Tutorial: Compiling Concurrent Languages for Sequential Processors

STEPHEN A. EDWARDS
Columbia University

Embedded systems often include a traditional processor capable of executing sequential code, but both control and data-dominated tasks are often more naturally expressed using one of the many domain-specific concurrent specification languages. This article surveys a variety of techniques for translating these concurrent specifications into sequential code. The techniques address compiling a wide variety of languages, ranging from dataflow to Petri nets. Each uses a different method, to some degree chosen to match the semantics of concurrent language. Each technique is considered to consist of a partial evaluator operating on an interpreter. This combination provides a clearer picture of how parts of each technique could be used in a different setting.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; D.3.4 [**Programming Languages**]: Processors—*compilers, code generators, and optimization*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation

General Terms: Languages

Additional Key Words and Phrases: Compilation, concurrency, Esterel, Lustre, Verilog, Petri nets, sequential, code generation, communication, dataflow, discrete-event, partial evaluation

1. INTRODUCTION

Although finding parallelism in a sequential program is useful when compiling it to a parallel machine, embedded software often needs to be translated in the other direction. Because an embedded system often responds to and controls multiple, concurrent physical processes, it is natural to describe these systems concurrently using one task per process. Yet these systems are often implemented using a single processor to reduce cost, so it is necessary to simulate the concurrency in software.

Simulating concurrency is traditionally the responsibility of an operating system. Timesharing operating systems strive for fair scheduling [Silberschatz

Part of this work was done while the author was with Synopsys, Inc., Mountain View, California. The author's research is currently supported by the NSF under CCR–0133348, SRC, MDC, and Intel Corporation.

Author's address: Computer Science Department, Columbia University, 1214 Amsterdam Avenue MC 0401, New York, NY 10027; email: sedwards@cs.columbia.edu.

Table I. Languages Compiled by Techniques Discussed in this Article

Language	Concurrency	Specification Style	Communication
Esterel	Single-rate synchronous	Imperative	Synchronous broadcast
Lustre	Single-rate synchronous	Declarative	Synchronous broadcast
Verilog	Discrete-event	Both	Memory and events
C	None	Imperative	Memory
Picasso C	Asynchronous	Imperative	Rendezvous
FlowC	Asynchronous	Imperative	Buffered FIFOs
SDF	Rational-rate synchronous	Declarative	Buffered FIFOs

and Galvin 1998; Tanenbaum 1992]; real-time operating systems aim to meet deadlines by running higher-priority processes in favor of lower-priority ones [Briand and Roy 1999; Labrosse 1998]. Using an operating system is appropriate for general-purpose computers designed to handle arbitrary applications, but can be wasteful for fixed-function embedded systems because of the time spent scheduling and context switching.

This article surveys an alternative approach that trades flexibility for efficiency: techniques that compile concurrent languages into sequential procedural code that can be executed on general-purpose processors without operating system support. By restricting and analyzing the behavior of the system before it runs, most overhead can be avoided, producing a faster, more predictable system. Lin [1998] showed an eightyfold speedup on one example.

Compiling concurrent specifications into sequential code has applications outside embedded software. Most digital hardware designs, which are highly concurrent, are first simulated on sequential computers before being fabricated to check for errors. Also, certain code-motion optimizations are more easily seen in a concurrent representation generated from a sequential one.

This article discusses 10 fairly different techniques for compiling concurrent programs into sequential ones. The techniques are, for the most part, not interchangeable since each depends greatly on the particular concurrent language being compiled. As a result, this article does not attempt to make quantitative comparisons of the efficiency of these techniques. The remainder of this section compares the techniques by viewing them from different perspectives: the languages being compiled, their styles of communication, the code-generation approach of each technique, the concurrent model it uses, and finally the sequential model ultimately generated by each. After this, Sections 2 through 11 contain detailed descriptions of each technique, each centered around an example specific to the language being compiled. Table II lists the techniques. Finally, Section 12 presents conclusions.

1.1 The Languages

The 10 compilation techniques discussed in this article work on the seven languages listed in Table I. The most significant difference among these languages is their model of concurrency. Esterel [Berry and Gonthier 1992] and Lustre [Caspi et al. 1987] are members of the synchronous family of languages whose semantics are based on a synchronous model of time like that used in synchronous digital hardware. A system in this formalism is composed of concurrently running processes synchronized to a single global clock. The local time of each process advances in lockstep with this global clock and hence with every other process. Halbwachs [1993] provides an overview of this class of languages; Benveniste et al. [2003] describe how these languages have evolved over the last 12 years.

Verilog, the only language in this list specifically designed to model hardware, uses the discrete-event model of concurrency. Intended to model concurrent events that take prescribed amounts of time, the discrete-event model also assumes a single global clock, but unlike Esterel and Lustre, the Verilog model is able to schedule each event at an arbitrary (simulated) time in the future. Discrete-event simulators are usually implemented with a single event queue that orders events by the simulated time at which they are meant to occur. Simulation consists of picking the most immediate event from this queue, executing it, and advancing simulated time. This model is richer than the synchronous one, but also more difficult to analyze and execute efficiently.

The literature on discrete-event simulation is vast (perhaps hundreds of books and thousands of articles) and dates back to at least the mid 1960s [Ulrich 1965, 1969]. Ironically, although discrete-event simulation is well suited to simulating concurrent systems, parallel computers are ill suited to running discrete-event simulations. This problem remains an active area of research and has been so for over 20 years [Fujimoto 1980; Madisetti et al. 1991].

The languages in this article include C and two variants. Picasso C (Section 10) represents a system as concurrently running processes running asynchronously. Each process is written in C augmented with statements for rendezvous communication (see Section 1.2). FlowC is similar, but communication between processes is done through unbounded first-in, first-out buffers (FIFOs), requiring even less synchronization. Perhaps the most unique technique in this article is based on the program dependence graph (Section 4). This finds concurrency in standard sequential C (surprisingly, some is easy to find) to expose opportunities for reordering instructions.

The final language on the list, synchronous dataflow (SDF), also models systems as concurrent processes that communicate through buffered FIFO channels, but unlike Picasso C and FlowC, places strong restrictions on how and when this communication takes place. Specifically, it forces the behavior of each process to be a sequence of firings; when a process fires, it consumes and produces the same number of data tokens on each FIFO. This is a more restrictive model, because it prohibits data-dependent communication behavior, but its simplicity enables a less expensive scheduling technique. See Lee and Parks [1995] for an overview of this philosophy.

This list includes both imperative and declarative languages. Only two of the languages, SDF and Lustre, are declarative, perhaps because they focus exclusively on the movement of data and do not attempt to define sequencing. The remaining languages (Esterel, Verilog, and C variants) are imperative and describe sequencing. Verilog also supports a mix of Lustre-like declarative dataflow and imperative C-like constructs, both of which prove useful in modeling digital hardware.

1.2 The Communication Styles

The model of communication used in each language has a substantial effect on how it can be analyzed, compiled, and otherwise manipulated. This is understandable: interprocess communication generally dictates the order in which concurrent processes must execute.

The languages here embody two communication styles. Esterel, Lustre, and to a lesser extent, Verilog and C use broadcast communication: a process sends an unaddressed communiqué (i.e., without specifying which process is meant to receive it) and the information it contains may either be ignored or retrieved by one or more processes. Assigning a variable in C or writing to a `reg` in Verilog places a value in memory that persists until changed. By contrast, the presence of an emitted signal in Esterel or the value of a signal in Lustre only persists during a single clock cycle; everything is recomputed in the next cycle.

The other group of languages, Picasso C, SDF, and FlowC, guarantees delivery of information by either waiting for the receiver before the transmitter can proceed (Picasso C) or by buffering it (SDF and FlowC). The former technique—rendezvous—is the cornerstone of Hoare's [1985] communicating sequential processes and tightly couples communication and synchronization. Rendezvous communication can be thought of as taking place through a "zero-place" buffer; that is, that the sender must wait for the receiver to grab the information before proceeding. SDF and FlowC, by contrast, assume unbounded-length buffers, meaning that a sender can place information in the buffer without waiting for the receiver to retrieve it; the data remain in the buffer until read.

The contrast between these two styles of communication is dramatic. Synchronous models tend to be finite state and lend themselves to analysis with automata-based techniques. The buffer model is superior for describing multirate behavior, that is, when different parts of the system must operate at different, possibly unrelated rates. However, this flexibility tends to result in a more challenging scheduling procedure: managing buffer sizes becomes the main focus. SDF severely restricts communication behavior to make it relatively easy to predict buffer sizes a priori, its key advantage. Both the FlowC and Picasso C compilers allow data-dependent communication behavior that must be analyzed to bound buffer sizes, a much more costly procedure.

1.3 The Techniques

Table II lists the different compilation techniques presented in this article and also serves as a table of contents. The technique in each section differs from the technique in the previous section in some fundamental way, such as its model of concurrency or source language.

Each of the techniques translates its source language into a particular concurrent intermediate representation, then uses some scheduling procedure to synthesize a sequential representation from it. Different concurrent representations produce different trade-offs among expressibility (i.e., some languages are very awkward to translate into certain representations), analysis complexity, and speed of generated code.

Table II. Compilation Techniques Described in this Article

§	Language	Technique	Concurrency Model	Sequential Model	Specialized w.r.t.
2	Esterel	V5	Logic network	Levelized network	Schedule
3	Esterel	EC	CCFG	CFG	Schedule
4	C, etc.	PDG to CFG	PDG	CFG	Schedule
5	Esterel	SAXO-RT	Event graph	Event sequence	Schedule
6	Verilog	VeriSUIF	Event graph	Looped sequence	Event queue
7	Esterel	V3	CCFG	FSM of CFGs	State, signals
8	Lustre		Equation network	FSM	Boolean variables
9	SDF		Block diagram	Looped sequence	Schedule
10	Picasso C		Petri net	FSM of CFGs	State, schedule
11	FlowC		Petri net	FSM of CFGs	State, schedule

The Esterel V5 compiler (Section 2) uses a simple concurrent intermediate representation, a combinational logic network, and levelizes it to produce a sequential representation that amounts to a sequence of assignments. This approach admits a simple scheduling procedure because of its easily analyzed concurrent representation, but it does not generate very efficient code because combinational networks are difficult to simulate efficiently on sequential processors.

To produce more efficient code, the EC Esterel compiler (Sectoin 3) uses a concurrent intermediate representation whose semantics are closer to that of sequential processors: a concurrent control-flow graph. The compiler then translates this to sequential code by abstractly simulating it and adding code that simulates context switches. Although this produces much faster code than the V5 compiler, the procedure is more complex and the generated code has more overhead.

The program dependence graph (PDG, see Sectoin 4) is an even more abstract representation of concurrent control- and dataflow. Its synthesis procedure is very complicated—it determines implication relations among nodes by propagating information around the graph—but the representation allows many clever program transformations.

The concurrent representations used by the the V5, EC, and PDG techniques each contain mechanisms for scheduling something to happen immediately (i.e., soon after the scheduling event), but the Esterel language also has the ability to schedule events in both the current clock cycle through normal sequencing and in the next cycle through statements such as *pause* that explicitly consume clock cycles. The event graph in the SAXO-RT compiler (Sectoin 5) unifies these two types of scheduling actions in a single graph. The generated code maintains a scoreboard that records which events (short code fragments from the source program) remain to be executed in the current cycle and which are scheduled for the next. Execution consists of stepping through this scoreboard in an order determined at compile time.

The Verilog language has the ability to schedule events at arbitrary times in the future. Such discrete-event models are usually simulated with a time-ordered event queue, but for efficiency the VeriSUIF compiler (Section 6) attempts to statically determine the behavior of this event queue to eliminate as many run-time decisions as possible. Although few models behave simply enough to allow this to be done completely, most simulations of digital hardware

are highly periodic due to the presence of periodic clocks driving computational elements with fixed delays. In the end, VeriSUIF generates code containing an initialization sequence followed by nested loops that terminate at the end of the simulation.

The VeriSUIF compiler deliberately keeps the automaton it generates simple to speed the analysis process and keep the size of the generated code under control. It treats each event independently and models each as being present, absent, or unknown and handled at run-time. Although this appears necessary for a discrete-event formalism, the simpler synchronous models make it feasible to generate more complicated automata by tracking more information at compile-time.

The Esterel V3 compiler and its derivatives (Section 7) track the points at which a program may pause between cycles, which can often produce complicated, but still reasonably sized automata. This simulation is exact in the sense that combinations of program counter values are considered (VeriSUIF considers everything to be independent). The more complex automata lead to faster code, but they may be exponentially larger than the source program, which is often impractical.

Because Lustre is a dataflow language that does not have explicit program counters, the Lustre compiler (Section 8) instead tracks Boolean-valued signals. This can produce even more efficient code at the possible expense of even larger automata. The Lustre compiler employs on-the-fly state minimization to keep this process under control, but the number of states can still grow exponentially.

The remaining three compilation techniques (SDF, Picasso C, and FlowC) operate on formalisms that permit asynchronous behavior; Esterel, Lustre, and Verilog all require their concurrent processes to proceed in lockstep. Such an assumption makes them easier to analyze, but some problems require parts of a system to execute at different rates.

The SDF formalism (Section 9) requires processes to execute at rates that are fractional multiples of each other. For example, an SDF process might execute at four-fifths the speed of another. Although restrictive, many multirate signal processing applications fit perfectly in this framework. SDF's big advantage is that it is easy to schedule statically: execution rates are established at compile-time and used to determine a repeating schedule for the system.

Both Picasso C and FlowC use the very flexible Petri net model (described in more detail below) as an intermediate representation and explore the state space of their systems to produce a schedule in the form of an automaton. Picasso C uses rendezvous communication, so buffer sizes are not an issue; FlowC processes communicate through unbounded buffers, so it also must analyze buffer capacities. An interesting challenge in these formalisms is choosing the boundaries of the automaton's states: Picasso C splits the state space when loops would occur; FlowC breaks states at points where a decision is necessary.

1.4 The Concurrent Models

The compilation techniques described in this article employ a variety of concurrent intermediate representations. Most languages can be represented in many different ways; for example, four compilers for Esterel each use a different representation. Each such representation is a particular abstraction of the source language; each simplifies certain types of analysis and complicates others.

A dataflow network may be the simplest concurrent representation. The combinational logic network used in Esterel V5 (Section 2) and the equation network in the Lustre compiler (Section 8) are similar: both represent a system as a directed graph whose nodes represent functions and whose edges represent data dependencies. Such networks have simple semantics: the behavior of a system comes from evaluating each node in topological order (i.e., respecting data dependencies). Such representations are easy to generate and analyze, but are a poor match for the semantics of a typical sequential processor, leading to either slow code (e.g., from Esterel V5) or very aggressive analysis (e.g., Lustre's combined state space exploration and minimization algorithm). Both formalisms prohibit cycles in the dependency graph to guarantee their systems are functions instead of constraints.

Synchronous Dataflow (SDF, Section 9) extends the dataflow representation by allowing multirate behavior; that is, instead of each node being evaluated exactly once per cycle, each is evaluated some fixed number of times to produce a stream of data instead of a single value. This, coupled with allowing nonzero-delay cycles in the dependency graph, makes scheduling and executing the representation more complicated, although all decisions can still be made at compile-time for efficiency.

A disadvantage of these dataflow specifications, especially when implementing them on sequential processors, is the absence of any notion of control. Every part of these dataflow systems runs at all times, which may not be necessary.

Control flow enables parts of a system to control whether other parts are executed. A decision is the most basic form: an input or state of the system may lead to part of the system being inactive temporarily. Decisions are particularly efficient on sequential processors because inactive portions of a program can be skipped with almost no cost.

Control flow is usually represented by a flowchart or control-flow graph, a directed graph whose nodes represent actions and whose arcs represent how control is passed between actions. During execution, control is always performing exactly one action. When that action terminates, control is passed along exactly one outgoing arc to another action.

The concurrent control-flow graphs employed in the V3 and EC Esterel compilers extend the basic control-flow graph by allowing control to be at multiple places at once and allowing control to pass along more than one outgoing arc from special fork nodes. The concurrent control-flow graph in the V3 compiler (called IC, Intermediate Code, and also used by EC as a starting point) is even richer, adding more complex synchronization nodes and state-holding primitives to the mix.

Control-flow graphs are a near-perfect match to the behavior of sequential processors, making their translation into sequential code efficient. Adding concurrency and other constructs makes the translation more difficult, but the results are generally better than dataflow models.

```
while the system runs do
  Select a transition that may fire: it must have a token in each of its predecessor places.
  Remove the tokens from its predecessor places.
  Place tokens in each of its successor places.
```

Fig. 1. A Petri net interpreter. Many transitions are often enabled at once, so an interpreter usually has many choices about how to proceed.

Event graphs are still more complex models that provide more complex control behavior. The event-graph in SAXO-RT (Section 5) includes arcs that both activate and deactivate nodes in both the current cycle and the next. The VeriSUIF event graph (Section 6, Figure 14(b)) includes arcs that activate after a delay and those that cause partial activation (i.e., for certain nodes to gain control, each must be activated by two incoming arcs).

Petri nets are perhaps the most flexible concurrent representation known. More flexible than control-flow graphs (Petri nets subsume them) and less ad hoc than event graphs, Petri nets have developed a vast literature and many formal analysis techniques. A good starting point is Murata's [1989] heroic 40-page, 315-reference survey. Petri [1962] started it all.

A Petri net is a bipartite directed graph whose nodes are either transitions or places. Its semantics are deceptively simple (Figure 1): the state of the system (a marking) is the number of tokens in each place. In each state, a transition may fire if all of its predecessor places have at least one token. When a transition is fired, it removes tokens from each of its predecessor places and adds one to each of its successor places to generate the next state. A Petri net is nondeterministic when more than one transition can be fired: the semantics say any choice is valid.

The Petri net formalism is capable of describing sequential, concurrent, buffering, and rendezvous behavior, so it is a natural representation for procedural code that communicates through buffers or with rendezvous communication such as Picasso C and FlowC. Transitions represent actions and places represent control points between instructions.

The synchronous dataflow model can be thought of as a special case of Petri nets; many results such as the compilation technique of Lee and Messerschmitt [1987b] based on solving balance equations and state-space exploration for scheduling are special cases of more general theorems for Petri nets. However, it is not fair to say work on SDF is completely subsumed by Petri net theory. Petri nets are almost too general a model, and can be nearly as unwieldy as, say, Turing machines. Most interesting results have come from restricting the general Petri net model to make it more tractable; synchronous dataflow and its variants should be considered important restrictions of Petri nets.

1.5 The Sequential Models and Partial Evaluation

The sequential models generated by the compilers described in this article all generate automata or variants thereof. This is natural because automata can be very efficiently converted into machine instructions for sequential processors and are also a universal way to represent finite-state behavior. None generate infinite-state objects such as pushdown automata; this follows

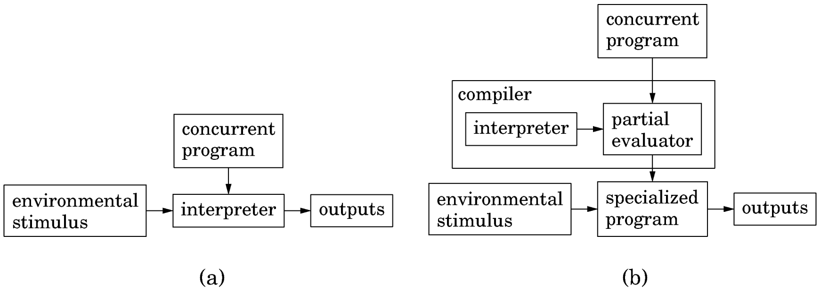


Fig. 2. The flow (a) of an interpreter and (b) of a partial evaluator acting as a compiler.

from the finite-state nature of the input languages (the buffered FIFOs of SDF and FlowC are required to be bounded in practice). Scheduling infinite-state concurrent representations into infinite-state sequential ones would probably be intractable.

Many of the automata generated are particularly simple. For example, the Esterel V5 compiler (Section 2) generates linear automata by topologically sorting the data dependencies; each automaton state evaluates a single logic gate. An automaton generated by the VeriSUIF compiler (Section 6) is only a little more complicated: it consists of a linear sequence of initialization actions followed by nested loops of ongoing actions.

Most of the compilers generate much more complicated automata by recording the results of approximate simulation, during which part of the system's overall state (e.g., certain variables) is tracked, but other parts of the state are treated as only knowable at run-time.

Approximate simulation to generate an automaton is more properly considered a form of partial evaluation, a technique developed in the compiler community for producing more efficient code (see Jones et al. [1993] for an introduction). Shown in Figure 2(a), a normal interpreter, which could be used to execute any of the languages in this article, takes a concurrent program and its environmental stimulus to produce outputs. A partial evaluator (Figure 2(b)) takes two programs, the concurrent program and an interpreter for it, and produces a third, which can be thought of as a compiled version of the concurrent program.

In theory, a partial evaluator applied to a fixed interpreter behaves exactly as does a compiler. Some, such as Hudak [1998], propose this as a general mechanism for more quickly building compilers, especially for "little" languages expected to have a limited user community, and some report success in applying this technique [Thibault et al. 1999].

Although few of the compilers described here contain components that are clearly interpreters or partial evaluators, conceiving of each compiler as consisting of these two components is useful as it makes it possible to conceptually separate the operational semantics of the concurrent specification (i.e., the interpreter) from how it is compiled (e.g., the partial evaluator). This clarifies how parts of the techniques could be combined to produce new compilers.

```
tick(s1, s2) {
  switch (s1) {
  case 1:
    f1(); ns1 = 2;
    break;
  case 2:
    f2(); ns1 = 1;
    break;
  }
  switch (s2) {
  case 1:
    f3(); ns2 = 2;
    break;
  case 2:
    f4(); ns2 = 2;
    break;
  }
  return (ns1, ns2);
}
```

(a)

```
tick11() {
  f1();
  f3();
  return (2, 2);
}

tick22() {
  f2();
  f4();
  return (1, 2);
}

tick12() {
  f1();
  f4();
  return (2, 2);
}
```

(b)

```
tick11() {
  f1();
  f3();
  return tick22;
}

tick22() {
  f2();
  f4();
  return tick12;
}

tick12() {
  f1();
  f4();
  return tick22;
}
```

(c)

Fig. 3. Using partial evaluation to speed execution: (a) a simple function whose inputs are its present state and its output is the next state; (b) the results of partially evaluating the function with respect to the state, propagating constants, and removing dead code. For example, `tick11` implements the function when s1 = 1 and s2 = 1; (c) after reencoding the state as a function pointer.

The partial evaluation techniques used implicitly in these compilers are more powerful than general approaches because they have been tuned to a particular interpreter. Such specialization enables optimizations that a general-purpose partial evaluator would be hard-pressed to find because they require a deep understanding of the semantics of the concurrent language. Furthermore, these partial evaluators can rise above interpreter implementation details and concentrate on the semantics of the language.

The compilers differ greatly in how aggressively they partially evaluate the system. Some (e.g., the SAXO-RT compiler in Section 5) generate little more than hardwired simulators, whereas others attempt to remove all traces of the interpreter (e.g., the PDG-to-CFG (control-flow graph) translation algorithm in Section 4) and generate very clean code. How much of the interpreter's state is analyzed during compilation is the main differentiator. Analyzing more of the state usually makes for faster code, but can also cause code bloat and longer compilation times inasmuch as more cases must be explored.

The most common optimization determines a program's schedule at compile-time. Depending on the language, scheduling decisions can be quite costly and thus are excellent candidates for being performed beforehand.

Figure 3 illustrates how partially evaluating a program with respect to state variables can improve its execution speed. Many of the compilation techniques presented here use this general technique to generate efficient code. Figure 3(a) is the original (sequential) program that uses two state variables s1 and s2 to control its operation. Partially evaluating the program with respect to these two variables while noting that the program can only reach certain combinations of them produces the three functions in Figure 3(b). This representation still explicitly passes around the values of the two variables, but this is probably unnecessary. Reencoding their values with function addresses as in Figure 3(c) produces even faster code. Here each function returns the address of the next function to call.

2. COMBINATIONAL LOGIC NETWORKS IN ESTEREL V5

Every compiler described in this article works by translating the high-level concurrent semantics of the source language into a simpler concurrent intermediate format that is later translated into a sequential mode. It is simpler to take two steps (complex concurrency to simple concurrency, then simple concurrency to sequential) than to perform the entire conversion at once.

The compiler presented in this section, the Esterel V5 compiler by Berry et al. [1992, 1999], uses perhaps the simplest concurrent model of any presented in this article and hence makes a good starting point. Beginning with the complex synchronous model of concurrency in the Esterel language, this compiler first creates a combinational logic network that represents the Esterel program's behavior, then generates sequential code for this using a simple topological-sort-based scheduling technique.

This approach has mixed success. On the positive side, it scales very well because the size of the logic network is roughly proportional to the length of the source and its simple semantics greatly simplify the complex algorithm necessary to handle a difficult class of Esterel programs (those with false dependency cycles). For these reasons, the V5 approach is currently the most robust. However, the code it generates can be vastly slower than that from other Esterel compilers because it is difficult for a sequential processor to efficiently simulate a logic network.

2.1 Esterel

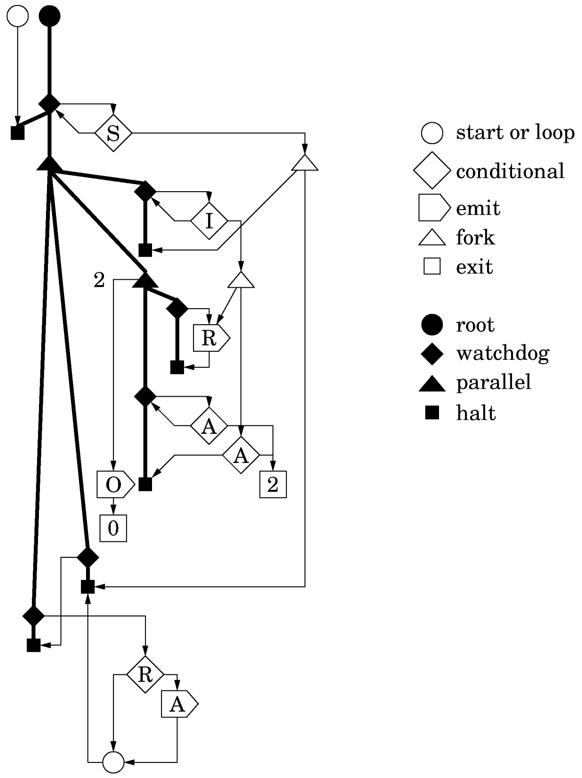
The Esterel language [Berry and Gonthier 1992; Berry 2000] is a procedural concurrent language that describes systems synchronized to a single global clock. In each clock cycle, each sequential process resumes where it paused in the last clock cycle, communicates with other processes and the environment through unbuffered signals, and suspends until the next cycle. Signals behave as wires: they are either present or absent each cycle and their values do not persist between cycles.

Figure 4(a) 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 arrives, at which point it emits the O signal and terminates. The second thread emits R in response to A in alternate cycles.

Figure 4(b) shows how this Esterel program is represented in the IC ("intermediate code") format, a control-flow graph (outlined nodes and arcs) hanging from a reconstruction tree that handles Esterel's concurrency and preemption

```
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 present
    end loop
  end every
end signal
end module
```

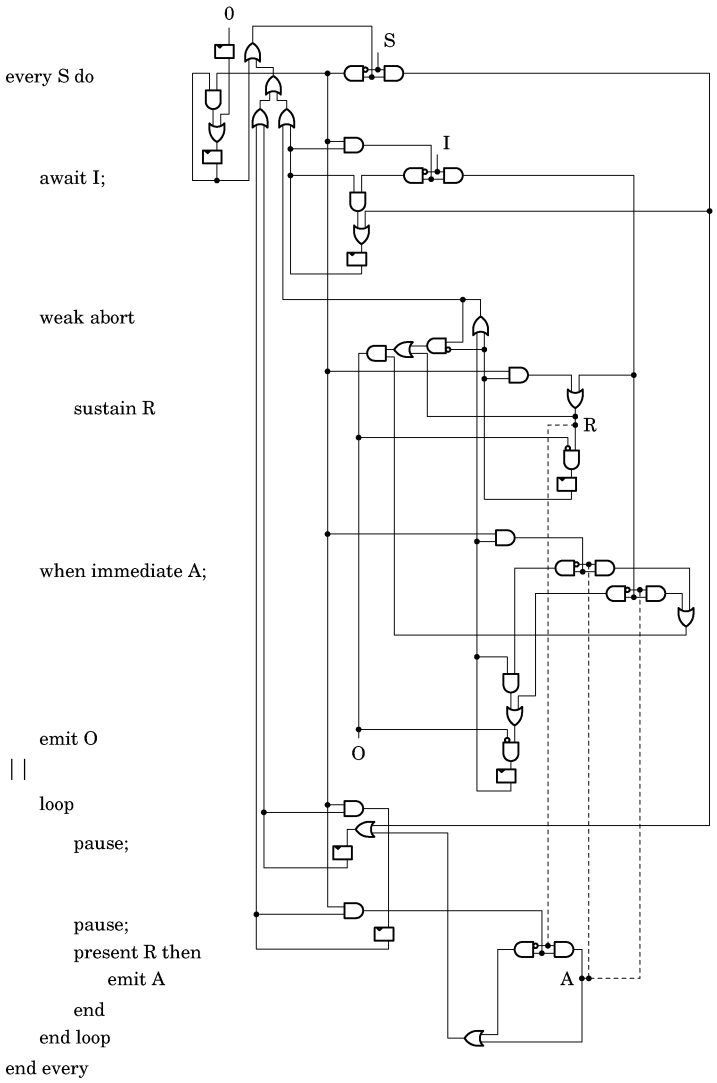
(a)



(b)

Fig. 4. (a) A simple Esterel module modeling a shared resource. The first thread generates requests in response to external requests, and the second thread responds to them in alternate cycles. The S input resets both threads. (b) The IC graph for the program. 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.

(solid nodes and arcs). Originally designed by Gonthier [1988] for the automata-based V3 compiler (Section 7), the IC format operates in three phases in each cycle. In the first phase, control starts at the root node and works its way toward all halt nodes that were reached at the end of the last cycle (this set of halts encodes the state of the program between cycles). Control splits at fork nodes to restart concurrent threads. Preemption conditions (e.g., *every* S) are tested along the way and may send control elsewhere. In the next phase, preemption conditions have been checked and the threads have resumed running: control-flows through the control flow portion of the IC graph. Eventually, control reaches halt or exit nodes and flows back up the reconstruction tree, checking for termination or exceptions (e.g., *weak abort*). Eventually control returns to the root node to indicate the program is done for the cycle.



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

Fig. 5. The circuit the V5 compiler generates for an Esterel program represented by the IC graph in Figure 4(b). Registers implement halts, points where the program will restart in the next cycle. Dashed wires communicate signals; all others represent control. The unusual layout of this circuit follows the structure of Figure 4(b). Removing the registers leaves the circuit acyclic.

2.2 Translating Esterel into Logic

The semantics of the IC format are complex (see Edwards [2002] for some details) but can be translated fairly directly into a circuit such as Figure 5. Berry [1992, 1999] explains how. To understand this circuit, begin at the latches (the rectangles). Each represents a pause statement, and its output is one if control reached it in the last cycle. The outputs of these latches feed into a tree of OR gates (the selection tree) whose structure matches the reconstruction tree, eventually forming the activation condition for the whole program. This output

```
E[0] = R[2] || R[3];
E[1] = R[1] || E[0];
E[2] = R[4] || R[5];
E[3] = E[1] || E[2];
E[4] = R[6] || E[3];  /* top of the reconstruction tree */
E[5] = E[4] && !S;
R[6] = R[0] || (E[5] && R[6]);  /* reset latch */
E[6] = R[1] && E[5];
E[7] = E[6] && I;
E[8] = R[5] && E[5];
E[9] = E[7] || (R[3] && E[5]);
E[10] = E[8] && E[9];
E[11] = E[7] && E[10];
E[12] = R[2] && E[5];
E[13] = E[12] && E[10];
E[12] = (E[7] && !E[10]) || (E[12] && !E[10] && R[2]);
E[7] = (E[0] && !R[2]) || E[12];
E[0] = (E[0] && !R[3]) || E[9];
E[13] = (E[11] || E[13]) && (E[7] || E[11] || E[13]) && E[0];
if (E[13]) emit_O();
```

Fig. 6. A fragment of the code generated by the V5 compiler for the circuit in Figure 5. This comes from generating code for each gate in the circuit ordered by a topological sort of the gates. The R array contains register values that persist between cycles; the E array holds temporary intermediates.

activates the S conditional (a pair of AND gates), which either restarts the two threads or heads back left and then down the reconstruction tree.

A watchdog, such as the one just before the test for I, is implemented with an AND gate activated when latches below it are set and control has come back down the reconstruction tree.

The circuitry near each latch can become complicated because it must handle control from the reconstruction tree, control from normal nodes, and a kill signal generated when its thread is preempted. The latch to the right of the O output illustrates this. First, an AND gate controls the reconstruction entry to the halt. This is ORed with a sequential control wire from the other test of signal A. Finally, an AND gate with an inverting input disables the latch when its parallel has been preempted (i.e., when A is present).

Generating code that simulates such a circuit is straightforward: Figure 6 shows such a fragment. It is nothing more than code for each gate listed in topological order.

2.3 Translation Issues

An IC graph can usually be translated one-to-one into circuitry, but any IC node executed more than once in a cycle must be duplicated. This occurs in so-called schizophrenic code, such as Figure 7. The algorithm for this, developed by Berry [1999], performs a depth-first walk of the IC code that visits each node once per reincarnation.

The synthesis procedure produces a redundant circuit. For example, the selection tree is redundant because at least one of the latches is always set (the program never terminates). A traditional area-optimizing logic synthesis pass

```
loop
  trap T in
    loop
      present A then emit B end; pause
    end
  ||
    pause; exit T
  end
end
```

Fig. 7. Schizophrenic code. The *present A* runs twice in the second cycle: once when the inner loop is restarted, once after the *exit* restarts the outer loop.

(see Hachtel and Somenzi [1996] for an introduction) can often remove half of the circuit because of such redundancies.

What happens when the generated circuit is cyclic? This can only be caused by communication dependencies since Esterel requires an acyclic control-flow graph (loops must always contain a pause) and schizophrenia is removed by unrolling the circuit. Cyclic communication dependencies can arise in nonsensical programs, such as

```
present A else emit A end
```

Here, A is present only if it is absent: a contradiction. This corresponds to an inverting loop in hardware that would probably oscillate.

Some cyclic circuits, such as token-ring arbiters, are well-behaved, useful, and easily expressed in Esterel. Such arbiters are examples of statically unstable circuits that are dynamically stable. The presence of a token makes the circuit stable; the circuit cannot reach any unstable state. Such a program is considered correct, but showing this requires knowledge of all the system's reachable states, a costly computation.

Berry's gate-based compilers analyze cyclic circuits by exploring their state spaces symbolically. This algorithm, developed by Shiple et al. [1996], uses Binary Decision Diagrams [Bryant 1986] to represent the circuit and the set of states that have been reached. At each step, the algorithm checks the stability of the circuit in the states reached so far using three-valued simulation (i.e., using the values 0, 1, and unknown). Once all the reachable states are identified and the circuit is known to be stable in all of them, the cyclic portion of the circuit is resynthesized to remove the cycles. Although this procedure can be costly, it is the only known technique for handling large cyclic systems.

In the partial evaluation framework, the V5 compiler is trivial; the only challenge comes in translating Esterel into a combinational logic network. Once it has been translated, the interpreter is straightforward (evaluate gates in topological order), and the partial evaluator is similarly simple (generate code to evaluate each gate). This simplicity suggests there is room to improve this technique.

The relatively low speed of the generated code (as much as 100 times slower than other approaches; see Edwards [2002]) is the biggest disadvantage of translating control into dataflow. The main problem is that the generated code must do something in every cycle for each program statement, even those that

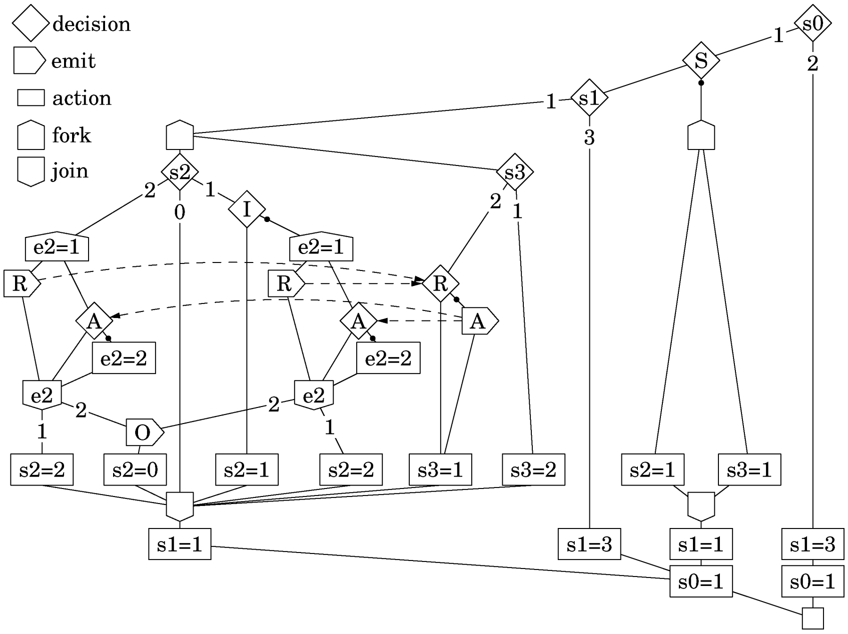


Fig. 8. The concurrent control-flow graph the EC Esterel compiler generates for the program in Figure 4(a). Dashed lines represent data dependencies. Variables s0, s1, s2, and s3 store state between cycles; e2 holds the exit level of the group of threads. Initially, s0=2 and all other variables are uninitialized.

do not run in that cycle. Unlike a traditional software compiler, code generated from a circuit does not skip over inactive portions of the circuit (e.g., the untaken branch of a conditional) because zeros that indicate statements do not run are propagated.

3. CONCURRENT CONTROL-FLOW GRAPHS IN ESTEREL EC

Although the logic netlist representation used in the V5 compiler described in the last section is elegant and simple, its semantics are not a good match for sequential processors. In particular, it is pure dataflow; every part of the circuit/program is assumed to be active and running at all times. This is true for digital hardware, but is emphatically not true for a sequential processor, which is only able to perform a few operations at any time. As a result, the generated code is slow because much time is spent evaluating gates that correspond to idle portions of the program.

My EC compiler [Edwards 2000] translates an Esterel program into a concurrent control-flow graph (CCFG, Figure 8), which attempts to be a better compromise between Esterel's concurrent semantics and a processor's sequential semantics. EC translates an Esterel program into a CCFG, schedules the nodes in that graph based on control and data dependencies; and walks through the graph node by node, generating sequential code that saves and restores state when the system "context switches" between concurrently running threads.

The result is a sequential control-flow graph (Figure 9(a)) that can easily be translated into procedural code (Figure 9(b)).

Handling threads that must be interleaved because they communicate back and forth within the same cycle is the main challenge. Signals R and A in Figure 4(a) behave this way and require the first thread to execute partially, relinquish control to the second thread, and return to the first thread.

EC handles these situations by inserting code that performs context switches. In general, a context switch consists of writing a constant to a variable representing the program counter of the thread being suspended followed by a multiway branch on the program counter of the thread being resumed. There are three such context switches in Figure 9(a).

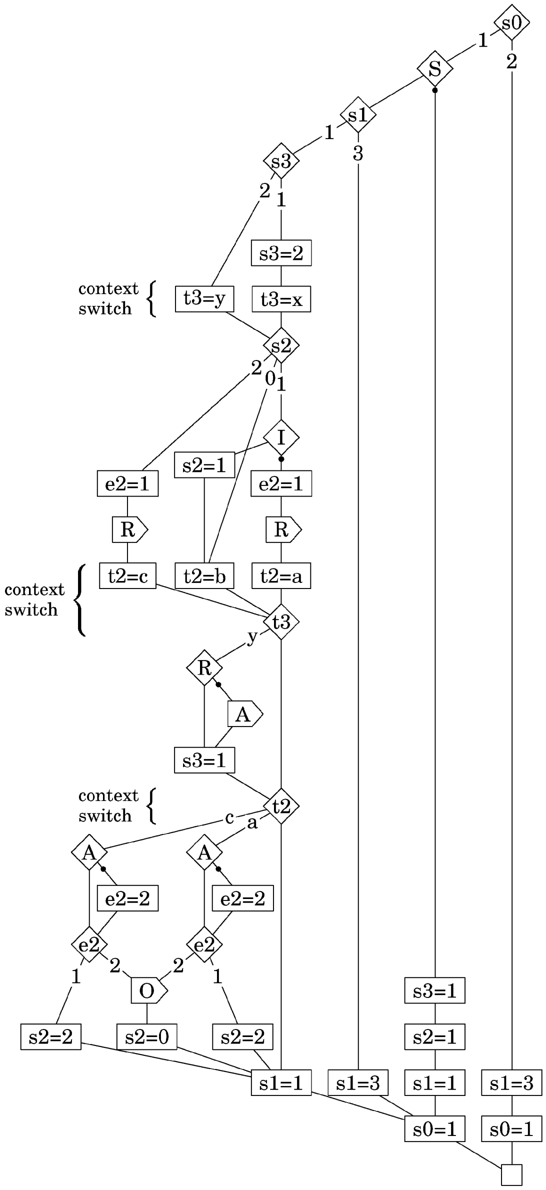
The first part of the compiler translates IC (Figure 4(b)) into a CCFG (Figure 8). The main challenges here are inserting conditionals to handle the semantics of the reconstruction tree and unrolling the graph to avoid loops. The reconstruction tree and the predicates it tests are translated into identically structured control-flow. Branching points in the reconstruction tree become two- or multiway conditionals that test groups of bits in their thread's program counter variable.

Using the same routine as the V5 compiler, EC unrolls the IC graph to avoid a cyclic graph when statements are reincarnated. Although the Esterel language prohibits single-cycle loops, certain instructions can be executed more than once in a cycle, which would lead a simple-minded translation algorithm to generate a loop. The unrolling procedure judiciously duplicates such nodes. Unrolling the example in Figure 4(a) duplicates the nodes that emit R and test A.

Once the CCFG has been generated, EC then schedules its nodes, respecting control and data dependencies. A schedule is simply a topological sort of the CCFG augmented with data dependencies. Although finding an optimal schedule (i.e., one that minimizes overhead) appears to be NP-hard, a bad schedule does not cause the code generation procedure to generate substantially worse code (i.e., the cost is at most quadratic). EC uses a simple-minded depth-first search to generate the schedule.

Generating a control-flow graph (CFG) from a scheduled CCFG is the most complicated algorithm in the EC compiler. It steps through the nodes in scheduled order and in effect simulates execution of each. In general, simulating a node consists of copying it and attaching control-flow arcs from all its predecessors. Context switches are more complicated. A context switch is simulated by attaching to each node that just ran in the thread being suspended a newly created suspension node that saves the state of the thread and jumps to a multiway branch that effectively recovers the state of the thread being resumed. The middle group of nodes labeled "context switch" in Figure 9(a) illustrates this. Three nodes write different constants into the thread state variable t2 then jump to a conditional that tests the thread state variable t3.

A CFG is fairly easy to translate into C source. Generating a program filled with *goto* statements is trivial, but this is hard to read and debug. Instead, EC attempts to generate structured code by using the immediate postdominator (see Lengauer and Tarjan [1979]) of each conditional to determine when to



(a)

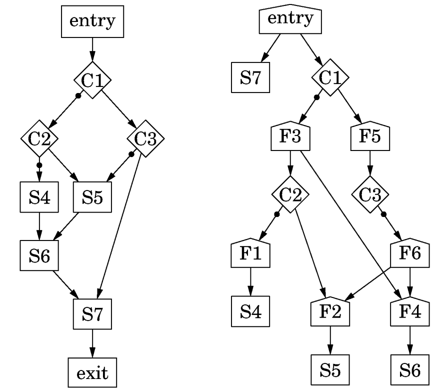
```
#define a 24
#define b 19
#define c 35
#define x 19
#define y 46
if ((s0 & 3) == 1) {
  if (S) {
    s3 = 1; s2 = 1; s1 = 1;
  } else
    if (s1 >> 1) s1 = 3;
    else {
      if ((s3 & 3) == 1) {
        s3 = 2; t3 = x;
      } else t3 = y;
      switch (s2 & 3) {
      case 0: goto L1;
      case 1:
        if (I) {
          e2 = 1; R = 1; t2 = a;
        } else {
          s2 = 1;
        L1:
          t2 = b;
        }
        break;
      case 2:
        e2 = 1; R = 1; t2 = c;
        break;
      }
      if (t3 == y) {
        if (R) A = 1;
        s3 = 1;
      }
      switch (t2) {
      default: break;
      case a:
        if (A) e2 = 2;
        if (e2 == 2) goto L2;
        s2 = 2;
        break;
      case c:
        if (A) e2 = 2;
        if (e2 == 1) s2 = 2;
        else {
        L2:
          O = 1; s2 = 0;
        }
        break;
      }
      s1 = 1;
    }
  s0 = 1;
} else {
  s1 = 3; s0 = 1;
}
```

(b)

Fig. 9. (a) The sequential control-flow graph the EC Esterel compiler generates for the program in Figure 4(a), and (b) the code generated from it. Three context switches (tests and assignments of t2 and t3) were introduced to interleave the execution of the threads. The symbols a, b, c, x, and y are arbitrary constants.

```
if (C1)
  if (C2)
    S4;
  else
    L: S5;
  S6;
else
  if (C3)
    goto L;
S7;
```

(a)



(b) (c)

C1C2C3	F2F4
0 0 0	1 1
0 0 1	1 1
0 1 0	0 1
0 1 1	0 1
1 0 0	0 0
1 0 1	1 1
1 1 0	0 0
1 1 1	1 1

(d)

Fig. 10. (a) A short procedure; (b) a control-flow graph for it. True branches are labeled with black dots; (c) a program dependence graph for it; (d) a truth table listing the conditions under which F2 and F4 run. Because F4 always runs when F2 does, code for S5 must appear before S6 in the CFG.

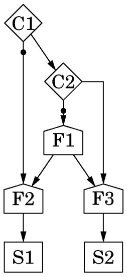
close the sequence of code in the *then* and *else* branches of an *if* statement and the cases in a *switch* statement. The result is fairly readable (Figure 9(b)).

In the partial evaluation framework, the interpreter in EC first computes a schedule for the nodes in the program, then simulates it by stepping program counters through each thread. The partial evaluator steps through this schedule once, noting where the generated code must explicitly manipulate these program counters.

4. PROGRAM DEPENDENCE GRAPHS

Like the concurrent control-flow graph in the EC compiler in the last section, Ferrante et al.'s [1987] Program Dependence Graph (PDG) is another, more abstract representation of concurrent control and data dependencies that strives to be a good match for sequential processors. Originally designed as an intermediate representation for optimizing compilers for sequential languages such as C, the PDG (e.g., Figure 10(c)) was designed to remove all nonessential control dependencies to expose every possible opportunity for instruction reordering and instruction-level parallelism. For example, a sequence of assignments with no data dependencies is represented as running concurrently in a PDG. This section focuses on the algorithms for translating a PDG into a sequential control-flow graph with equivalent behavior.

A PDG (Figure 10(c)) is a rooted graph whose nodes represent statements, conditionals, and forks, and whose arcs represent control and data dependence. The discussion here focuses on control dependence, which is the more challenging of the two. A fork node executes all its children in arbitrary order. A node is executed if there is a control path in the PDG from the entry to the node that is consistent with the conditional statements along that path; for example, if a conditional on the path is true, the path must go through its true child. As an illustration, in Figure 10(c), S6 is executed if C1 is true (path: entry → C1 →



(a)

C1	C2	F2	F3
0	0	0	1
0	1	1	1
1	0	1	0
1	1	1	0

(b)

```
if (C1) {
  S1;
} else {
  if (C2) {
    S1;
  }
  S2;
}
```

(c)

Fig. 11. (a) A PDG with no concise CFG; (b) a truth table listing the conditions under which F1 and F2 execute. Because F1 and F2 can execute both alone and jointly, no concise CFG exists; (c) one possible implementation. Note that the code for S1 is duplicated.

F3 → F4 → S6), or if C1 is false and C3 is true (path: entry → C1 → F5 → C3 → F6 → F4 → S6).

Existing algorithms for translating PDGs into CFGs attempt to construct concise CFGs, that is, ones in which each PDG node appears exactly once. This is ideal because it means the concurrency can be simulated with no overhead. For example, Figure 10(c) has the concise CFG in Figure 10(b). When a concise CFG exists, an n-node PDG with e edges can be synthesized in $O(ne)$ time, but implementing other PDGs, such as Figure 11(a), requires code to be duplicated or guards to be added. Finding a minimal CFG when no concise one exists appears to be NP-hard, although heuristics exist. This problem is not well studied.

The challenge in synthesizing a concise CFG is determining the order in which a fork's subgraphs should execute based on constraints imposed when only some subgraphs may execute. Although all the subgraphs under a fork node run when a path to the fork is active, under other conditions only certain subgraphs may execute if there are outside edges entering the subgraphs. For example, if C1 is false and C3 is true in Figure 10(c), F6 is active and both F2 and F4 run. However, if C1 and C2 are true, only F2 runs. Such a relationship imposes an order on the nodes in a concise CFG for the following reason. In a concise CFG for Figure 10(c), there must be some point from which S5 and S6 are both executed with no intervening conditionals to handle the case when C1 is false and C3 is true. Furthermore, if S6 came before S5, executing S6 would always execute S5, which is incorrect when C1 and C2 are true. Thus S6 must follow S5.

Enumerating the conditions under which the children of a fork run can establish their order. For example, Figure 10(d) is a truth table listing the conditions under which F2 and F4 run. Code for F2 must come before F4 because F2 running implies F4 will run. By contrast, Figure 11(b) shows there are cases in Figure 11(a) where F2 and F3 run both separately and together. This shows there is no implication relationship, and hence no concise CFG exists for the PDG in Figure 11(a). Such brute-force analysis is exponential, but fortunately there is a more efficient algorithm.

Simons and Ferrante [1993] presented the first efficient $O(ne)$ algorithm to determine these orders. Steensgaard [1993] later extended it to handle irreducible flowgraphs, those with multiple-entry loops. Both versions compute for each node n the set of nodes that are always executed when any descendant of n executes. Specifically, a node e is in the set for n if e has a parent that is a fork node along a path from the entry to any descendant of n. These algorithms use a complicated two-pass bit-vector analysis to compute these sets. The type of each child and its membership in these sets is then used to establish constraints between children of a fork node. Steensgaard summarizes these rules in a table.

Once the order of fork node children is established, the CFG is synthesized starting from the statements and working up. The arcs from a conditional are simply connected to the nodes for their children. Synthesizing the code for a fork is the only complicated operation: the subgraph for each child is constructed and arcs are added from each statement missing a successor in one child to the first node in the next. For example, to reconstruct Figure 10(b) from Figure 10(c), the subgraph at C1 is synthesized first, then S7. Arcs are then connected from the false branch of C3 and from S6 to S7.

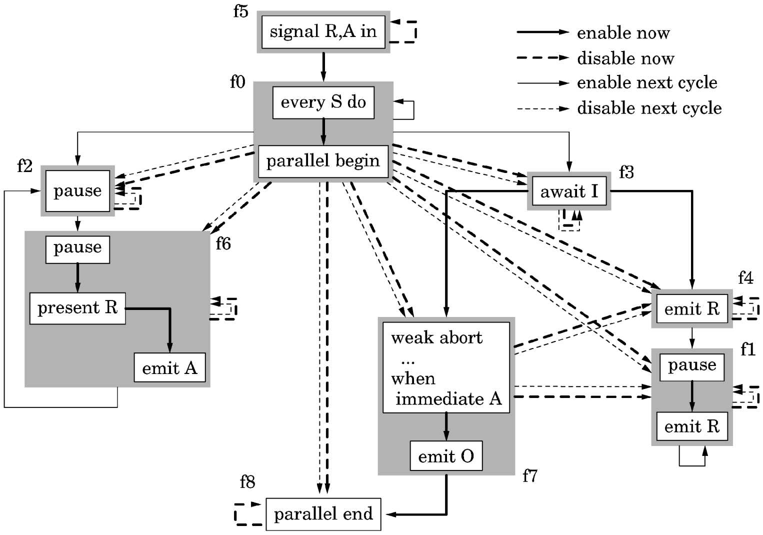
In the partial evaluation framework, the interpreter for a PDG is trivial except for the scheduler. The partial evaluator is the algorithm for building the CFG, which takes the unusual step of working backwards through the program.

5. EVENT GRAPHS IN SAXO-RT ESTEREL

The control and data dependencies in the concurrent representations described in the last three sections can only control what happens more-or-less immediately. But languages such as Esterel also provide mechanisms for scheduling things in the future. For example, although Esterel's sequencing operator (the semicolon) causes its second statement to be executed immediately after the first, the *pause* statement delays the execution of the succeeding statement to the next statement.

The event graph representation used in the SAXO-RT Esterel compiler developed at France Telecom R&D [Bertin et al. 1999; Weil et al. 2000] allows instructions to be scheduled and removed from the schedule in both the current cycle and the next. SAXO-RT builds an event graph for an Esterel whose nodes are small segments of code that can execute atomically (i.e., not crossing a *pause* or signal test) and whose arcs represent four types of control dependence. The compiler orders the nodes according to control and data dependencies and a small function is generated for each. The main program (the `tick` function) consists of a hard-coded scheduler that calls each function in turn if it is currently active.

Four types of control dependence can occur between nodes: enabling and disabling in the current and next cycle. Enabling in the current cycle is easiest to understand. Referring to Figure 12, if signal I is present and node f3 is active (runs), the *weak abort* and *sustain R* instructions should run in the same cycle (*sustain R* has been transformed into *emit R* with a self-loop). The "enable current" arcs from f3 to f7 and f4 indicate this.



```
#define F0 (1 << 0)
#define F1 (1 << 1)
/* ... */
#define F8 (1 << 8)

static unsigned int curr = F5;
static unsigned int next = 0;

static void f0() {
  if (!S) return;
  complete = 0;
  curr &= ~(F0 | F2 | F6
               | F3 | F4 | F1 | F7 | F8);
  next &= ~(F0 | F2 | F6
               | F3 | F4 | F1 | F7 | F8);
  next |= F0 | F2 | F3;
}
static void f1() {
  emit(R);
  curr &= ~F1;
  next &= ~F1; next |= F1;
}
static void f2() {
  curr &= ~F2;
  next &= ~F2; next |= F6;
}
static void f3() {
  if (!I) return;
  curr &= ~F3; curr |= F4 | F7;
  next &= ~F3;
}
static void f4() {
  emit(R);
  curr &= ~F4;
  next &= ~F4; next |= F1;
}
```

```
static void f5() {
  R = 0; A = 0;
  curr &= ~F5;
  next |= F0;
}
static void f6() {
  if (R) emit(A);
  curr &= ~F6;
  next &= ~F6; next |= F2;
}
static void f7() {
  if (!A) return;
  emit(O); complete++;
  curr &= ~(F1 | F4); curr |= F8;
  next &= ~(F1 | F4);
}
static void f8() {
  if (complete != 2) return;
  complete = 0;
  curr &= ~F8;
}
void tick()
{
  if (curr & F0) f0();
  if (curr & F1) f1();
  if (curr & F2) f2();
  if (curr & F3) f3();
  if (curr & F4) f4();
  if (curr & F5) f5();
  if (curr & F6) f6();
  if (curr & F7) f7();
  if (curr & F8) f8();

  curr |= next;
  next = 0;
}
```

Fig. 12. (top) The event graph the SAXO-RT Esterel compiler generates for the program in Figure 4(a). Each gray area is a node that becomes a single function in the generated code (bottom). Control and communication dependencies between these groups dictate the order in which they appear in the `tick()` function.

"Enable next" arcs implement the behavior of instructions such as *pause* and *await*, which wait a cycle before proceeding, by activating their targets for the next cycle. For example, when the outer *every S* statement runs, it starts the two threads that begin with *await I* and *pause*. The "enable next" arcs leading from f0 to f2 and f3 activate these statements in the next cycle. The f0 node also uses such an arc to schedule itself in the next cycle, which ensures signal S is checked in the next cycle.

By default, active nodes whose conditions are not met (e.g., f3 is active but signal I is not present) remain active in the next cycle. Thus, when a statement does run, it usually disables itself. Self-loops with disable arcs, such as those on f5, f2, and f6 accomplish this.

Preemption instructions also use disable arcs. For example, when f7 is active and signal A is present, f7 preempts its body (which contains f1 and f4) by disabling them in both the current and next cycles. Node f0, which preempts most of the program, has many such disable arcs.

The compiler encodes the event queue as a bit vector for efficiency. Each node is assigned a bit (the compiler uses multiple words when the number of nodes exceeds the processor's word size) and logical operations add and remove nodes from the queue.

The nodes are ordered based on control and data dependencies to generate the final linear schedule. For example, nodes f1 and f4 both emit the R signal, and node f6 checks it; thus f6 appears later in the schedule than does f1 or f4. Control dependencies also impose ordering constraints. Because f7 is a weak abort, which only preempts its body (f1 and f4) after it has had a chance to execute for the cycle, f7 appears after f1 and f4.

This compiler rejects programs whose nodes have no linear order. In some cases, this corresponds to a nonsensical program, such as one that says "emit this signal only if it is not present," a contradiction considered illegal in Esterel. However, some valid programs may require different instruction (node) orders in different states. The automata compilers (Section 7) allow this since they consider each state separately. The V5 compiler (Section 2) employs a sophisticated analysis and resynthesis technique that allows it to remove false ordering cycles. The SAXO-RT compiler has no such facility, but a more dynamic scheduling technique might permit it.

In the partial evaluation framework, the SAXO-RT compiler is quite straightforward. The interpreter, shown in Figure 13, is fairly simple and the partial evaluator does little more than schedule the nodes statically and generate code for each.

6. EVENT GRAPHS IN VERISUIF VERILOG

Esterel programs can only schedule operations in the current cycle and the next. By contrast, the Verilog language [Thomas and Moorby 1998; IEEE Computer Society 1996] is a much more flexible discrete-event simulation language able to schedule events far in the future.

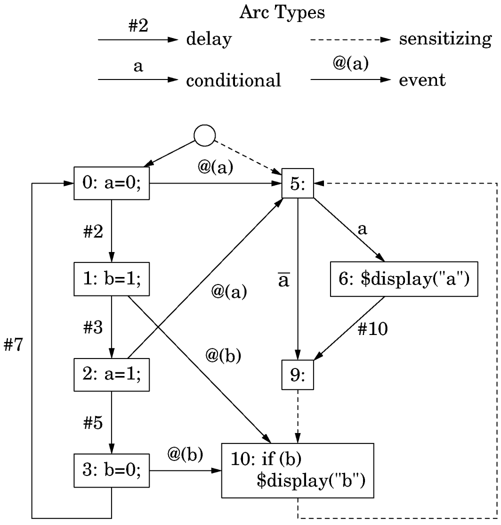
A Verilog program, which is designed to model digital hardware, consists of hierarchically composed modules, each containing instances of modules or

```
while the program is running do
  for all nodes in scheduled order do
    if the node is currently active then
      if the node's condition is met then
        execute the actions of the node
        mark all destinations of disable now arcs as inactive
        mark all destinations of enable now arcs as active
        mark all destinations of disable next cycle arcs as inactive next
        mark all destinations of enable next cycle arcs as active next
  mark all nodes active in the next cycle active this cycle
```

Fig. 13. The interpreter for the SAXO-RT Esterel compiler that is partially evaluated. The compiler generates code for each node that performs its actions and updates the current and next markings. The schedule is determined at compile-time.

```
module ex;
  reg a, b;

  always begin
    a = 1; #2;
    b = 1; #3;
    a = 0; #5;
    b = 0; #7;
  end

  always begin
    @(a);
    if (a) begin
      $display("a"); #10;
    end
    @(b);
    if (b)
      $display("b");
  end

endmodule
```

(a)



(b)

Fig. 14. (a) A small Verilog program. The two *always* blocks run concurrently and are each surrounded by an implicit infinite loop. When a delay statement such as #2 executes, it suspends its thread for two units of simulated time. Similarly, when an event statement such as @(a) executes, its thread is suspended until a change occurs on signal a. (b) The event graph VeriSUIF generates for this program. A delay arc schedules its successor in the future. A conditional arc schedules its successor immediately if its expression is true. An event arc schedules its successor immediately only if the successor has been sensitized.

primitives, or concurrent processes described using procedural code (assignments, conditionals, loops, etc.). A process may also contain delay statements (e.g., #3) that suspend the process and schedule it to resume in a later time step. Event control statements (e.g., @(posedge clk)) suspend the process until the designated event occurs. Figure 14(a) is a small Verilog program illustrating some of these constructs. It is a single module consisting of two *always* blocks: concurrent processes wrapped in implicit infinite loops. The first *always* creates a simple waveform by setting a true initially, waiting two time units, setting b

```
set simulation time $t = 0$
schedule the first event in each process at time 0
**while** the event queue is not empty **do**
  **while** there are events at time $t$ **do**
    select and remove event $e$ from the set of events at time $t$
    mark $e$ as not sensitive
    execute the code for event $e$ (e.g., update variable states)
    **for all** outgoing delay arcs $e \xrightarrow{d} e'$ **do**
      schedule $e'$ at time $t + d$ (note: $d \geq 0$)
    **for all** outgoing conditional arcs $e \xrightarrow{b} e'$ **do**
      **if** expression $b$ is true **then**
        schedule $e'$ now
    **for all** outgoing event arcs $e \xrightarrow{v} e'$ **do**
      **if** event $e'$ is sensitive and event $v$ has occurred **then**
        schedule $e'$ now
    **for all** outgoing sensitizing arcs $e \rightarrow e'$ **do**
      mark $e'$ as sensitized
  advance simulation time $t$ to the earliest scheduled event
```

Fig. 15. The VeriSUIF simulation algorithm. The event queue is a set of event-time pairs. Additionally, each event can be marked as sensitive. VeriSUIF partially evaluates this algorithm along with the Verilog program to produce the final executable code.

true, waiting three time units, and so forth. The implicit loop means seven time units after b is set false the block restarts and a is set true again.

The second *always* block in Figure 14(a) begins by waiting for a change on a, then prints "a" and waits 10 time units if the value of a has become true. After this, the process waits for a change on b and prints "b" if the change made b true.

Compiled discrete-event simulation is the traditional efficient way to compile a discrete-event language such as Verilog. The compiler splits the Verilog program into pieces (events), generates a short C function for each, then links them with a generic (but carefully written) scheduler that maintains an event queue. During simulation, the scheduler selects the earliest event in the queue and invokes its function, which may add other events to the queue.

The VeriSUIF compiler of French et al. [1995] improves on this traditional technique by partially evaluating the operation of the event queue, which is often the performance bottleneck. It first translates a Verilog program into an event graph such as Figure 14(b), then partially evaluates the interpreter algorithm in Figure 15 to build a linear automaton. The interpreter consists of two loops: the outer runs all the events at the current simulation time and advances time once they are exhausted. The inner loop chooses a scheduled event, runs it, and schedules its successors in the event graph depending on the type of arc.

Figure 14(b) depicts the event graph VeriSUIF builds for the Verilog program in Figure 14(a). Each node is an event, a segment of procedural code from the original program that will run completely without scheduler intervention. For example, the conditional that checks b and prints "b" becomes event 10.

Four types of directed arcs connect the events. Delay arcs correspond to delay statements within procedural code. For example, the arc between events 0 and 1 is due to the #2 statement in the first *always* block. When the simulator executes an event with an outgoing delay arc, it schedules the event targeted by the arc for a later time instant. For example, running event 0 sets a false and schedules event 1 two time units in the future.

Conditional arcs allow the different branches of a conditional statement to contain delays. When both branches can execute atomically, such as with the test for b, the whole conditional is placed in a single event (e.g., event 10), but when one contains a delay, such as with the test for a, conditional arcs allow the scheduler to delay the execution of one of the branches. After executing the code for an event, the simulator checks the condition on each conditional arc and schedules the arc's target if the condition evaluates true.

Sensitizing arcs and event arcs work together to implement event-control statements. For control to be passed to a statement following an event-control statement such as `@(a)` two things must happen: control must pass to the event-control statement and the event must occur. A sensitizing arc controls the former: executing an event with outgoing sensitizing arcs marks the target of each arc as sensitized. Once an event is sensitized, an event arc can schedule it. When an event with outgoing event arcs is executed, the target event of the the arc is scheduled if it is sensitized and the event has occurred. For example, for event 10 to execute, event 9 must execute to sensitize event 10. Once it is sensitized, either event 1 or event 3 can cause event 10 to run.

To partially evaluate the behavior of the interpreter, VeriSUIF also manipulates unknown events, sensitive flags, and memory state. These are things whose presence or value cannot be completely established at compile-time and must be tested when the program runs. When the interpreter executes an unknown event it schedules unknown events in response. Similarly, unknown variables or sensitization conditions can also cause unknown events to be scheduled.

Figure 16 shows the code VeriSUIF generates for the Verilog program in Figure 14(a). The event queue is simulated with "t" variables that indicate whether certain events have been scheduled. Events that were present during simulation need no such variables: only "unknown" events need them. Furthermore, since each event may appear more than once in the queue, certain events may have more than one variable. Information about whether events are sensitive are stored in "s" variables, which are also only needed for "unknown" states.

The code generated for each event performs the operations that the algorithm in Figure 15 would perform. For example, the code for event 10 marks itself as not sensitive (`s10=0`), removes itself from the event queue (`t10a=0`), executes its code (calls `E10()`), and sensitizes event 5 (`s5=1`).

The partial evaluation algorithm produces exactly one next state for each state it reaches. Even if an event is known to be present in one case and absent in another, that event is marked "unknown" to ensure the number of states does not grow more than linearly.

To ensure the partial evaluation does not run forever, the simulator keeps track of all the states it has already seen and simply adds a branch back to the code for an earlier state when one is encountered. However, this can cause

```
int simulation_time;
int a, b;
int t6a, t6b, t6c;           /* Triggers for event 6        */
int t9a, t9b, t9c, t9d, t9e; /* Triggers for event 9        */
int t10a, t10b;              /* Triggers for event 10       */
int s5;                      /* Sensitize flag for event 5  */
int s10;                     /* Sensitize flag for event 10 */

void E0()  { a = 1; }
void E1()  { b = 1; }
void E2()  { a = 0; }
void E3()  { b = 0; }
void E6()  { display("a"); }
void E10() { if (b) display("b"); }

void run() {
  simulation_time = 0;
  s5 = s10 = 0;
  t6a = t9a = t10a = t6b = t9b = t9c = t10b = t9d = t6c = t9e = 0;
  s5 = 1;
  E0();                                              /* Event 0  */
  s5 = 0; if (a) t6a = 1; if (!a) t9a = 1;           /* Event 5  */
  if (t6a) { t6a = 0; E6(); t9c = 1; }               /* Event 6  */
  if (t9a) { t9a = 0; s10 = 1; }                     /* Event 9  */
  simulation_time += 2;
  E1(); if (s10) t10a = 1;                           /* Event 1  */

  for (;;) {
    if (t10a) { s10 = 0; t10a = 0; E10(); s5 = 1; }  /* Event 10 */
    simulation_time += 3;
    E2();                                            /* Event 2  */
    s5 = 0; if (a) t6b = 1; if (!a) t9b = 1;         /* Event 5  */
    if (t6b) { t6b = 0; E6(); t9d = 1; }             /* Event 6  */
    if (t9b) { t9b = 0; s10 = 1; }                   /* Event 9  */
    simulation_time += 5;
    E3(); if (s10) t10b = 1;                         /* Event 3  */
    if (t9c) { t9c = 0; s10 = 1; }                   /* Event 9  */
    if (t10b) { s10 = 0; t10b = 0; E10(); s5 = 1; }  /* Event 10 */
    simulation_time += 5;
    if (t9d) { t9d = 0; s10 = 1; }                   /* Event 9  */
    simulation_time += 2;
    E0();                                            /* Event 0  */
    s5 = 0; if (a) t6c = 1; if (!a) t9e = 1;         /* Event 5  */
    if (t6c) { t6c = 0; E6(); t9c = 1; }             /* Event 6  */
    if (t9e) { t9e = 0; s10 = 1; }                   /* Event 9  */
    simulation_time += 2;
    E1(); if (s10) t10a = 1;                         /* Event 1  */
  }
}
```

Fig. 16. Code generated by VeriSUIF for the example in Figure 14(a) (simplified: actual C code manipulates Verilog's four-valued variables).

Table III. Ways to Build Automata from Esterel

Compiler	Interpreted Representation	Generated Code Style
V2	Program text	Routine per state
V3	IC	Routine per state
Polis	IC	Fused states (BDD)
V5 automata	Combinational logic network	Routine per state

problems in situations where a system has zero-delay feedback that can stabilize. In such a situation, the simulation would treat all the events in the feedback loop as unknown and create an infinite loop. However, the system may stabilize to the point where none of the events in the loop is present. To avoid this problem, the compiler adds code to the end of the loop that tests whether any events remain. If none does, the generated code breaks from the loop and jumps to a new state created by assuming all the offending events are known not to be scheduled. This procedure can produce nested loops: inner loops correspond to zero-delay feedback; outer loops correspond to feedback with delay.

The Verilog language is complicated, but very flexible. As such, partially evaluating the behavior of the queue is difficult and only part of its behavior can be predicted at compile-time. Nevertheless, this can provide a significant speedup.

7. AUTOMATA IN ESTEREL

The VeriSUIF compiler described in the last section transforms the behavior of the Verilog event queue into a simple automaton whose states are arranged in nested loops. This works because most digital logic is clocked to produce periodic behavior, but is also a side effect of only considering one next state for each current state. This suggests other approaches could track more state variables more precisely.

The synchronous nature of Esterel suggests dividing states at cycle boundaries. The compilers in this section use this approach to generate more complicated, but also more efficient automata from Esterel code (Table III). Each state in such automata represents the set of control points where the program can reawaken at the beginning of a cycle. Within each state, the compiler tracks the values of internal signals completely, allowing it to compile away all operations on internal signals. As a result, the code for each state only manipulates external signals and persistent data (e.g., variables). Figure 17 shows the automata-generated code for the program in Figure 4(a).

Four techniques for compiling Esterel using automata have been developed (Table III). The early V2 compiler [Berry and Cosserat 1984] used an interpreter written in LISP to directly manipulate the program text according to its operational semantics [Berry and Gonthier 1992], which were written in Plotkin's [1981] structural rewriting style. These semantics give rules for rewriting a program into another program that behaves like the original program in the next cycle. This rather literal implementation was based on Brzozowski's [1964] idea of taking derivatives of regular expressions.

Although conceptually elegant, the V2 approach makes for a slow, memory-hungry compiler, so for his thesis, Gonthier [1988] developed the equivalent

```
int state_0() {
  return 0;
}

int state_1() {
  return 2;
}

int state_2() {
  if (S) return 3;
  return 2;
}

int state_3() {
  if (S) return 3;
  if (I) return 4;
  return 5;
}

int state_4() {
  if (S) return 3;
  emit_O();
  return 6;
}
```

```
int state_5() {
  if (S) return 3;
  if (I) { emit_O(); return 6; }
  return 3;
}

int state_6() {
  if (S) return 3;
  return 7;
}

int state_7() {
  if (S) return 3;
  return 6;
}

int state = 1;

int tick() {
  typedef int (*StateFunction)();
  static StateFunction stateFunctions[] = {
    state_0, state_1, state_2, state_3,
    state_4, state_5, state_6, state_7
  };
  state = stateFunctions[state]();
  S = I = 0;
  return state != 0;
}
```

Fig. 17. Automata-style code generated by the V3 compiler for the program in Figure 14(a). The tick() function performs the program's actions for a cycle by calling one of the state functions. Each of these check inputs, emit outputs, and return an integer representing the next state.

but much faster V3 technique that simulates an Esterel program in the IC representation (Figure 4(b)). See Section 2 for a discussion of the IC format's semantics.

The V3 tracks which signals may still be emitted in a cycle to order concurrently running instructions. Esterel's rule is that no statement that checks a signal may run until all possible emitters have run. While simulating the program in each cycle, V3 waits at any statement that checks a signal with an *emit* statement reachable from one of the current program counters.

Generating separate code for each state can be wasteful, inasmuch as many states contain similar code. The Polis group's compiler [Chiodo et al. 1995; Balarin et al. 1999] attempts to share code between states by representing the automaton and its branching programs as a single reduced, ordered binary decision diagram (BDD) [Bryant 1986]. Figure 18 shows the control-flow graph their compiler generates for the example in Figure 4(a).

The Polis approach can generate more compact code than other automata-based compilers. For example, there is only one test for S in Figure 18, compared with one per state in Figure 17. However, the generated code may still be exponentially larger than the source because each state is still represented explicitly.

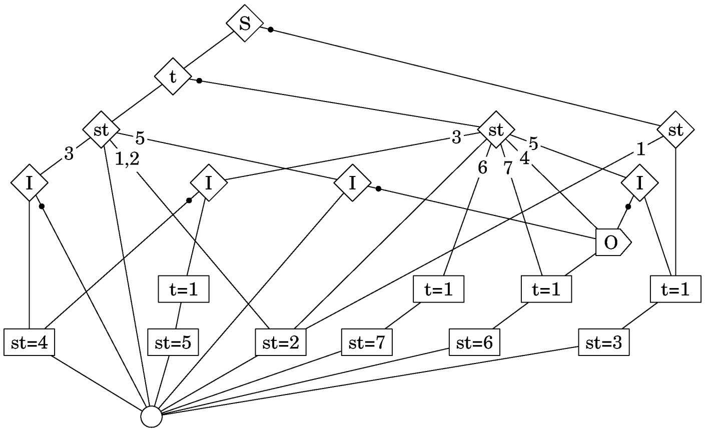


Fig. 18. The control-flow graph generated by the Polis system for the program in Figure 4(a). S, I, and O are interface signals, st represents the current state, and t represents a selftrigger flag that indicates the program will run in the next cycle even if no input signal is true. Arcs with dots represent true branches. Because this graph is directly synthesized from a BDD, the variables are tested and assigned in the same order along any path from top to bottom.

Reducing the size of the code generated by the Polis compiler requires minimizing the number of nodes in a BDD, an NP-complete problem. Fortunately, effective heuristics exist (the Polis group uses Rudell's [1993] popular sifting algorithm). One drawback of the BDD representation they use is the requirement that variables be tested in the same order along all paths. For simply nested conditionals, this is not a drawback, because outer preemption conditions must always be tested before inner ones, but some programs always have poor orderings because different states test variables in different orders.

Berry [1999] now bases the semantics of Esterel on constructive logic, which uses a more complex rule to decide which statements are reachable. To match these semantics, the latest automata-based compiler (V5 in automata mode) from Berry's group simulates a combinational logic network (see Section 2) to derive the behavior of each state.

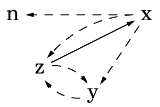
Automata generated from Esterel programs usually have little redundancy because it is relatively difficult in Esterel to add control state and later ignore it. But it can happen: states 6 and 7 in Figure 17 are equivalent and correspond to the case where the second thread continues to switch between states but the program reacts the same way in each. Nevertheless, state minimization is not part of the normal Esterel automata compiler flow.

8. AUTOMATA IN LUSTRE

Esterel programs have a complicated mixture of control and data dependencies that makes them fairly difficult to analyze. By contrast, the Lustre language presented in this section only contains acyclic data dependencies, making it much easier to simulate. A Lustre program is a system of equations that dictate how its state, held in variables, evolves between cycles. The compiler presented in this section exhaustively simulates a Lustre program to build an automaton

```
node ex(i: bool) returns (n:int);
  var x, y, z : bool;
let
  n = if true->pre(x) then 0 else 0->pre(n) + 1;
  x = if true->pre(x) then false else z;
  y = if true->pre(x) then true->pre(y) and i
                      else true->pre(z) or  i;
  z = if true->pre(x) then true->pre(z)
                      else (true->pre(y) and true->pre(z)) or i;
tel;
```

(a)



(b)

Fig. 19. (a) A contrived Lustre program to illustrate compilation techniques, from Halbwachs et al. [1991]; (b) its variable dependency graph. The solid line denotes a direct dependence; the dashed line denotes a delayed dependence. Self-loops are omitted.

whose states represent assignments of concrete values to Boolean variables. Pruning redundant states on the fly is the main challenge, because these states are created when a program ignores something computed in a previous state.

Programs in the dataflow language Lustre [Caspi et al. 1987; Halbwachs et al. 1991] are a set of definitions (e.g., Figure 19(a)), each assigning the value of an expression to a variable. It is a synchronous language, so all definitions are evaluated in each clock cycle in an order given by data dependencies. Variables may take Boolean, integer, or floating-point values, and expressions contain the usual combination of constants; variable references; arithmetic, logical, and comparison operators; and conditionals. By itself, a variable reference refers to the value of that variable in the current cycle, but `pre(v)` refers to the value of the variable `v` in the previous cycle. The `pre()` operator may be nested, but the depth of this nesting is constant and may not be controlled by a variable, meaning Lustre programs only use finite memory that can be determined at compile time. The `->` operator controls how delayed variables are initialized; for example, `0->pre(x)` is 0 in the first cycle, and the previous value of `x` in later cycles.

Halbwachs et al. [1991] describe how to compile Lustre programs. As with generating code for a logic network, the easiest way is to topologically sort the definitions according to data dependencies and evaluate each expression in scheduled order. Lustre prohibits cyclic dependencies, so a topological sort always exists.

Figure 19(b) shows the dependency graph for the program in Figure 19(a). First, note that there is only one direct dependency (the solid line), because the present value of `x` depends on the present value of `z`. Without the dashed lines, the graph is acyclic, so the program is valid and thus can be scheduled.

Although expressions may read the present and earlier values of a particular variable, it may be possible to use the same storage to represent both, provided the earlier value is not needed after the present value is computed. Reversing the direction of the dashed lines in Figure 19(b) corresponds to forcing every use of a variable's past value to come before its new value is computed. Doing this to every variable in Figure 19(a) results in a cycle because of the mutual dependency between the past and present values of the variables `z` and `y`, but introducing a variable for the last value of `z` effectively removes the arc from

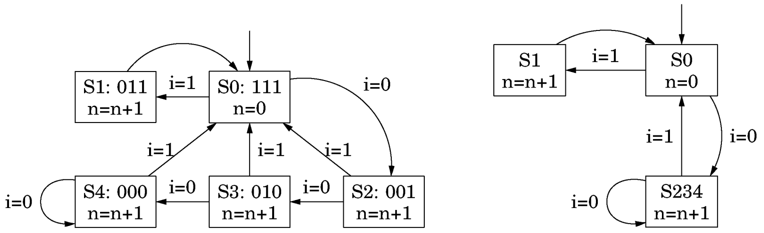
```
int n = 0;
bool x = 1;
bool y = 1;
bool z = 1;
bool pre_z;

int ex(int i)
{
  pre_z = z;
  n = x ? 0 : n + 1;
  z = x ? z : (y && z) || i;
  y = x ? y && i : pre_z || i;
  x = x ? false : z;
  return n;
}
```

(a)

```
int ex(int i)
{
  pre_z = z;
  if (x) {
    n = 0;
    y = y && i;
    x = false;
  } else {
    n = n + 1;
    z = (y && z) || i;
    y = pre_z || i;
    x = z;
  }
  return n;
}
```

(b)

Fig. 20. (a) A straightforward C implementation of the Lustre program in Figure 19(a) based on the schedule n, z, y, x; (b) an implementation factored with respect to x.



(a) (b)

Fig. 21. (a) An automaton for the Lustre program in Figure 19(a). Each state is labeled with the previous values of x, y, and z along with the rule for updating n; (b) a minimal automaton for the same program. States S2, S3, and S4 were merged.

z to y, resulting in an acyclic graph that can be scheduled. Figure 20(a) shows the code this produces.

This style of code generation is fine for pure dataflow programs such as simple filters that compute essentially the same arithmetic function in each cycle, but there is a more efficient alternative when programs contain Boolean variables and conditionals.

The tests of x in Figure 20(a) are redundant. Factoring them as in Figure 20(b) will improve both the size and speed of the program, but further optimization is possible. The test of x in Figure 20(b) is still somehow redundant because the next value of x is a known constant when x is true.

The key optimization of Lustre programs comes from factoring the code with respect to some subset of its Boolean variables, effectively treating the program as a state machine. This produces more efficient code because it removes code that tests and sets certain variables, but this can also substantially increase the amount of generated code.

Simulating the program in Figure 19(a) produces the automaton in Figure 21(a). Each state is labeled with the values of x, y, and z in the previous

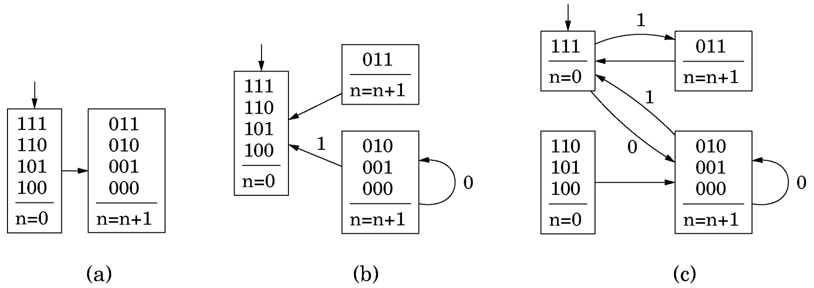


Fig. 22. Partitioning the state space of the Lustre program in Figure 19(a). Each state is labeled with the concrete states it represents (values of x, y, and z) and the action it performs. (a) Divide according to the two different types of output. (b) Split the successor of the initial state into two, because its successor depends on whether y and z are true. (c) Split the successors of the just-split states. The state in the lower-left is unreachable and will be pruned.

state along with the rule for updating n in that state. For example, from the initial state S0 the i input determines the next value of y, but x always becomes 0 and z becomes 1, leading to states S1 and S2.

The automata generated by such simulations are correct but often redundant. For example, in Figure 21(a), states S2 to S4 are equivalent and can be merged to form Figure 21(b) without affecting behavior. Such nonminimality does not affect the speed of the generated code, only its size. These redundant states are generated when the program ultimately ignores the values of internal variables: such states appear different but behave identically.

The main contribution of Halbwachs et al. [1991] is an algorithm for minimizing the automaton as it is generated. This improves compilation speed, capacity, and code size since the nonminimal automaton can be much larger.

Like classical state minimization algorithms, the incremental one proposed by Halbwachs et al. begins with a single state and partitions states when it discovers differences. Each state is characterized by a set of concrete state values, rather than a single state value.

To illustrate this, again consider the program in Figure 19(a). The first step is to characterize all the possible outputs of the program. The value of n is computed in two ways, depending on the value of x, so there are initially two states (Figure 22(a)).

Now consider the successors of the state on the right of Figure 22(a). Its successors depend both on the input i and whether y and z are both true. Thus this state is split to produce Figure 22(b) and its successor considered.

Now the initial state is split, since its successors depend on i only when x, y, and z are all true (Figure 22(c)). The process is finished because the successors of these states do not need to be split.

In the Lustre compiler, the set of concrete states associated with each state is represented with a BDD to simplify splitting it with respect to a predicate.

Figure 23(a) shows the code generated from the automaton in Figure 22(c). Note that the state in the lower left of Figure 22(c) is unreachable and was pruned away with a simple depth-first search from the initial state.

```
int state = 0;
int n = 0;
int ex(int i)
{
  switch (state) {
    case 0: n = 0;     state = i ? 1 : 2; break;
    case 1: n = n + 1; state = 0;         break;
    case 2: n = n + 1; state = i ? 2 : 0; break;
  }
}
```

Fig. 23. The final code generated from the minimized automaton in Figure 22(c).

In the partial evaluation framework, the Lustre interpreter is straightforward: it evaluates the definitions in topological order. The partial evaluator, however, is the sophisticated state partitioning algorithm that divides the state space of the system based on nonequivalent states.

Building automata from Lustre is effective because the compiler can make a good guess about how to abstract the system (it abstracts all but Boolean-valued signals, which are presumed to select different modes). It is still a fairly simple dataflow language, however, because the whole program runs at the same rate. Compiling Synchronous Dataflow, described in the next section, is harder because parts of the system may operate at different rates.

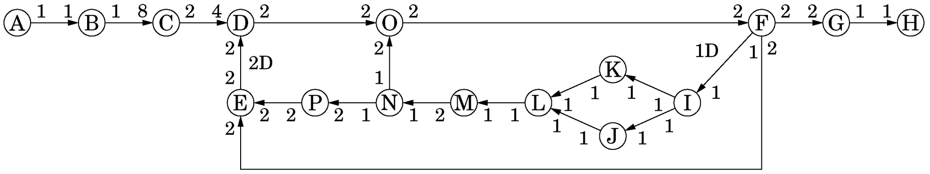
9. SYNCHRONOUS DATAFLOW NETWORKS

The Lustre language presented in the last section is a single-rate dataflow language in which all parts of the system execute at the same rate. This is conceptually simple and leads to minimal, predictable buffering between concurrently running sections of the program. By contrast, the synchronous dataflow networks presented in this section (synchronous dataflow or SDF is an unfortunate name because it is not synchronous in the usual sense; a better name would have been periodic dataflow) permit different concurrently running parts of a system to execute at different fixed rates.

Synchronous Dataflow [Lee and Messerschmitt 1987b] is a multirate block-diagram dataflow language. Each block has a certain number of input and output ports, each labeled with the number of data tokens the block consumes or produces on the port each time the block fires. Ports are connected by unbounded first-in first-out queues. Figure 24(a) shows a typical multirate SDF system with feedback.

The main challenge in compiling SDF is determining a periodic block firing schedule that allows the system to run indefinitely without buffer underflow (i.e., attempting to fire a block with insufficient tokens in its input buffers) or an unbounded accumulation of tokens. In the partial evaluation framework, SDF is straightforward: the interpreter evaluates blocks in scheduled order, and the partial evaluator simply lays down code for each block in that order. Scheduling is the only challenge.

Finding a correct schedule is fairly easy using the procedure developed by Lee and Messerschmitt [1987a]. The first step is determining the number of times each block will fire per schedule period by solving the constraints imposed



(a)

(16 AB)(2 C) I J K L M (2 N) D O F P E G H

(b)

```
for (i=1 ; i < 16 ; ++i) { A(); B(); }
for (i=1 ; i < 2  ; ++i) C();
I(); J(); K(); L(); M();
for (i=1 ; i < 2  ; ++i) N();
D(); O(); F(); P(); E(); G(); H();
```

(c)

Fig. 24. (a) A modem in SDF. The arcs represent FIFO queues, and each is labeled with the number of tokens produced and consumed by each block when it fires. Arcs with labels such as "2D" start with two tokens, which behaves like a delay; (b) a single-appearance looped schedule; (c) code generated from the looped schedule. (After Bhattacharyya et al. [1999]).

by the relative production/consumption rates along each arc. For example, if c and d are the number of times blocks C and D fire per cycle in Figure 24(a), then $2c - 4d = 0$ must be satisfied. Each arc leads to a similar equation, and the resulting system of equations is called the balance equations. Lee and Messerschmitt [1987a] show the balance equations only have the all-zero solution when the graph has inconsistent rates, and a unique minimum positive integer solution otherwise. For Figure 24(a), this solution is $a = b = 16, c = n = 2$, and all others 1.

That the balance equations have a nonzero solution is necessary but not sufficient for a schedule to exist, because the system might deadlock. Fortunately, Lee and Messerschmitt [1987a] show that any algorithm which correctly simulates buffer behavior (i.e., firing a block only when enough tokens are available) will always find a schedule if one exists; that is, choosing to fire a block never leads to a deadlock that could have been avoided.

Code generated from SDF generally consists of copies of the code for each block wrapped in simple counted loops, such as in Figure 24(c). This is represented as a looped schedule: a sequence of blocks and loops, which are parenthesized terms consisting of a count followed by a sequence of blocks or loops. Figure 24(b) is a looped schedule for Figure 24(a).

Most SDF schedulers seek a single-appearance schedule (SAS)—one in which each block appears exactly once—because it results in minimum code size when block code is inlined. A divide-and-conquer approach is generally used to find single-appearance schedules. A topological sort of an acyclic portion of an SDF graph trivially leads to an SAS for that portion of the graph, so an SDF graph is recursively divided into strongly connected components (SCCs). Bhattacharyya et al. [1993] have shown that any arc in an SCC that starts with at least as many tokens as will pass through it during the complete schedule does not constrain the block firing order and can be ignored during scheduling. Thus these arcs are removed and the SCC decomposition can proceed.

Minimizing the buffer storage needed to execute a schedule is another common objective. Acyclic SDF graphs often have more than one SAS (due to different topological orders), each with different buffer requirements. A number of heuristics have been proposed to choose suitable orders.

Although the SDF formalism is simple to describe and understand, its scheduling flexibility is so great that much work has been done on scheduling it efficiently. Bhattacharyya et al. [2000] provide a summary of work in this area. See also the book by Bhattacharyya et al. [1996].

Many SDF variants have been proposed. Cyclostatic dataflow [Bilsen et al. 1995] is one of the more successful. CSDF blocks may only produce or consume a single token per firing, but may have periodic firing rules. The increased granularity generally allows for smaller buffers. Buck's [1993, 1994] Boolean dataflow adds limited data-dependent behavior, which makes the formalism Turing-complete, but many systems can be still be scheduled statically using SDF-like techniques.

SDF has so many variants because it is a significant restriction of two much more powerful formalisms. It is a subset of Kahn [1974] process networks, a more general model of data-dependent dataflow whose behavior remains deterministic. Scheduling these networks is difficult; see Kahn and MacQueen [1977], Lee and Parks [1995], and Parks [1995] for online algorithms for doing this.

SDF is also equivalent to a restricted class of Petri nets, a very powerful, flexible model of concurrency at the center of the two techniques in the next two sections. Many of the SDF scheduling results are actually simplified versions of more general results for Petri nets; for example, the balance equations and related theorems are related to the concepts of S- and T-invariants. See Murata [1989] for a survey of this vast domain.

10. PETRI NETS IN PICASSO C

The Synchronous Dataflow model described in the previous section is in some sense very restrictive because it does not allow data-dependent actions. Although this does make for very efficient scheduling, the model can be too restrictive for many applications. This section and the next discuss models of concurrency based on the much more flexible Petri net model, which allows for both data-dependent actions and buffered communications among concurrently running sequential processes specified with imperative languages.

Lin's [1998] language[1] describes systems of concurrently running processes written in C that communicate using rendezvous-based communication inspired by Hoare's [1985] communicating sequential processes: a transmitter will block until the receiver is ready to receive and a receiver will block until

[1] Lin does not name this language, but mentions it is part of a system called Picasso; I refer to it as Picasso C.

```
myprocess (int n, output chan(int) nn) {
  for (;;) {
    nn <-= n;  /* send on nn */
    if (n == 1) break;
    if (n&1)
      n = 3*n+1;
    else
      n = n/2;
  }
}
```

(a)

```
rep (input chan(int) nn) {
  int k;
    for (;;) {
      k = <-nn; /* read from nn */
      if (k>0) {
        while (--k>0) p();
        q();
      }
    }
}
```

(b)

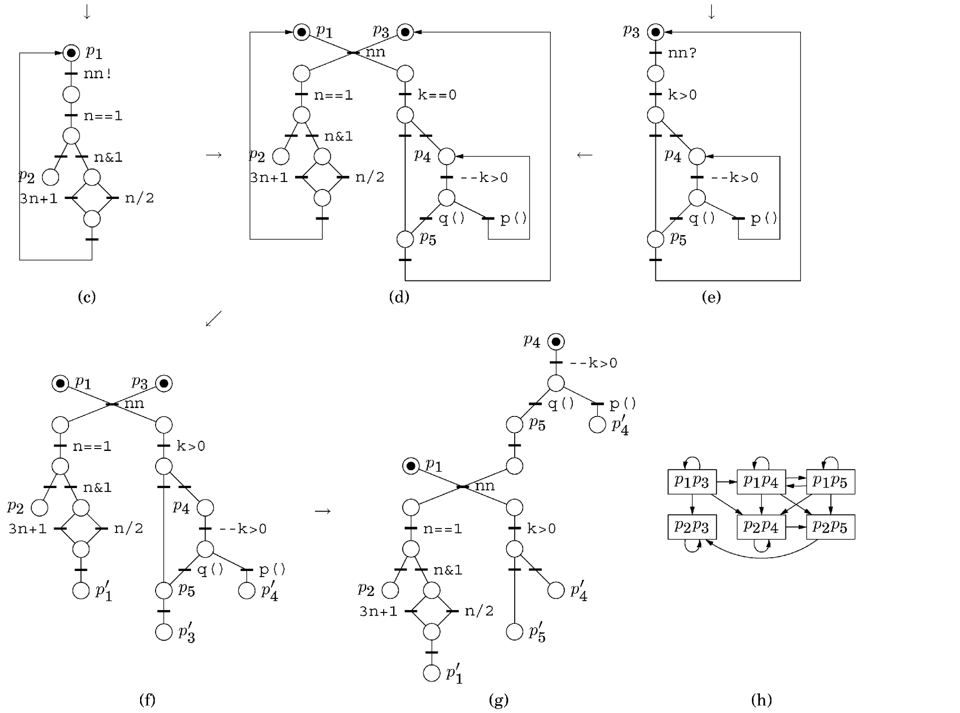


Fig. 25. Building a finite-state model of a system in Lin's language. Processes (a) and (b) are transformed into Petri nets (c) and (e) and fused to form (d). The maximal expansion segment (f) is generated by unrolling (d) starting at the initial marking p1p3. The ME (g) is generated by unrolling (d) from p1p4. (h) is the state machine generated by all the expansion segments.

the transmitter sends data. These semantics are elegantly captured by a Petri net, which Lin uses as an intermediate representation.

Lin's synthesis procedure is based on the idea of unrolling first proposed by McMillan [1992, 1995], and is similar to the automata generation procedure in the Esterel V3 compiler (Section 7). However, instead of breaking states at clock cycle boundaries (which are not well defined in Petri nets), Lin breaks states at the boundaries of looping behavior.

Figure 25 illustrates the first part of Lin's synthesis procedure. Starting from processes written in a C dialect that adds send and receive operations (Figures 25(a) and (b)), Lin transforms them into Petri nets that represent their control flow (Figures 25(c) and (e)). Transitions generally represent statements (some are null) and places represent where control can be between statements.

The compiler fuses all the processes into a single Petri net, joining them at transitions that represent communication on the same channel. In Figure 25(d), only a single transition is shared, but in general more transitions may need to be added so that each send on channel c, say, could conceivably rendezvous with any receive on channel c. This is potentially a quadratic increase in the number of transitions.

After the Petri nets for each process are fused, the compiler enumerates the states of the system and generates code for each. Each state is a maximal expansion (ME): a maximal unrolling of the fused Petri net starting at a particular marking and going until places are encountered again. For example, Figure 25(f) is a ME from the initial marking p1p3. Simulating this ME completely produces four "cut-off markings": configurations where no transitions are enabled. These generally mark the final places in an ME, but may also include situations where one process is blocked waiting for the other to communicate.

Forming a maximal expansion, finding all its cut-off markings, and expanding from those markings produces a state machine such as that in Figure 25(h). All that remains is to generate code for each ME as shown in Figure 26.

Code for each ME is generated by simulating it according to a schedule and recording the simulation results as a control-flow graph easily transformed into code. A schedule is a topological ordering of the transitions in an ME that may group transitions. Because an ME is acyclic by construction, a schedule is simply an order that ensures that no transition can run before any of its predecessors. In Figure 26, the position of each transition indicates where it is in the schedule: transitions aligned horizontally are in the same group.

A control-flow graph is generated from a scheduled ME by simulating it. Each state represents a group of transitions in the ME that are all enabled and can fire simultaneously, and a transition of the state machine corresponds to a marking of the ME. Different schedules can trade code size and speed. For example, Figure 26(a) is an as-soon-as-possible schedule that leads to large fast code because it fuses decisions, forcing code to be duplicated. By contrast, Figure 26(b) schedules one thread after the other, rather than running them simultaneously, and produces smaller code that is slightly slower (requiring four decisions instead of three).

Because Lin's technique to generate code encodes all the state information in the program counter, there is a danger of exponentially large code. Although mild in this example, already there is no schedule that makes it possible to avoid duplicating the calls to p and q. The problem becomes more pronounced as more processes with conditionals run concurrently. The EC compiler, described in Secton 3, uses a different algorithm for sequentializing such graphs that avoids the exponential blow-up at the expense of additional run-time overhead.

Later work by Zhu and Lin [1999] addresses the exponential explosion by synthesizing multiple state machines for the set of processes. They then run each state machine in round-robin fashion. Each machine checks whether it is blocked waiting for a rendezvous before advancing its state.

In the partial evaluation framework, Lin's compiler uses the Petri net interpreter in Figure 1 and a partial interpreter that builds automata. Because

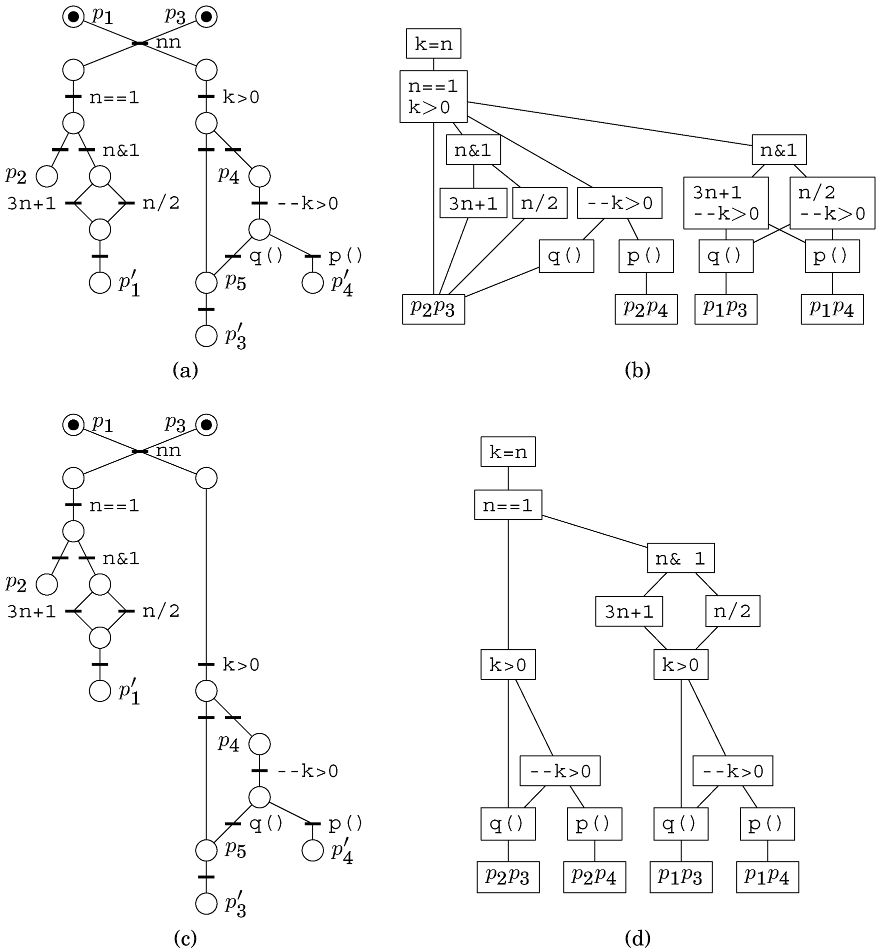


Fig. 26. Synthesizing sequential code from maximal expansions: (a) a scheduled ME; (b) the code generated from it; (c) a different schedule, one that delays the execution of one of the processes; (d) the resulting code.

his language is not synchronous as is Lustre or Esterel, Lin uses a different criterion—the boundaries of maximal expansion segments—for determining state boundaries. Another novel aspect of Lin's automata approach is his mechanism for generating code for each state. Rather than directly using the interpreter, there is an additional scheduling step between when the behavior of a state is identified and when it is synthesized.

11. PETRI NETS IN FLOWC

Lin's formalism described in the previous section uses rendezvous communication, which forces communicating processes to synchronize and by definition does not involve SDF-like buffered communication. The FlowC compiler in this section uses buffered communication and data-dependent actions. This is one of the richer models considered in this article, and it makes the scheduling

```
PROCESS GetData(InPort IN,
                OutPort DATA)
{
  float sample, sum; int i;
  for (;;) {
    sum = 0;
    for (i=0 ; i<N ; i++ ) {
      READ(IN, sample, 1);
      sum += sample;
      WRITE(DATA, sample, 1);
    }
    WRITE(DATA, sum/N, 1);
  }
}
```

```
PROCESS Filter(InPort DATA, InPort COEF,
               OutPort OUT)
{
  float c,d; int j;
  c = 1.0; j=0;
  for (;;) {
    SELECT( DATA, COEF ) {
    case DATA:
      READ(DATA, d, 1);
      if (j==N) {
        j=0; d *= c; WRITE(OUT, d, 1);
      } else j++;
      break;
    case COEF:
      READ( COEF, c, 1);
      break;
    }
  }
}
```

Fig. 27. A pair of communicating processes in FlowC.

problem harder because it must consider and avoid over- and underflowing buffers. The main difference is that Lin considers the entire state space of the system, whereas the FlowC compiler must choose a subset carefully.

Cortadella et al.'s FlowC [1999, 2000] describes systems as a collection of concurrently running dataflow processes that react to inputs from the environment and communicate through buffers, which may be bounded or unbounded. Each process is specified in its own variant of C that adds the ability to read and write data tokens to ports. Figure 27 shows a pair of communicating processes written in FlowC.

Their approach to generating code starts by generating Petri net representations of each process and connecting them with places that represent the buffers between processes. For example, the place labeled p_{data} in Figure 28 represents the buffer for the DATA channel of Figure 27.

After building a monolithic Petri net for the system, they then search for a schedule: a finite, cyclic, and in some sense complete, portion of the usually infinite state space of the system. Here a state of the system is a marking of the Petri net: the position of the program counter in each process plus the number of data tokens in interprocess buffers. Because it is finite and cyclic, such a schedule can be executed indefinitely without any unbounded accumulation of data in buffers. Concretely, a schedule is a cyclic directed graph whose nodes represent markings (states) and whose arcs represent transitions.

Data-dependent decisions make the scheduling problem for FlowC harder than the equivalent one for Synchronous Dataflow. The state space of an SDF system is very regular, so regular that there is a theorem that says there are no immitigable scheduling choices. By contrast, a bad choice made early while scheduling FlowC can lead to no schedule even if one exists. Just finding a correct FlowC schedule may be as difficult as finding an optimized SDF schedule.

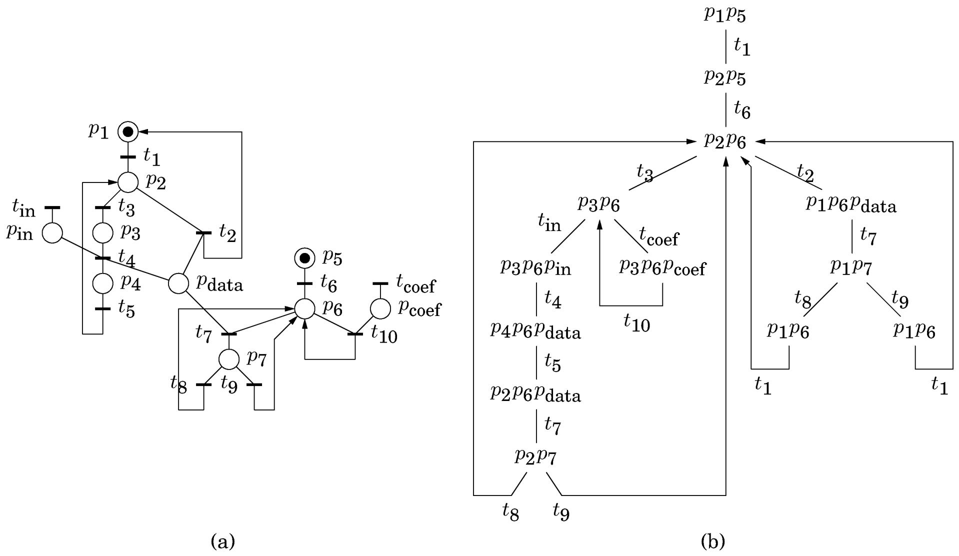
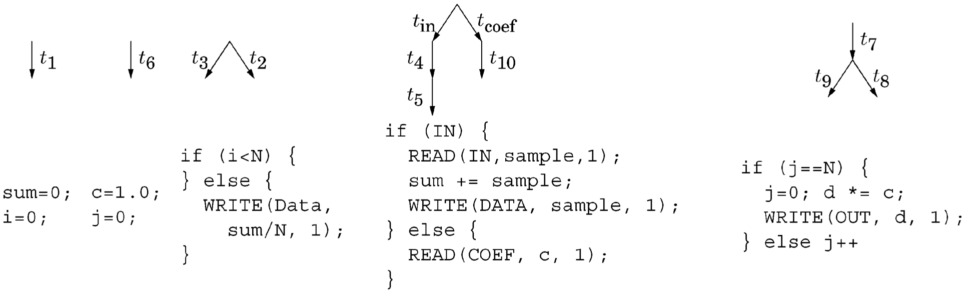


Fig. 28. (a) The Petri net representing the processes in Figure 27; (b) a schedule for that net.

Choosing which transitions to fire at a particular marking to find its successor states is the fundamental problem in scheduling. The first question is how many transitions need to be considered. Firing a single transition to produce a single successor state usually suffices, but when a process is at a decision point (e.g., an if-then-else statement), each possible outcome of that decision must be considered. For example, at the initial marking p_1p_5, both t_1 and t_6 are enabled, but only one needs to be fired because firing the other would only produce a different interleaving. By contrast, at the marking p_2p_6, the left process is making a decision so both t_2 and t_3 need to be fired, producing the two successor states p_3p_6 and $p_1p_6p_{\rm data}$. Formally, at each marking, all the transitions in one of the enabled equal conflict sets (ECSs) must be taken, but no others need be. Each ECS is a set of transitions that for any marking are either all enabled or all disabled. An ECS is either trivial (a single transition) or the set of transitions under a decision point in a process. The ECSs are a unique partition of the transitions in the system. In Figure 28(a), $\{t_1\}$, $\{t_2, t_3\}$, $\{t_5\}$, and $\{t_8, t_9\}$ are each equal conflict sets.

Their scheduling algorithm recursively explores the state space, considering each enabled ECS at each marking. If the algorithm cannot find a schedule after firing each transition in the first ECS, it considers the transitions in the second ECS, and so forth. The algorithm returns “no schedule” if the number of tokens in a buffer place exceeds its bound.

At a particular marking, the order in which ECSs are considered affects both the size (quality) of the schedule and how quickly it is found because the algorithm always returns the first schedule it finds. Cortadella et al. choose this order heuristically. ECSs that participate in a T-invariant are considered first. Briefly, a T-invariant is a collection of transitions that, when fired, leave the system in the same marking (equivalent to a solution of the balance equations



```
sum=0;
i=0;
```

```
c=1.0;
j=0;
```

```
if (i<N) {
} else {
  WRITE(Data,
    sum/N, 1);
}
```

```
if (IN) {
  READ(IN,sample,1);
  sum += sample;
  WRITE(DATA, sample, 1);
} else {
  READ(COEF, c, 1);
}
```

```
if (j==N) {
  j=0; d *= c;
  WRITE(OUT, d, 1);
} else j++
```

Fig. 29. The segments for the schedule in Figure 28(b) (above) and the fragment of code generated for each (below).

in SDF). These are desirable because the schedule must ultimately be cyclic (i.e., always be able to return to some marking). ECSs of environmental inputs (e.g., $\{t_{\text{in}}\}$, $\{t_{\text{coef}}\}$) are considered last because firing them generally requires buffering.

Code could be generated directly from the schedule by interpreting it as a control-flow graph, but many paths in a schedule often contain the same transitions and thus would produce identical code. For example, the parts of the schedule rooted at $p_2 p_6 p_{\text{data}}$ and $p_1 p_6 p_{\text{data}}$ fire transition t_7 and then either t_8 or t_9, and so correspond to the same sequence of code. To reduce code size, common subtrees are identified and code generated for each unique one.

Their goal is a set of code segments: small tree-structured sequences of code (Figure 29) built of transitions (actually ECSs) from the system's Petri net. Code is never duplicated: each transition (statement) appears in exactly one code segment. Furthermore, the segments (suitably duplicated) must cover the schedule, making them sufficient to run the whole system.

Making each ECS a code segment is correct but wasteful inasmuch as many ECSs always appear in the same sequence (e.g., t_7 is always followed by t_8 and t_9 in Figure 28(b)). Combining ECSs that always run together into a larger code segment produces faster code by reducing the need for intersegment run-time scheduling. Their algorithm (described in a technical report by Cortadella et al. [1999]) walks through the schedule, adding previously undiscovered transitions to existing segments and splitting segments when encountering transitions in the middle of an existing one.

Each code segment is treated as a control-flow graph. The code for each transition is a sequence of statements from the original FlowC specification (e.g., t_1 corresponds to `sum=0; i=0;` from Figure 27), and branching in a code segment corresponds to a conditional test: either an *if* or a *switch* statement. Figure 29 shows the segments their algorithm generates along with the fragments of code each implies.

Cortadella et al.'s technique, as do all in this article, works on an abstraction of a concurrent system to simplify analysis. Specifically, the scheduler does not model the information used at a data-dependent choice such as the ECS after p_2 in Figure 28 and must assume both choices are always possible. This makes the scheduling problem decidable at the expense of declaring unschedulable some systems that could run with finite buffers. For example, a process that generated tokens in a loop that was guaranteed to terminate after, say, 10 iterations might appear to require unbounded buffers because the scheduling routine would not know this bound.

The synthesis technique described above produces a monolithic function for the entire system, but the FlowC compiler has the option of splitting a system into multiple tasks, one per environmental input, invoked in response to an input. The Petri net for the system and its schedule are identical, but instead of considering the whole schedule, the code segment generation routine instead considers the slice of the schedule with respect to the environmental input. Such a slice starts at the transition corresponding to an environmental input and contains all paths from this transition that eventually return the same marking. Code for each task is generated from each segment using the same technique as before.

Related to the FlowC technique is the work of Sgroi et al. [1998, 1999], which performs similar scheduling of Petri net networks with the ultimate goal of code generation, albeit on a more restricted model. Instead of the explicit techniques used in the FlowC compiler, the technique of Strehl et al. [1999] works implicitly, representing behavior using BDDs. This approach requires the designer to provide buffer-size bounds explicitly. This makes the scheduling problem decidable, but determining these bounds can be difficult.

12. CONCLUSIONS

This article surveys a variety of compilation techniques for translating concurrent specifications written in different languages into code that can be compiled and run on a sequential processor without operating system support for concurrency. The techniques were classified based on their concurrent formalism, which ranges from dataflow specifications that impose no control relationships among parts of the system to Petri nets that focus on synchronization. The techniques differ dramatically, even among those for the same language.

Each compiler can be thought of as a partial evaluator operating on an interpreter, allowing each portion to be considered separately. Such a framework makes it easier to envision how a partial evaluator used in one compiler might be applied to a different interpreter.

The main point of surveying these techniques is to facilitate combining ideas from these techniques to produce new better compilers. Little work has been done on combining these ideas; I believe the time has come.

Due to space, I have not included other noteworthy techniques. For example, the standard subset construction technique used to convert nondeterministic finite automata into deterministic automata (see, e.g., Hopcroft and Ullman [1979]) is a means to convert a concurrent specification into a sequential one. Compiling Statecharts [Harel 1987; Harel and Naamad 1996] is another case. Ackad [1998] compiles Statecharts with an eye toward reducing the number of intermediate variables that resembles the same problem in Lustre. Compiling the multirate Signal language [Le Guernic et al. 1991] presents a similar problem. The concurrent Reactive C language [Boussinot and Doumenc 1992] is compiled using coroutines. The concurrent Squeak language of Cardelli and Pike [1985] is compiled using an automata approach.

Ultimately, I believe a mixture of these techniques will prove superior to any single one. The Lustre compiler's technique for minimizing the state space on-the-fly seems especially powerful and has many potential applications. I suspect this idea could be combined with the linear automata approach of the VeriSUIF compiler to produce a far more clever way to compile discrete-event simulations. The machinery developed for compiling Program Dependence Graphs can produce amazingly efficient code in restricted circumstances. Extending it to gracefully degrade into a technique such as that used in my EC compiler will probably improve code generated for many control-flow-based languages. Perhaps some of the techniques developed for scheduling Petri net systems can also be applied to systems based on event graphs.

In any case, my hope is that broader understanding of these techniques will lead to more efficient and powerful ways to create software for embedded systems.

REFERENCES

Ackad, C. 1998. Software synthesis from Statechart models for real time systems. In *Proceedings of the International Workshop on Distributed and Parallel Embedded Systems (DIPES)*. IFIP Conference Proceedings, vol. 155, Kluwer, Paderborn University, Germany, 73–84.

Balarin, F., Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., Lavagno, L., Sangiovanni-Vincentelli, A., Sentovich, E. M., and Suzuki, K. 1999. Synthesis of software programs for embedded control applications. *IEEE Trans. Comput. Aided Des. Integ. Circ. Syst. 18,* 6 (June), 834–849.

Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., and de Simone, R. 2003. The synchronous languages 12 years later. *Proc. IEEE 91,* 1 (Jan.), 64–83.

Berry, G. 1992. Esterel on hardware. *Philosoph. Trans. Roy. Soc. London. Ser. A 339*, 1652, Mechanized Reasoning and Hardware Design, 87–103.

Berry, G. 1999. The constructive semantics of pure Esterel. Draft book.

Berry, G. 2000. *The Esterel v5 Language Primer*. Centre de Mathématiques Appliquées. Part of the Esterel compiler distribution.

Berry, G. and Cosserat, L. 1984. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, S. D. Brooks, A. W. Roscoe, and G. Winskel, Eds. Springer-Verlag, Heidelberg, 389–448.

Berry, G. and Gonthier, G. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Programm. 19,* 2 (Nov.), 87–152.

Bertin, V., Poize, M., and Pulou, J. 1999. Une nouvelle méthode de compilation pour le language ESTEREL [A new method for compiling the Esterel language]. In *Proceedings of GRAISyHM-AAA* (Lille, France).

Bhattacharyya, S. S., Buck, J. T., Ha, S., Murthy, P. K., and Lee, E. A. 1993. A scheduling framework for minimizing memory requirements of multirate DSP systems represented as dataflow graphs. In *Proceedings of the IEEE Workshop on VLSI Signal Processing VI*. The Institute of Electrical and Electronics Engineers (IEEE), Veldhoven, The Netherlands, 188–196.

Bhattacharyya, S. S., Leupers, R., and Marwedel, P. 2000. Software synthesis and code generation for signal processing systems. *IEEE Trans. Circ. Syst.—II: Analog Digital Signal Process. 47,* 9 (Sept.), 849–875.

Bhattacharyya, S. S., Murthy, P. K., and Lee, E. A. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer, Boston.

Bhattacharyya, S. S., Murthy, P. K., and Lee, E. A. 1999. Synthesis of embedded software from synchronous dataflow specifications. *J. VLSI Signal Process Syst. 21,* 2 (June), 151–166.

BILSEN, G., ENGELS, M., LAUWEREINS, R., AND PEPERSTRAETE, J. A. 1995. Cyclo-static data flow. In *Proceedings of the IEEE International Conference on Acoustics, Speech, & Signal Processing (ICASSP)* (Detroit), 3255–3258.

BOUSSINOT, F. AND DOUMENC, G. 1992. RC reference manual.

BRIAND, L. P. AND ROY, D. M. 1999. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. IEEE Computer Society Press, New York.

BRYANT, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. C-35*, 8 (Aug.), 677–691.

BRZOZOWSKI, J. A. 1964. Derivatives of regular expressions. *J. ACM 11*, 4 (Oct.), 481–494.

BUCK, J. T. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD Thesis, University of California, Berkeley. Available as UCB/ERL M93/69.

BUCK, J. T. 1994. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems & Computers*. The Institute of Electrical and Electronics Engineers (IEEE), Pacific Grove, CA, 508–513.

CARDELLI, L. AND PIKE, R. 1985. Squeak: A language for communicating with mice. In *Proceedings of the Twelfth ACM Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (San Francisco), ACM, New York.

CASPI, P., PILAUD, D., HALBWACHS, N., AND PLAICE, J. A. 1987. LUSTRE: A declarative language for programming synchronous systems. In *ACM Symposium on Principles of Programming Languages (POPL)* (Munich), ACM, New York.

CHIODO, M., GIUSTO, P., JURECSKA, A., LAVAGNO, L., HSIEH, H., SUZUKI, K., SANGIOVANNI-VINCENTELLI, A., AND SENTOVICH, E. 1995. Synthesis of software programs for embedded control applications. In *Proceedings of the 32nd Design Automation Conference* (San Francisco), ACM, New York, 587–592.

CORTADELLA, J., KONDRATYEV, A., LAVAGNO, L., MASSOT, M., MORAL, S., PASSERONE, C., WATANABE, Y., AND SANGIOVANNI-VINCENTELLI, A. 1999. Task generation and compile-time scheduling for mixed data-control embedded software. Tech. Rep. LSI-99-47-R, Department of Software, Universitat Politècnica de Catalunya. November.

CORTADELLA, J., KONDRATYEV, A., LAVAGNO, L., MASSOT, M., MORAL, S., PASSERONE, C., WATANABE, Y., AND SANGIOVANNI-VINCENTELLI, A. 2000. Task generation and compile-time scheduling for mixed data-control embedded software. In *Proceedings of the 37th Design Automation Conference* (Los Angeles), ACM, New York, 489–494.

EDWARDS, S. A. 2000. Compiling Esterel into sequential code. In *Proceedings of the 37th Design Automation Conference* (Los Angeles), ACM, New York, 322–327.

EDWARDS, S. A. 2002. An Esterel compiler for large control-dominated systems. *IEEE Trans. Comput. Aided Des. Integ. Circ. Syst. 21*, 2 (Feb.), 169–183.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July), 319–349.

FRENCH, R. S., LAM, M. S., LEVITT, J. R., AND OLUKOTUN, K. 1995. A general method for compiling event-driven simulations. In *Proceedings of the 32nd Design Automation Conference* (San Francisco), 151–156.

FUJIMOTO, R. M. 1980. Parallel discrete event simulation. *Commun. ACM 33*, 10 (Oct.), 30–53.

GONTHIER, G. 1988. 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.

HACHTEL, G. D. AND SOMENZI, F. 1996. *Logic Synthesis and Verification Algorithms*. Kluwer, Boston.

HALBWACHS, N. 1993. *Synchronous Programming of Reactive Systems*. Kluwer, Boston.

HALBWACHS, N., CASPI, P., RAYMOND, P., AND PILAUD, D. 1991a. The synchronous data flow programming language LUSTRE. *Proc. IEEE 79*, 9 (Sept.), 1305–1320.

HALBWACHS, N., RAYMOND, P., AND RATEL, C. 1991b. Generating efficient code from data-flow programs. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, Lecture Notes in Computer Science, vol. 528, Springer-Verlag, Passau, Germany.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8*, 3 (June), 231–274.

Harel, D. and Naamad, A. 1996. The Statemate semantics of Statecharts. *ACM Trans. Softw. Eng. Method. 5*, 4 (Oct.), 293–333.

Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall, Upper Saddle River, NJ.

Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Hudak, P. 1998. Modular domain specific languages and tools. In *Proceedings of the Fifth International Conference on Software Reuse (ICSR)* (Victoria, Canada), 134–132.

IEEE Computer Society. 1996. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Computer Society, New York, 1364–1995.

Jones, N. D., Gomard, C. K., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Upper Saddle River, NJ.

Kahn, G. 1974. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, North-Holland, Stockholm, 471–475.

Kahn, G. and MacQueen, D. B. 1977. Coroutines and networks of parallel processes. In *Information Processing 77: Proceedings of IFIP Congress 77*. North-Holland, Toronto, 993–998.

Labrosse, J. 1998. *MicroC/OS-II*. CMP Books, Lawrence, KS.

Le Guernic, P., Gautier, T., Le Borgne, M., and Le Maire, C. 1991. Programming real-time applications with SIGNAL. *Proc. IEEE 79*, 9 (Sept.), 1321–1336.

Lee, E. A. and Messerschmitt, D. G. 1987a. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput. C-36*, 1 (Jan.), 24–35.

Lee, E. A. and Messerschmitt, D. G. 1987b. Synchronous data flow. *Proc. IEEE 75*, 9 (Sept.), 1235–1245.

Lee, E. A. and Parks, T. M. 1995. Dataflow process networks. *Proc. IEEE 83*, 5 (May), 773–801.

Lengauer, T. and Tarjan, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1 (July), 121–141.

Lin, B. 1998. Efficient compilation of process-based concurrent programs without run-time scheduling. In *Proceedings of Design, Automation, and Test in Europe (DATE)* (Paris), 211–217.

Madisetti, V. K., Walrand, J. C., and Messerschmitt, D. G. 1991. Asynchronous algorithms for the parallel simulation of event-driven dynamical systems. *ACM Trans. Model. Comput. Simul. 1*, 3 (July), 244–274.

McMillan, K. L. 1992. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Computer Aided Verification, Fourth International Workshop*, Lecture Notes in Computer Science, vol. 663, Springer-Verlag, Montreal, 164–177.

McMillan, K. L. 1995. A technique of state space search based on unfolding. *Formal Meth. Syst. Des. 6*, 1 (Jan.), 45–65.

Murata, T. 1989. Petri nets: Properties, analysis, and applications. *Proc. IEEE 77*, 4 (April), 541–580.

Parks, T. M. 1995. Bounded scheduling of process networks. PhD Thesis, University of California, Berkeley. Available as UCB/ERL M95/105.

Petri, C. A. 1962. Kommunikation mit automaten. PhD Thesis, Institutes für Instrumentelle Mathematik, Bonn, Germany. In German.

Plotkin, G. D. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Aarhus University, Åarhus, Denmark.

Rudell, R. 1993. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)* (San Jose, CA.), 42–47.

Sgroi, M., Lavagno, L., and Sangiovanni-Vincentelli, A. 1998. Quasi-static scheduling of free-choice Petri nets. Tech. Rep. UCB/ERL M98/9, University of California, Berkeley.

Sgroi, M., Lavagno, L., Watanabe, Y., and Sangiovanni-Vincentelli, A. 1999. Synthesis of embedded software using free-choice Petri nets. In *Proceedings of the 36th Design Automation Conference* (New Orleans), 805–810.

Shiple, T. R., Berry, G., and Touati, H. 1996. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Test Conference* (Paris), 328–333.

SILBERSCHATZ, A. AND GALVIN, P. B. 1998. *Operating System Concepts*, 5th ed. Addison-Wesley, Reading, MA.

SIMONS, B. AND FERRANTE, J. 1993. An efficient algorithm for constructing a control flow graph for parallel code. Tech. Rep. TR–03.465, IBM, Santa Teresa Laboratory, San Jose, CA, February.

STEENSGAARD, B. 1993. Sequentializing program dependence graphs for irreducible programs. Tech. Rep. MSR-TR-93-14, Microsoft. October.

STREHL, K., THIELEAND, L., ZIEGENBEIN, D., ERNST, R., AND TEICH, J. 1999. Scheduling hardware/software systems using symbolic techniques. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES '99)* (Rome), 173–177.

TANENBAUM, A. S. 1992. *Modern Operating Systems*. Prentice-Hall, Upper Saddle River, NJ.

THIBAULT, S. A., MARLET, R., AND CONSEL, C. 1999. Domain-specific languages: From design to implementation application to video device drivers generation. *IEEE Trans. Softw. Eng. 25*, 3 (May), 363–377.

THOMAS, D. E. AND MOORBY, P. R. 1998. *The Verilog Hardware Description Language*, 4th ed. Kluwer, Boston.

ULRICH, E. G. 1965. Time-sequenced logic simulation based on circuit delay and selective tracing of active network paths. In *Proceedings of the Twentieth ACM National Conference*, 437–448.

ULRICH, E. G. 1969. Exclusive simulation of activity in digital networks. *Commun. ACM 12*, 2 (Feb.), 102–110.

WEIL, D., BERTIN, V., CLOSSE, E., , POIZE, M., VENIER, P., AND PULOU, J. 2000. Efficient compilation of Esterel for real-time embedded systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (San Jose, CA), 2–8.

ZHU, X. AND LIN, B. 1999. Compositional software synthesis of communicating processes. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)* (Austin, TX), 646–651.

Received October 2001; revised January 2003; accepted January 2003