exit of the control-flow graph passes through d. The immediate postdominator is the unique d such that no other node both postdominates n and is postdominated by d. This is a classical relationship in graph theory, and we use the standard fast algorithm by Lengauer and Tarjan to compute it [16].

Fig. 15 shows the immediate postdominators for the conditional nodes in Fig. 6. As expected, they are choke-points in the control-flow graph and match what a programmer would probably choose.

Fig. 16 shows the recursive algorithm used to build an abstract syntax tree (i.e., that can be easily traversed to produce C source code) from a control-flow graph. The function takes three arguments—the node to be synthesized, the node that follows it, and a flag indicating whether a break statement is necessary to reach this nod—-and returns an AST for the node.

Fig. 16 actually builds the program backward, constructing the nodes that will appear later in the program before those that will appear earlier. This is to ensure all *goto* statements are forward, which is expected when the control-flow graph contains no loops. This is why code for an *else* branch is computed before the *then* branch, and why code for each *case* of a *switch* statement is inserted at the beginning of its body rather than at the end.

Fig. 17 shows the code Fig. 16 generates from the SCFG in Fig. 6. Because the successors of a multiway *switch* statement are built in an undefined order, some arbitrary choices were made. For example, the two *case*s in the second switch statement could have been reversed, but the code after L6 would still have appeared in the later case since Fig. 16 ensures all *goto*s are forward.

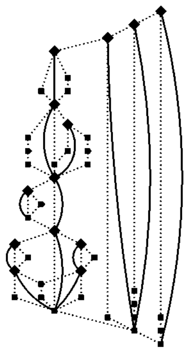


Fig. 15. Postdominators of the conditional nodes in Fig. 6. The solid line from each conditional leads down to the first instruction outside the body of the conditional. The immediate postdominator of a node is the closest node through which all paths from the conditional pass.

```
function astFor($n$, $f$, $b$)
  if $n = f$ then
    return a break statement if $b$ is true; nothing otherwise
  if $n$ has already been synthesized then
    return new "goto $n$" statement
  if $n$ is a conditional then
    $p$ = immediate postdominator of $n$
  else
    $p$ = successor of $n$
  $p'$ = astFor($p$, $f$, $b$)
  case number of successors of $n$ of
  1 :
    $n'$ = new expression statement for $n$
  2 :
    $n'$ = new if-then-else statement
    $n'$.else = astFor($n$.else, $p$, false)
    $n'$.then = astFor($n$.then, $p$, false)
  $\geq 3$ :
    $n'$ = new switch statement
    for all successors $c$ of $n$ do
      insert astFor($c$, $p$, true) at the beginning of $n'$
  append $p'$ to $n'$
  return $n'$
```

Fig. 16. The algorithm for building an AST from a SCFG such as Fig. 6. This recursive function builds later statements first (e.g., the else before the then branch) to ensure all *goto* statements branch forward.

IX. Experimental Results

We ran experiments to compare the quality of code generated by the new compiler to that from the V3 automata compiler, the V5 gate-based compiler, the output of V5 after being passed through logic synthesis, and the compiler by Bertin *et al.* [9], [10]. To measure average cycle times, we ran the generated program for a second and counted the number of cycles it executed. We generated pseudorandom input patterns with an testbench generated by the algorithm of Yuan *et al.* [22] to produce inputs that observed environmental constraints (e.g., mutually exclusive input signals) but did not weight the inputs. Measured times

```
/* L1 .. L4 are unique small-integer constants */
if ((s0 & 3) == 1) {
  if (S) {
    s3 = 1; s2 = 1; s1 = 1;
  } else
    if (s1 >> 1)
      s1 = 3;
    else {
      if ((s3 & 3) == 1) {
        s3 = 2; t3 = L1;
      } else {
        t3 = L2;
      }
      switch (s2 & 3) {
      case 0: goto L0;
      case 1:
        if (I) {
          e2 = 1; R = 1; t2 = L4;
        } else {
          s2 = 1;
        L0:
          t2 = L1;
        }
        break;
      case 2:
        e2 = 1; R = 1; t2 = L3; break;
      }
      if (t3 == L2) {
        if (R) A = 1;
        s3 = 1;
      }
      switch (t2) {
      default: break;
      case L4:
        if (A) e2 = 2;
        if (e2 == 2) goto L6;
        s2 = 2;
        break;
      case L3:
        if (A) e2 = 2;
        if (e2 == 1) s2 = 2;
        else {
        L6:
          O = 1; s2 = 0;
        }
        break;
      }
/* L5: */
      s1 = 1;
    }
  s0 = 1;
} else {
  s1 = 3; s0 = 1;
}
```

Fig. 17. Code generated from Fig. 6 by the algorithm in Fig. 16. This behaves like the concurrent Esterel program in Fig. 1 and is the final output of the compiler.

do not include time to run the testbench. We ran the testbench separately and subtracted out its effects, which was as high as 75% of the execution time for the small examples, and probably distorted their times, but was less than 1% for the large examples.

The example programs, listed in Table I, were all handwritten and range from toy examples (hundreds of lines) to industrial size (nearly 10 000 lines).

Fig. 18 shows the average cycle times and executable sizes we measured for these examples on two machines: a 336-MHz UltraSPARC-II and a 233-MHz Intel Pentium. Compared to EC, the V5 compiler produces consistently slower, larger code, but running the output of V5 through an aggressive logic optimizer can eliminate the gap, albeit only after many minutes of work and only for the smaller examples. Logic optimization did not finish in an hour on the examples over 1000 lines; EC was able to compile the largest example in 10 s.

The V3 compiler, when it runs, generally produces faster code, but at the expense of very large executables. It did not complete in an hour for any example larger than 1000 lines, and failed on a 600-line example. More precisely, V3 appears not to be able to compile programs with more than about 300 states.

Bertin *et al.*'s compiler produces code about twice as big and twice as fast as that from EC. The missing data points for large examples were due to the source not being available for some and because of a bug in their compiler that incorrectly rejected some programs. In theory, their approach should scale as well as EC's.

Small programs run faster on the lower clock-rate Pentium, but larger examples run slower. We suspect this is due to different cache sizes on the two processors.

The speed of V5 without logic optimization very closely tracks the size of the program. This is expected since each source statement becomes a few gates, and each gate is executed once per cycle, thus the speed should be directly proportional to the source program size. The speed of the code from the other compilers appears to have little to do with the size of the source. This is expected, since the speed of the code from these compilers is related to the number of source instructions that must execute each cycle, which differs from the number of instructions in the program an varies among programs.

The wristwatch example (465 nodes) shows the least variance among the different compilers because it calls a substantial number of external routines. Much of its execution time is spent in these routines, so the quality of the generated code matters less.

Compilation time becomes noticeable for the larger examples, but the C compiler is the bottleneck. For example, V5 without logic optimization was able to produce C code for the largest example in 7 s, and EC was able to produce it in 10 s, but it took Sun's CC about 2 min to compile the output from EC, and 45 min to compile V5's output.

Compilation times for V3 can range from very short to unacceptably long (hours). Unfortunately, patience in running V3 is not rewarded since long runs produce impractically large executables.

X. Conclusion and Future Work

This paper has presented a new way to compile the synchronous language Esterel that preserves much of the program's original control structure for a code size and speed advantage. It translates Esterel's preemption and exception constructs into conditional branches and compiles away its concurrency by statically scheduling the instructions and inserting code that saves control state in variables and restores it with conditional branches. Ultimately, it produces mostly structured C code that contains some *goto*s.

Experiments show EC produces code that can be 100 times faster and half the size of code from other high-capacity compilers.

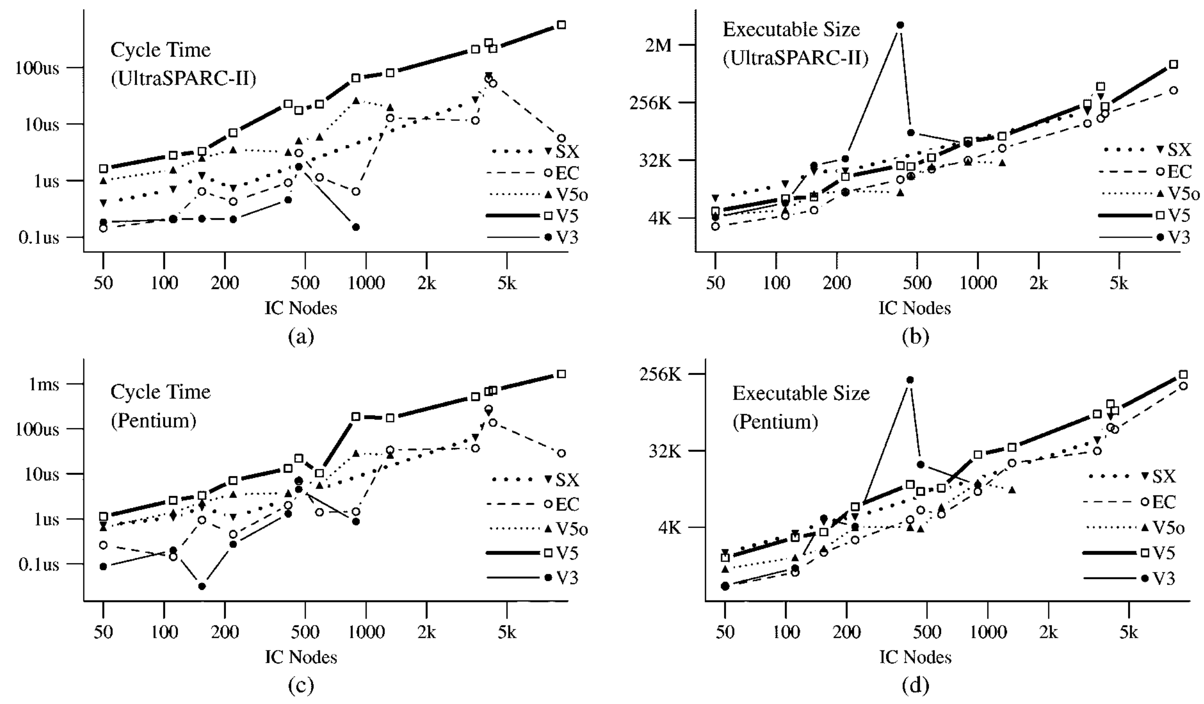


Fig. 18. Average cycle times for random inputs and executable sizes as a function of source program size for the examples in Table I. The four compilers are the automata-based V3, the gate-based V5, the output of V5 run through logic optimization, the new compiler EC, and Bertin *et al.*'s compiler (SX). Missing data points indicate part of the compilation chain failed or the source code was not available. (a) Average per-cycle execution times as a function of the source program size for a 336-MHz Sun UltraSPARC-II with a 4-MB cache, (b) sizes of the executable (generated by Sun's "cc -O"), (c) average cycle times on a 233-MHz Pentium with a 512-kB cache, and (d) executable sizes for the Pentium (generated by egcs 2.91.66 -O).

EC is currently used to generate simulation code in CoCentric System Studio (described under an earlier name by Buck and Vaidyanathan [23]), an environment that allows designers to specify systems using a mixture of dataflow graphs and hierarchical finite-state machines. To compile a simulation, System Studio translates control behavior into IC programs, EC compiles them into C, and the result is linked code generated by System Studio for the dataflow portion.

Other applications are possible, in addition to compiling Esterel, EC could easily be adapted to compile other synchronous, concurrent languages, such an Lavagno and Sentovich's ECL [24].

Many further optimizations are possible. The automata compilers can produce much better code for small examples. One possibility is to apply the automata compilation technique to small segments of a much larger program, such as those with frequent synchronizing communication. Such things usually have far fewer states than a simple product would suggest and are exactly those where automata code would be far better than that from EC.

Handling apparently cyclic programs is another challenge. One approach would be to resynthesize cyclic portions of the circuit as the V5 compiler does using the technique due to Shiple *et al.* [8]. They generate an exact three-valued next-state function by unrolling the circuit according to the algorithm of Bourdoncle [25] and try to use it to prove that no state with unknown outputs is reachable. If they succeed, they resynthesize the cyclic portion of the circuit by forcing the exact three-valued next state function to take two values. Specifically, if in some state an output of the three-valued function is unknown, they replace that output with a 1. This does not affect the behavior of the program because their technique proves that none of these states can be reached.

How to apply this Boolean technique to control-flow graphs is not obvious, but another approach (suggested to me by Berry) is possible. A cyclic network can always be evaluated by unrolling it and simulating it using three-valued logic (i.e., each signal is either present, absent, or unknown), but simulating three-valued signals is costly in software, especially when simulating the program counter. However, it is not necessary when the program being evaluated is monotonic and guaranteed to always have defined outputs. Esterel's constructive semantics [26] guarantees programs are monotonic; a more defined input always produces an equal or more-defined output. Specifically, changing an input from unknown to known can only change an undefined output to known or leave the output unchanged. It follows that all unknown inputs can be set to arbitrary, known values without affecting the output provided the program is known never to produce undefined outputs.

Concretely, this technique would unroll each strongly connected component of the CCFG using either Bourdoncle's [25] or our [27] scheduling algorithm. The amount of unrolling necessary follows from the structure of the program and noting that each signal can be either undefined or defined. Signals in the SCC would be initialized to absent (the value does not matter) and constant propagation on the resulting code would then greatly simplify it. However, there is still the strong possibility of a quadratic blow-up in code size with this technique.

One of the reviewers noted that EC can be thought of as having factored the automata code from V3. In effect, EC shares code common to two or more states by predicating it with variables that represent control state and internal signals. This observation raises the possibility of a compiler that analyzes the state space of an Esterel program and generates distinct pieces of code not for single states, but for groups of states for which EC generates efficient code. The result should be faster but larger than normal EC output.

Most future work involves combining the ideas from all the existing compilers, each of which have certain strengths.We are confident that the result would be capable of producing fast, small code for virtually all programs.

REFERENCES

[1] A. Benveniste and G. Berry, "The synchronous approach to reactive realtime systems," *Proc. IEEE*, vol. 79, pp. 1270–1282, Sept. 1991.
[2] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Programming*, vol. 19, pp. 87–152, Nov. 1992.
[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE*, vol. 79, pp. 1305–1320, Sept. 1991.
[4] P. Caspi, A. Girault, and D. Pilaud, "Automatic distribution of reactive systems for asynchronous networks of processors," *IEEE Trans. Software Eng.*, vol. 25, pp. 416–427, May 1999.
[5] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. Sangiovanni-Vincentelli, and E. Sentovich, "Synthesis of software programs for embedded control applications," in *Proc. 32nd Design Automation Conf.*, San Francisco, CA, June 1995, pp. 587–592.
[6] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki, "Synthesis of software programs for embedded control applications," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 834–849, June 1999.
[7] C. Castelluccia, W. Dabbous, and S. O'Malley, "Generating efficient protocol code from an abstract specification," *IEEE/ACM Trans. Networking*, vol. 5, pp. 514–524, Aug. 1997.
[8] T. R. Shiple, G. Berry, and H. Touati, "Constructive analysis of cyclic circuits," in *Proc. European Design and Test Conf.*, Paris, France, Mar. 1996, pp. 328–333.
[9] V. Bertin, M. Poize, and J. Pulou, "Une nouvelle méthode de compilation pour le language ESTEREL [A new method for compiling the Esterel language]," in *Proc. GRAISyHM-AAA*, Lille, France, Mar. 1999.
[10] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou, "Efficient compilation of Esterel for real-time embedded systems," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, CA, Nov. 2000, pp. 2–8.
[11] B. Lin, "Software synthesis of process-based concurrent programs," in *Proc. 35th Design Automation Conf.*, San Francisco, CA, June 1998, pp. 502–505.
[12] X. Zhu and B. Lin, "Compositional software synthesis of communicating processes," in *Proc. IEEE Int. Conf. Computer Design (ICCD)*, Austin, TX, Oct. 1999, pp. 646–651.
[13] G. Gonthier, "Sémantiques et modèles d'exécution des langages réactifs synchrones; application à Esterel. [Semantics and models of execution of the synchronous reactive languages: Application to Esterel]," Thèse d'informatique, Université d'Orsay, 1988.
[14] B. S. Baker, "An algorithm for structuring flowgraphs," *J. Assn. Computing Machinery*, vol. 24, pp. 98–120, Jan. 1977.
[15] C. Cifuentes, "Structuring decompiled graphs," in *Proc. Int. Conf. Compiler Construction*. Linkoping, Sweden: Springer-Verlag, Apr. 1996, vol. 1060, Lecture Notes in Computer Science, pp. 91–105.
[16] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Programming Languages and Systems*, vol. 1, pp. 121–141, July 1979.
[17] R. Bernhard, G. Berry, F. Boussinot, G. Gonthier, A. Ressouche, J.-P. Rigault, and J.-M. Tanzi. (1989, May) Programming a reflex game in Esterel V3. [Online]. Available: http://www.esteret.org
[18] G. Berry. (1991) Programming a digital wristwatch in Esterel V3. Rapport de recherche 8, Centre de Mathematiques Appliquees, Ecole des Mines de Paris. [Online]. Available: http://www.esterel.org
[19] F. Clouté, J.-N. Contensou, D. Esteve, P. Pampagnin, P. Pons, and Y. Favard, "Hardware/software co-design of an avionics communication protocol interface system: An industrial case study," in *Proc. 7th Int. Workshop Hardware/Software Codesign (CODES)*, Rome, Italy, May 1999, pp. 48–52.
[20] G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. De Simone, "Esterel: A formal method applied to avionic software development," *Sci. Comput. Programming*, vol. 36, pp. 5–25, Jan. 2000.
[21] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, and A. Sangiovanni-Vincentelli, "A case study in computer-aided co-design of embedded controllers," *Design Automation Embedded Syst.*, vol. 1, pp. 51–67, Jan. 1996.
[22] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling design constraints and biasing in simulation using BDDs," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, San Jose, CA, Nov. 1999, pp. 584–590.
[23] J. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in El Greco," in *Proc. 8th Int. Workshop Hardware/Software Codesign (CODES)*, San Diego, CA, May 2000.
[24] L. Lavagno and E. Sentovich, "ECL: A specification environment for system-level design," in *Proc. 36th Design Automation Conf.*, New Orleans, LA, June 1999, pp. 511–516.
[25] F. Bourdoncle, "Efficient chaotic iteration strategies with widenings," in *Formal Methods in Programming and Their Applications: Int. Conf. Proc.*. Novosibirsk, Russia: Springer-Verlag, June 1993, vol. 735, Lecture Notes in Computer Science.
[26] G. Berry. The constructive semantics of pure Esterel. [Online]. Available: http://www.esterel.org, 1999
[27] S. A. Edwards, "The specification and execution of heterogeneous synchronous reactive systems," Ph.D. dissertation, Available as UCB/ERL M97/31, Univ. California, Berkeley, 1997.

Stephen A. Edwards (M'97) received the B.S. degree in electrical engineering from the California Institute of Technology, in 1992, and the M.S. and Ph.D. degrees, also in electrical engineering, from the University of California, Berkeley, in 1994 and 1997, respectively.

He is currently an Assistant Professor in the Computer Science Department of Columbia University in New York, which he joined in 2001 after a three-year stint with Synopsys, Inc., in Mountain View, CA. His research interests include embedded system design, domain-specific languages, and compilers. He is the author of *Languages for Digital Embedded Systems* (Boston, MA: Kluwer, 2000).