Compiling Concurrent Languages for Sequential Processors

Stephen A. Edwards
Computer Science Department, Columbia University
1214 Amsterdam Avenue
New York, NY 10027
sedwards@cs.columbia.com

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 paper 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 technique, 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.

---

1. INTRODUCTION

Although finding parallelism in a sequential algorithm is useful when compiling to a parallel machine, embedded software often needs to be translated 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 sequential software.

Operating systems traditionally manage concurrent tasks on a single processor. Timesharing operating systems strive for fair scheduling [Silberschatz and Galvin 1998; Tanenbaum 1992], while 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]. An OS-based approach is appropriate for general-purpose computers designed to handle arbitrary applications, but can be wasteful for embedded systems where the application is known and fixed. Specifically, an OS wastes time scheduling and context switching.

This paper 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 to produce a faster, more predictable system. Lin [1998] showed an eighty-fold speedup on one example.

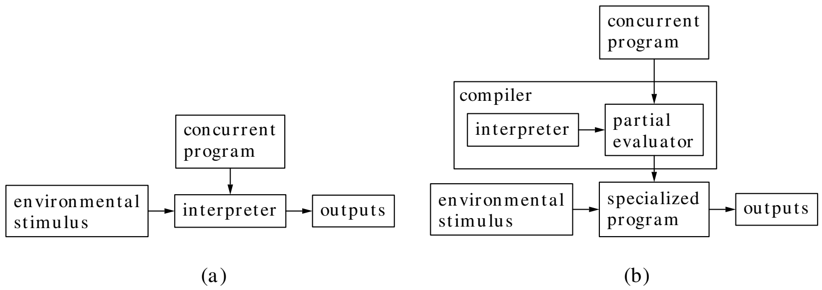


Fig. 1. The flow (a) of an interpreter and (b) of a partial evaluator acting as a compiler. The compilers in this paper are considered to have this structure.

Compiling concurrent specifications into sequential code also finds application outside embedded software. Almost every digital hardware design, which is naturally concurrent, is first simulated on sequential hardware before it is fabricated to check for errors. Also, certain compilers for sequential languages prefer to optimize a concurrent version of the sequential code since it exposes opportunities for code restructuring. The restructured concurrent code must be translated back to sequential code to finish the compilation process.

The compilation techniques presented here are not generally interchangable since each depends greatly on the particular concurrent language being compiled. As a result, I do not make any quantitative comparisons.

1.1 Partial Evaluation

To compare these compilation techniques, I propose each be thought of a partial evaluator operating on an interpreter as shown in Figure 1b. Although all these concurrent programs could be run by an interpreter (Figure 1a), this approach is rarely efficient enough to be practical. Instead, concurrent programs are compiled into sequential programs that a processor can execute directly.

Conceptually, all these approaches use a partial evaluator, which is an algorithm that takes a program along with some of its input data and produces a residual program that, when given the remaining input, behaves like the original program would were it given the data all at once (see Jones, Gomard, and Sestoft [1993] for an introduction). A partial evaluator running on an interpreter is a compiler, and any compiler can be thought of in this way.

Although few of the compilers presented in this paper contain components that are clearly interpreters or partial evaluators, conceiving of each compiler as consisting of these two components is useful as it allows us to 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.

The partial evaluation techniques used implicitly in these compilers are more powerful than general approaches because they have been tuned to the particular interpreter they are being applied to. 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 con-

```
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. 2. 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.

current language. Furthermore, these partial evaluators can rise above any 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) generate little more than hard-wired simulators, whereas others attempt to remove all traces of the interpreter (e.g., the PDG-to-CFG translation algorithm) 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 since more cases must be explored.

The most common optimization establishes the order in which parts of the concurrent program will run when the system is compiled. These decisions are costly and thus are excellent candidates for compile-time analysis.

Figure 2 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 2 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 2b. 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 2c produces even faster code. Here, each function returns the address of the next function to call.

Table I. The compilation techniques discussed in this paper

Section	Language	Technique	Communication	Concurrent Model	Sequential Model	Specialized w.r.t.
2.1	Esterel	V5	Synchronous	Logic Network	Levelized Network	Schedule
2.2	Lustre		Synchronous	Flow equations	FSM	Boolean variables
2.3	SDF		Buffered	Multirate DFG	Looped sequence	Schedule
3.1	Esterel	V3	Synchronous	CFG	FSM of CFGs	State, signals
3.2	C, etc.	PDG to CFG	Memory	PDG	CFG	Schedule
3.3	Esterel	EC	Synchronous	CFG	CFG	Schedule
4.1	Verilog	VeriSUIF	Events	Event Graph	Looped sequence	Event queue
4.2	Esterel	SAXO-RT	Synchronous	Event Graph	Event sequence	Schedule
5.1	C variant		Rendezvous	Petri Net	FSM of CFGs	State, schedule
5.2	FlowC		Buffered	Petri Net	FSM of CFGs	State, schedule

1.2 The Techniques

Table I lists the different compilation techniques presented in this paper and also serves as a table of contents. I divide the techniques by the type of concurrent model and order them by increasing complexity of control relationships in the model. The first group usees dataflow models that concentrate on how information flows between operations, which are all assumed to run all the time. Every action is performed at the same rate in the simpler models, but Synchronous Dataflow (SDF) adds multi-rate behavior to provide a richer and harder-to-compile model. The second group operates on concurrent control-flow graphs, which specify a sequence of actions and decisions: control flows through a (possibly cyclic) path, performing the action at each node along the way. Event graphs, used by the third group, support more complicated relationships between actions, such as causing an action to run later, prohibiting an action, or modifying the conditions under which an action may run. The fourth class of concurrent model, the Petri net, is a formal, flexible concurrent specification centered around the idea of allowing each transition to fire only if the system is in an enabling state. They are well-suited for describing rendezvous communication, which forces two communicating processes to reach a common point before proceeding, and buffered communication with data-dependent decisions.

2. COMPILING DATAFLOW

This first collection of compilers work on dataflow representations, which specify how data moves between parts of the system and only vaguely define sequencing. A dataflow specification is a collection of function-like blocks connected by data pathways. All blocks typically run all the time. The challenge of executing dataflow systems comes in deciding the order and rate at which to execute the blocks.

The dataflow models described in this section range from the very simple to fairly complicated. Although a combinational logic network is a simple form of dataflow, it is powerful enough to model Esterel program behavior. The

Lustre language is a slight step up: it can describe integer and floating-point arithmetic and one-cycle delays, but only uses wire-like communication. Synchronous Dataflow supports multi-rate behavior through multi-rate functions that communicate through buffered channels.

Generating sequential code for a single-rate dataflow specification is straightforward: communication imposes dependencies between blocks, and a topological sort of the blocks can determine the order in which to run them. Since every block always runs, generating code is no more complicated than listing the code for each block in that order (e.g., Figure 5). Multi-rate behavior greatly complicates things by requiring some blocks to run more frequently than others. Since cyclic dependencies are permitted (provided there is sufficient slack in the buffered communication to avoid a deadlock) the order in which the blocks may be fired becomes subtle.

2.1 Compiling Esterel into a Combinational Logic Network

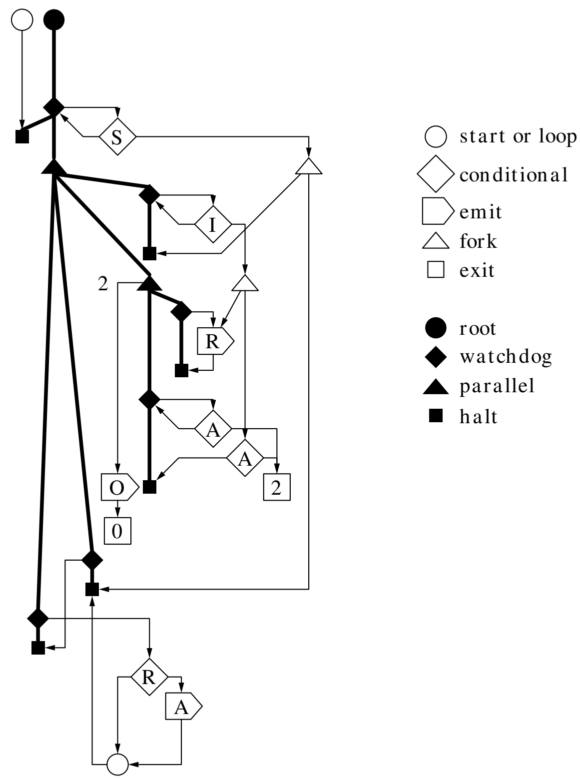
A combinational logic network is a single-rate dataflow specification with simple, powerful semantics, can compactly represent most functions, and can easily be translated into sequential software. For these reasons, it is a practical intermediate representation for concurrent software. This section describes the V5 compiler for the Esterel language built using this principle.

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 like wires: they are either present or absent each cycle and their values do not persist between cycles.

Figure 3a 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 3b shows how this Esterel program is represented in the IC format, a control-flow graph (outlined nodes and arcs) hanging from a "reconstruction tree" that handles Esterel's concurrency and preemption (black nodes and arcs). Originally designed by Gonthier [1988] for the automata-based V3 compiler (Section 3.1), 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 second 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 ter-

```
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. 3. (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, which the V3 compiler simulates to produce the automaton code in Figure 13. 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.

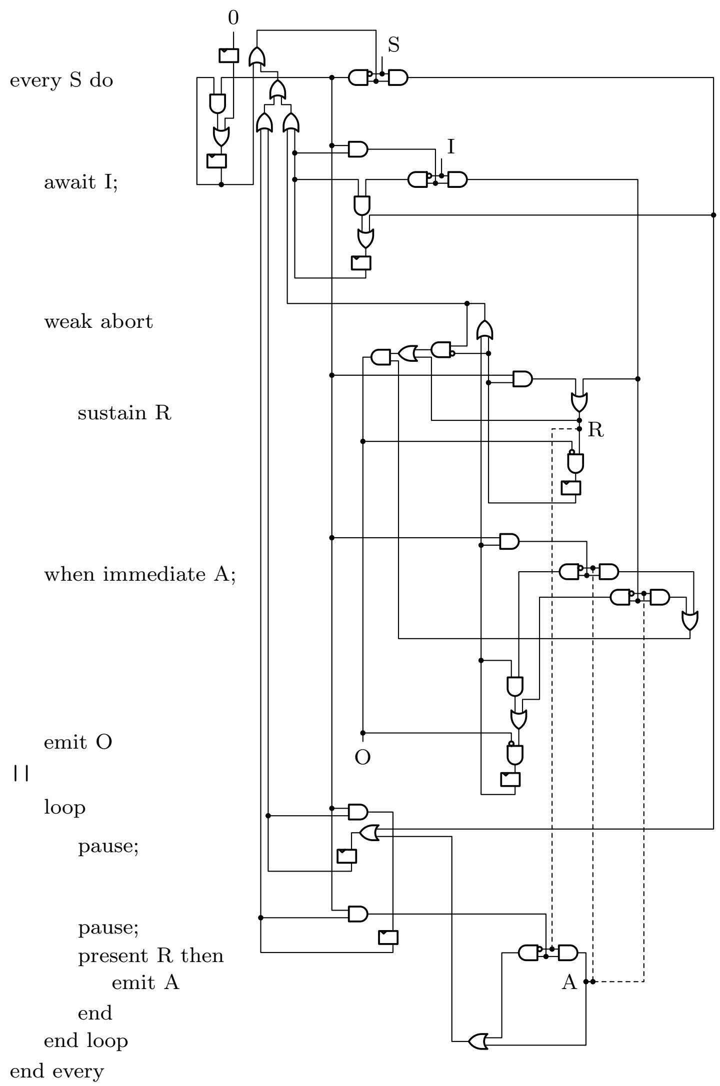


Fig. 4. The circuit the V5 compiler generates for an Esterel program represented by the IC graph in Figure 3b. Registers implement halts: points where the program will restart in the next cycle. Dashed wires communicate signals; all others represent control. The unusual layout of this circuit follows the structure of Figure 3b. Removing the registers leaves the circuit acyclic.

mination or exceptions (e.g., *weak abort*). Eventually control returns to the root node to indicate the program is done for the cycle.

The semantics of the IC format are complex (see my forthcoming paper [Edwards 2001] for more details) but can be translated fairly directly into a circuit such as Figure 4. 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

```
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. 5. A fragment of the code generated by the V5 compiler for the circuit in Figure 4. 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.

matches the reconstruction tree, eventually forming the activation condition for the whole program. This output 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 5 shows such a fragment. It is nothing more than code for each gate listed in topological order.

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 6. 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 since 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. 6. 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 overview) 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, Berry, and Touati [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 (potentially two orders of magnitude slower than other approaches—see Edwards [2001]) 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 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) since zeros that indicate statements do not run are propagated.

The V5 compiler implicitly assumes the network does roughly the same thing each cycle. While this is often true for signal processing applications, control-dominated Esterel programs often do radically different things in different cycles. So the generated code contains many conditionals that decide whether a particular thing should happen. Instead, generating separate code for each behavior can significantly improve performance, an idea used by the compiler in the next section.

2.2 Compiling Lustre

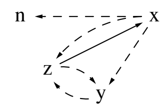
A concurrent representation can be compiled into an automaton. A compiler builds such an automaton by simulating the program in each state; recording its behavior, which becomes the code executed in that state; and determining which states follow the one being simulated. An interpreter performs the simulation, and a partial evaluator coordinates automaton construction. Since part of the program's state is known, the code for each state can be simplified it can deal with constants instead of variables. The concurrency disappears because the simulator returns a sequence of instructions. The advantage of this approach is the high speed of the generated code, which is usually more specialized that the source and simulates none of the interpreter's state. The drawback is that most concurrent systems have an enormous number of states—often exponentially more than the size of the source program. This can produce enormous executables and require long compilation times, making it practical only for small examples.

Since each state in the automaton has behavior, a compiler may choose the granularity of the states it generates. For example, a state can contain variable values and control points, but tracking all of these generates more states. Tracking just control points seems to be a good compromise: the behavior of each state does the data manipulation.

Two techniques can reduce the number of states and hence the size of the generated code from an automata compiler. First, some states generated by the simulation algorithm may be redundant and can be merged. Such states are generated when the program build state that it never observes, which may be easy or difficult depending on the language. Second, reducing the amount of state the simulator tracks can greatly reduce the number of states considered. In general, tracking "control" state that substantially changes the behavior of the program is beneficial: different behaviors can be cleanly separated into different states. By contrast, tracking "data" state such as integer variables on which arithmetic is performed generally greatly increases code size without substantially improving speed. The Lustre compiler presented in this section employs both these techniques: it avoids redundant states as it is simulating the system and chooses to only track boolean variables.

```
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. 7. (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.

Lustre [Caspi et al. 1987; Halbwachs et al. 1991] is a dataflow language whose programs are a set of definitions (e.g., Figure 7a), each assigning the value of an expression to a variable. It is a synchronous language, so all the definitions are evaluated simultaneously in each clock cycle. 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, e.g., `0->pre(x)` is 0 in the first cycle, and the previous value of `x` in later cycles.

Halbwachs, Raymond, and Ratel [1991] describe how to compiler Lustre programs. Like 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 specifically prohibits cyclic dependencies, so a topological sort always exists.

Figure 7b shows the dependency graph for the program in Figure 7a. First, note that there is only one direct dependency (the solid line), since 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 7b corresponds to forcing every use of a variable's past value to come before it its new value is computed. Doing this to every variable in Figure 7a this 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 `z` to `y`, resulting in an acyclic graph that can be scheduled. Figure 8a shows the code this produces.

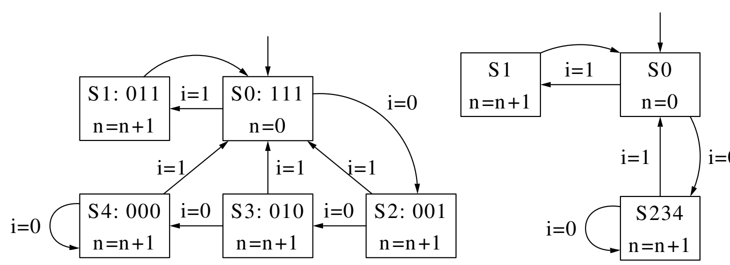
```
int n = 0;
bool x = 1;
bool y = 1;
bool z = 1;
bool pre_z;

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. 8. (a) A straightforward C implementation of the Lustre program in Figure 7a based on the schedule $\{n, z, y, x\}$. (b) An implementation factored with respect to x.



(a) (b)

Fig. 9. (a) An automaton for the Lustre program in Figure 7a. Each state is labeled with the previous values of x, y, and y along with the rule for updating n. (b) A minimal automata for the same program. States S2, S3, and S4 were merged.

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 8a are redundant. Factoring them as in Figure 8b will improve both the size and speed of the program, but further optimization is possible. The test of x in Figure 8b is still somehow redundant since 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 since 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 7a produces the automaton in Figure 9a. Each state is labeled with the values of x, y, and z in the previous state along with the rule for updating n in that state. For example, from the initial state

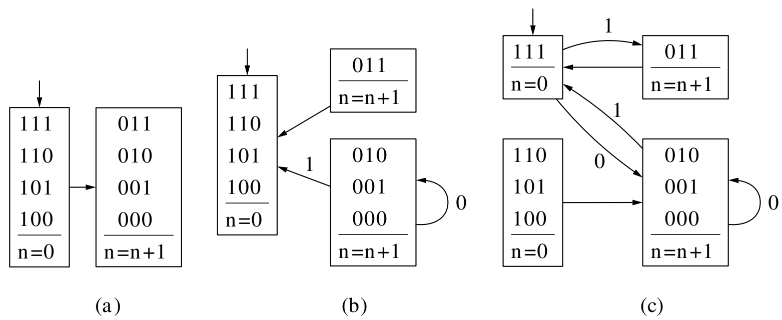


Fig. 10. Partitioning the state space of the Lustre program in Figure 7a. Each state is labeled with the concrete states it represents (values of x, y, and z) and the action it performs. (a) Divide according to the two different types of output. (b) Split the successor of the initial state into two, since 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.

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 simulation are correct but often redundant. For example, in Figure 9a, states S2, S3, and S4 are equivalent and can be merged to form Figure 9b without affecting behavior. Such non-minimality does not affect the speed of the generated code, only its size. Such 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, Raymond, and Ratel [1991] is an algorithm for minimizing the automaton as is it generated. This improves compilation speed, capacity, and code size since the non-minimal 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 7a. 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 10a).

Now consider the successors of the state on the right of Figure 10a. Its successors depend both on the input i and whether y and z are both true. Thus, this state is split to produce Figure 10b and its successor considered.

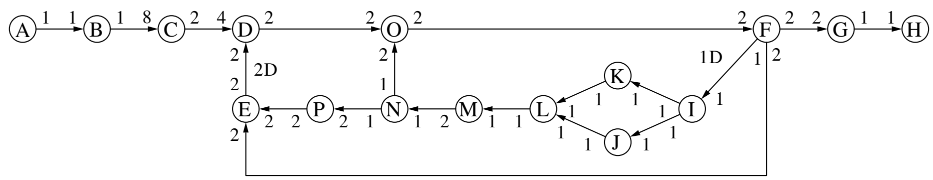
Now the initial state is split, since its successors depend on i only when x, y, and z are all true (Figure 10c). The process is finished since the successors of these states do not need to be split.

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 11 shows the code generated from the automaton in Figure 10c. Note that one of the states in Figure 10c is unreachable and was been 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. 11. The final code generated from the minimized automaton in Figure 10c.



(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. 12. (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])

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 non-equivalent 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 system operates at a constant rate. Compiling Synchronous Dataflow, described in the next section, is harder because parts of the system may operate at different rates.

2.3 Compiling Synchronous Dataflow

Synchronous Dataflow (SDF) [Lee and Messerschmitt 1987b] is a multi-rate 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 (FIFO) queues. Figure 12a shows a typical multi-rate 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 each times each block will fire per schedule period by solving the constraints imposed 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 12a, then $2c - 4d = 0$ must be satisfied. Each arc leads to a similar equation, and the resulting system of equations is called the set of 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 12a, this solution is $a = b = 16$, $c = n = 2$, and all others 1.

That the balance equations have a non-zero solution is necessary but not sufficient for a schedule to exist, since the system might deadlock. Fortunately, Lee and Messerschmitt [1987a] show any algorithm that correctly simulates buffer behavior (i.e., firing a block only when enough tokens are available) will always find a schedule if one exists, i.e., 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 12c. 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 12a has the looped schedule in Figure 12b.

Most SDF schedulers seek a single-appearance schedule (SAS)—one in which each block appears exactly once—since it results in minimum code size assuming no loop overhead. 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 a SAS for that portion of the graph, so an SDF graph is recursively divided into strongly-connected components (SCCs). Bhattacharyya, Buck, Ha, Murthy, and Lee [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, Leupers, and Marwedel [2000] provide a recent summary of work in this area. Also of interest is the book by Bhattacharyya, Murthy, and Lee [1996].

3. COMPILING CONCURRENT CONTROL-FLOW

Dataflow specifications are good for a certain class of applications, and are fairly easily translated into sequential code, but they are not a perfect match for how a typical sequential processor executes. A control-flow graph—a flowchart such as Figure 18a—is a much closer match.

The compilers in this section operate on concurrent variants of control-flow graphs. Unlike a sequential control-flow graph, in which exactly one statement is executing at any time, control may be at multiple points in a concurrent control-flow graph. After the action at an active node is performed, control is passed to one of its successors. Such semantics are close to those of a standard processor except for the concurrency—the ability for control to be at more than one node at once. Thus, these compilers' task is to build a sequential control-flow graph that executes the concurrent nodes in an appropriate order.

The compilers presented here take three different approaches to compiling CCFGs. The compilers discussed in the first section build an automaton for the program being compiled by simulating the program in each possible state and recording the results, much like the Lustre compiler in Section 2.2. This works for the Esterel language, which divides its execution into cycles: the state is the positions of all the program counters at the beginning of a cycle. This technique produces very fast code that tends to grow exponentially large. The partial evaluation technique is straightforward: for each possible state, run the interpreter, record the sequence of instructions executed, and build a routine that executes each instruction in turn. The main program calls each routine based on the current state.

The second section discusses compiling program dependence graphs, which are a more abstract form of CCFG used originally to represent a sequential program. The goal here is a concise CFG: one that simulates a concurrent PDG with zero overhead. While concise CFGs often exist when the PDG was translated from a sequential specification, most concurrent specifications do not have one. The main challenge here is scheduling: determining an order that is both correct and does not require any additional decisions. The partial evaluation in this scheme is aggressive: if scheduling is successful, no code for manipulating the interpreter's state appears in the generated code.

The third section discusses a compiler that attempts a compromise: it compiles Esterel without duplicating code, but is willing to insert additional code when necessary to ensure correct behavior. Compared to the automaton compilers for Esterel, the partial evaluation in this compiler is less aggressive: more of the interpreter's state specifically program counters, is manipulated by the generated code. While this code is slightly slower, it can be orders of magnitude smaller.

```
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. 13. Automata-style code generated by the V3 compiler for the program in Figure 3a. 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.

3.1 Compiling Esterel into Automata

The synchronous semantics of Esterel make it an obvious choice to compile into an automaton. The compilers in this section do this in a variety of ways that differ in how they interpret the language and the character of the generated code (Table II).

The synchronous nature of Esterel suggests dividing states on cycle boundaries. Eeach state represents the set of control points where the program can reawaken at the beginning of a cycle. Although the states do not encode information about signals because their values do not persist between cycles, the values of internal signals are compiled away because the simulator tracks their values within a cycle. The code for each state checks and emits external signals, makes decisions, and manipulates internal data. Figure 13 shows the automata-style code for the program in Figure 3a, which can be derived by more than one compiler.

Four techniques for compiling Esterel using automata have been developed (Table II). 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 structural rewriting style [Plotkin 1981]. These semantics give rules for rewriting a program into another program that behaves like the original

Table II. 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	Network	Routine per state

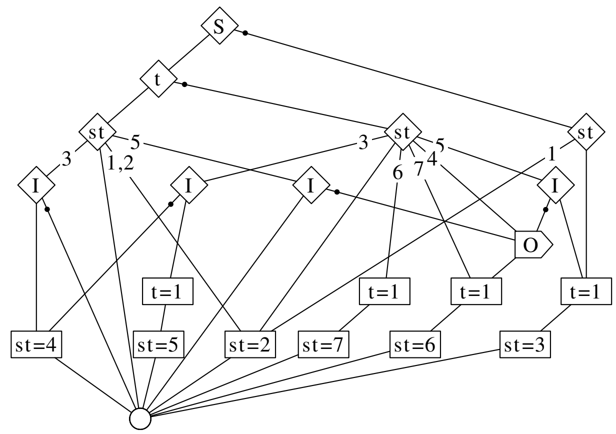


Fig. 14. The control-flow graph generated by the Polis system for the program in Figure 3a. S, I, and O are interface signals, st represents the current state, and t represents a selftrigger flag that indicates the program will run in the next cycle even if no input signal is true. Arcs with dots represent true branches. Since 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.

program in the next cycle. This rather literal implementation was based an idea from Brzozowski [1964] 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 but much faster V3 technique that simulates an Esterel program in the IC representation (Figure 3b). See Section 2.1 for a discussion of the IC format's semantics.

The V3 uses potentials 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, since 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 14 shows the control-flow graph their compiler generates for the example in Figure 3a.

The Polis approach can generate more compact code than from other automata-based compilers. For example, there is only one test for S in Figure 14, compared with one per state in Figure 13. However, the generated code may still be exponentially larger than the source since each state is represented explicitly.

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 the popular sifting algorithm of Rudell [1993]). 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, since 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 2.1) to derive the behavior of each state.

Unlike those from Lustre, 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 13 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. Despite this, state minimization is not part of the normal Esterel automata compiler flow.

State explosion is automata-based compilation's main problem because code is generated for each separate program state. The compilation technique in the next section goes to the other extreme by insisting that the generated code have a one-to-one correspondence with the source. The result is efficient and compact when it exists.

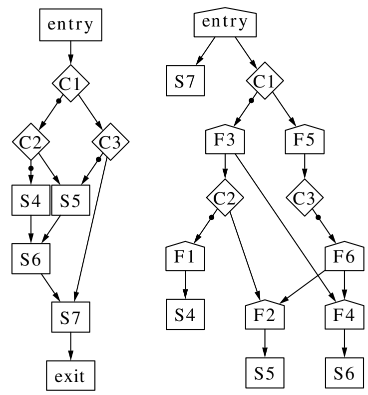
3.2 Building a Control-Flow Graph from a Program Dependence Graph

During optimization, certain compilers for sequential languages represent a program using a concurrent Program Dependence Graph (PDG) [Ferrante et al. 1987]. A PDG (e.g., Figure 15c) represents only the essential data and control dependencies, treating as unordered statements that are always executed under the same conditions. Removing non-essential dependencies exposes opportunities for instruction reordering, but synthesizing a CFG from such an abstract specification requires analyzing subtle relationships between the conditions under which nodes execute.

A PDG (Figure 15c) is a rooted graph whose nodes represent statements, conditionals, and forks, and whose arcs represent control and data dependence. The discussion here will only discuss 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 path in the CDG from the entry to the node that is consistent with the the conditional statements along that path, e.g., if a conditional on the path is true, the path must pass through its true child. For example, in Figure 15c, S6 is executed if C1 is true (path: entry $\rightarrow$ C1 $\rightarrow$ F3 $\rightarrow$

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

(a)

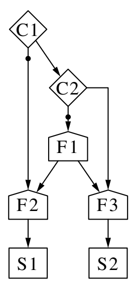


(b) (c)

C1	C2	C3	F2	F4
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. 15. (a) A short procedure. (b) A control flow graph for it. (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.



(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;
}
```

Fig. 16. (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.

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

Compilation algorithms for PDGs focus on constructing concise CFGs, i.e., ones in which each PDG node appears exactly once. For example, Figure 15c has the concise CFG in Figure 15b. When a concise CFG exists, an n-node PDG with e edges can be synthesized in $O(ne)$ time, but implementing all other PDGs, such as Figure 16a, require 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 15c, 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 15c, 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 15d 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 16b shows there are cases in Figure 16a where F2 and F3 run both separately and together. This shows there is no implication relationship, and hence no concise CFG exists for the CDG in Figure 16a. 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. Steensgard 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, in Figure 15c, the subgraph

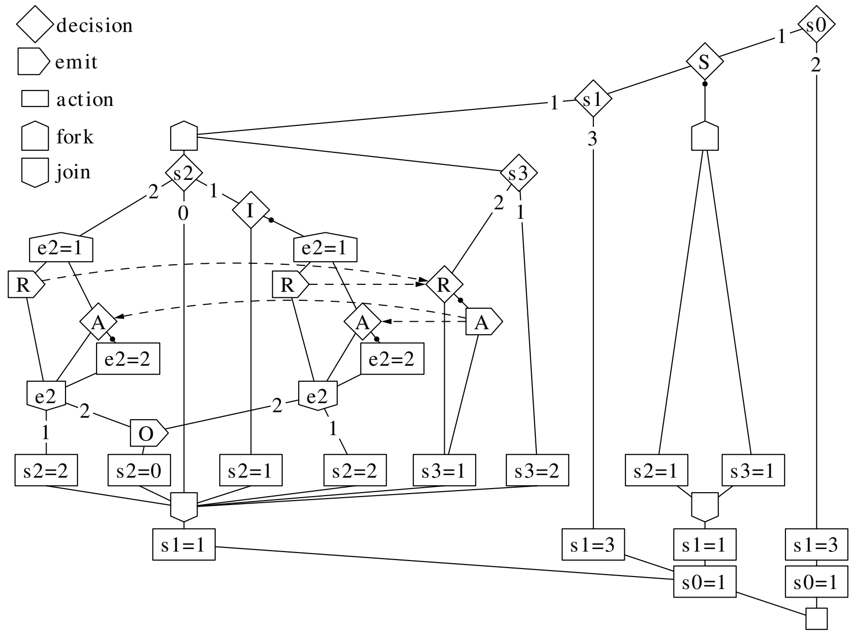


Fig. 17. The concurrent control-flow graph the EC Esterel compiler generates for the program in Figure 3a. Dashed lines represent data dependencies. Variables s0, s1, s2, and s3 store state between cycles; e2 holds the exit level of the group of threads. Initially, s0=2 and all other variables are uninitialized.

at C1 is synthesized first, then S7. Arcs are then connected from the false branch of C3 and from S6 to S7.

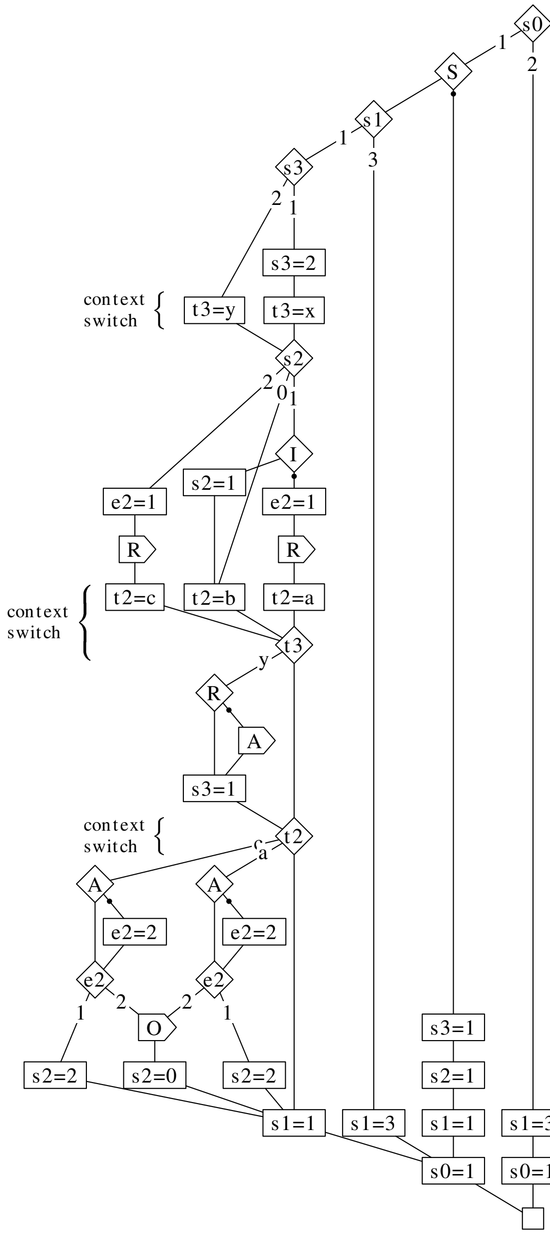
In the partial evaluation framework, the interpreter for a PDG is trivial except for the scheduler. The partial evaluator is the algorithm for building the CFG, which takes the unusual step of working backwards through the program.

A concise CFG, when it exists, is a very efficient sequentialization of a PDG. As expected, concise CFGs often do exist for PDGs generated from sequential code, but do not for more general concurrent specifications. The next section discusses a compiler for Esterel able to generate compact, efficient code when additional conditionals must be added.

3.3 Compiling Esterel into a Control-Flow Graph

My EC compiler [Edwards 2000] translates an Esterel program into a concurrent control-flow graph (CCFG, Figure 17), schedules the nodes in that graph based on control and data dependencies, and then 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 18a) that can easily be translated into procedural code (Figure 18b).

The main challenge in this framework is handling threads that must be interleaved because they communicate back and forth within the same cycle.




```
#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. 18. (a) The sequential control-flow graph the EC Esterel compiler generates for the program in Figure 3b, 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.

Signals R and A in Figure 3a 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 variable representing the program counter of the thread being suspended followed by a multi-way branch on the program counter of the thread being resumed. There are three such context switches in Figure 18a.

The first part of the compiler translates IC (Figure 3b) into a CCFG (Figure 17). 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 multi-way 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 3a 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. While 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 quadradic instead of exponential as with some sequentialization procedures). As such, 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 creating suspension nodes that write the state of the thread being suspended into its program counter, attaching their predecessors to each of the nodes in the thread that have run recently, creating a new node that will become a multi-way branch on the state of the thread being resumed, and attaching arcs from each of the just-created suspension nodes. This procedure is applied recursively because Esterel threads may nest.

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 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 18b).

In the partial evaluation framework, the interpreter in the EC compiler explicitly represents the program counters of each process, and the partial evaluator builds code by considering sets of possible values. Unlike an automata compiler, the partial evaluator makes one pass through the program instead of simulating it repeatedly in different states.

4. EVENT GRAPHS

Event graphs provide more powerful sequencing relationships between nodes than do concurrent control-flow graphs. For example, a part of an event graph may disable another part from running, or may schedule it to be executed in the distant future. These more complicated relationships make it harder to compile, but also provide richer constructs for specification and modeling.

Event graphs are commonly executed by discrete event simulators, which aim to speed system models by only doing work when the system state changes. This works particularly well for modeling the behavior of digital logic gates with delays since at any point in time only a few wires in the system are changing. The basic simulation algorithm maintains a queue of events—changes to some part of the system state—ordered so that sooner events appear first. The simulation loop removes the soonest event and executes it, possibly generating more events.

Compilied discrete-event simulators are common. These divide the program into small fragments and generate a routine for each fragment that is responsible for simulating itself and scheduling other parts of the system. Each routine does this by communicating with a central event queue: the simulation kernel linked in to the final executable. Such an arrangement is an improvement over a pure interpreter since the behavior of each system fragment has been specialized into executable code, but the interaction with and operation of the event queue can still impose substantial overhead.

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

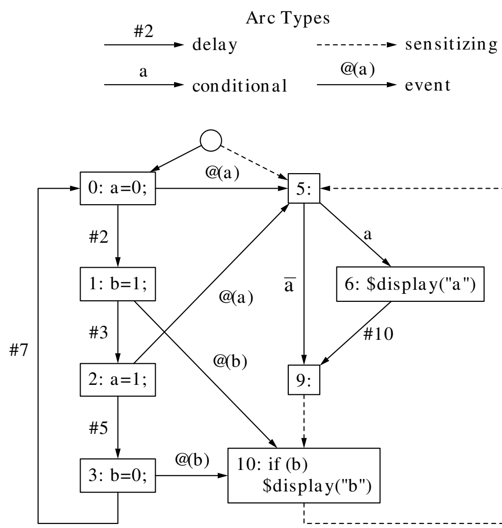
The two compilers described in this section remove the need for a generalized event queue by analyzing its behavior at compile time. The first approach, the VeriSUIF compiler, builds a simple (linear) automaton for the system and generates code for it. When it encounters two or more next states, it adds state to the generated code to track it, ensuring the number of states does not grow exponentially. The second approach, the SAXO-RT compiler for Esterel, similarly specializes the behavior of the event queue.

4.1 Compiling Verilog

The Verilog language [Thomas and Moorby 1998; IEEE Computer Society 1996] provides discrete event semantics to model digital hardware. A Verilog program consists of hierarchically composed modules, each containing in-

```
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. 19. (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.

stances of modules or 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 schedules it to resume in a later time step. Event control statements (e.g., `@(posedge clk))` suspend the process until the designated event occurs. Figure 19a 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` 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 19a begins by waiting for a change on `a`, then prints "a" and waits ten 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.

The VeriSUIF compiler of French, Lam, Levitt, and Olukotun [1995] improves on traditional compiled discrete-event simulators by compiling away the behavior of the event queue. It first translates a Verilog program into an event graph such as Figure 19b, the partially evaluates the interpreter algorithm in Figure 20 to build a linear automaton. The interpreter consists of two loops: the outer runs all the events at the current simulation time and

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. 20. 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.

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 19b depicts the event graph VeriSUIF builds for the Verilog program in Figure 19a. 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 for when the system 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 21 shows the code VeriSUIF generates for the Verilog program in Figure 19a. 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 20 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 problems in situations where a system has zero-delay feedback yet 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 are present. To avoid this problem, a test for whether any events remain is added at the end of the loop in the generated code. If all are absent, the generated code breaks from the loop and proceeds to a new state created by assuming all the offending events are known to be not 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 and very flexible. As such, partially evaluating is often difficult since its behavior can be difficult to predict. By contrast, the Esterel language has fairly predictable behavior and can be understood more completely. The Esterel compiler in the next section makes use of this in an event graph framework.

4.2 Compiling Esterel with Event Graphs

Like Verilog, the Esterel language discussed earlier provides constructs that can impose complicated relationships among instructions. While an Esterel

```
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. 21. Code generated by VeriSUIF for the example in Figure 19a (simplified: actual C code manipulates Verilog's four-valued variables).

program can be represented with a variant of a concurrent control-flow graph (IC, Figure 3b), it can also be represented with an event graph (Figure 23). The SAXO-RT Esterel compiler from the France Telecom R&D group [Bertin et al. 1999; Weil et al. 2000], takes this approach.

The compiler 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 23, if signal I is present and node f3 is active (runs), the *weak abort* and *sustain R* instructions should run in the same cycle. The "enable current" arcs from f3 to f7 and f4 indicate this.

"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 program 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 Esterel. However, some valid programs may require different instruction (node) orders in different states. The automata compilers allow this since they consider each state separately. The gate-based compilers employ a sophisticated analysis and resynthesis technique that allows them to remove false ordering cycles.

```
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. 22. 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.

This compiler has no such facility, but a more dynamic scheduling technique might permit it.

In the partial evaluation framework, the SAXO-RT compiler is fairly straightforward. The interpreter, shown in Figure 22, is simpler than VeriSUIF's, and the partial evaluator does little more than schedule the nodes (usually possible in Esterel, but much harder in Verilog) and generate code for each. In this sense, SAXO-RT is more like a traditional compiled discrete event simulator.

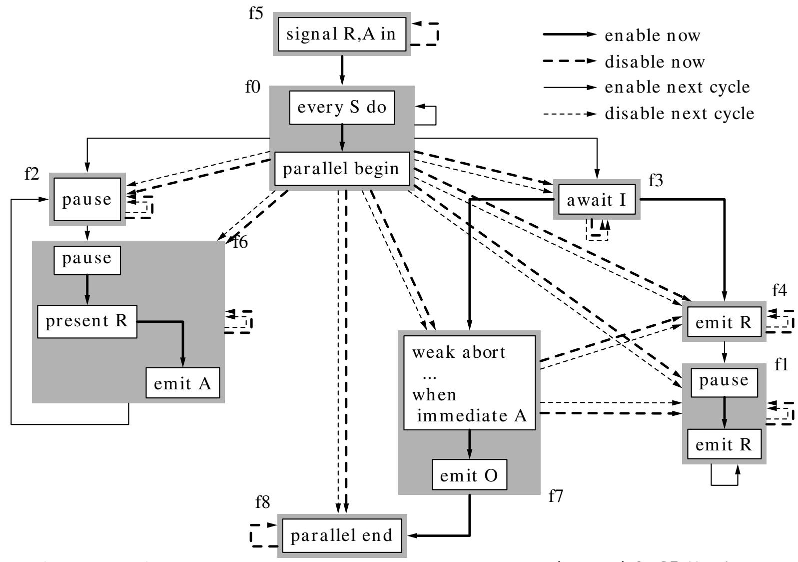
5. COMPILING PETRI NETS

A Petri net (Figure 25c–g) is a general formalism for defining sequencing relationships in concurrent systems. 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 the heroic survey of Murata [1989], with no fewer than 315 references. Petri [1962] started it all.

A Petri Net is a bipartite directed graph whose nodes are either transitions or places. Its semantics are beautifully simple (Figure 24): 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 good for 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 the two languages presented in this section. Transitions represent actions and places represent control points between instructions.

The languages presented here use two different types of communication. The first, due to Lin, uses Hoare's rendezvous communication [Hoare 1985]. The second uses buffered communication, which means it must consider and avoid overflowing buffers.




```
#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. 23. (top) The control-flow graph the SAXO-RT compiler generates for the example in Figure 3b. Each gray area is a node that becomes a single function in the generated code (bottom). Control and communication dependencies between these groups dictates the order in which they appear in the `tick()` function.

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

5.1 Compiling Lin's CSP Language

Lin [1998] proposed a concurrent language that extends C with CSP-style rendezvous communication: a transmitter will block until the receiver is ready to receive and a receiver will block until the transmitter sends data. These semantics are elegantly captured by a Petri Net, which Lin uses as an intermediate representation.

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 (Figure 25a and b), Lin transforms them into Petri nets that represent their control flow (Figure 25c 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 25d, 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 conceivable rendezvous with any receive on channel `c`. This is 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 25f 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 25h. 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. Since an ME is acyclic by construction, a schedule is simply an order that ensure 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.

```
myprocess (int n, out-
put 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 (in-
put chan(int) nn) {
  int k;
    for (;;) {
      k = <-
nn; /* read from nn */
      if (k>0) {
        while (--
k>0) p();
      q();
      }
    }
}
```

(b)

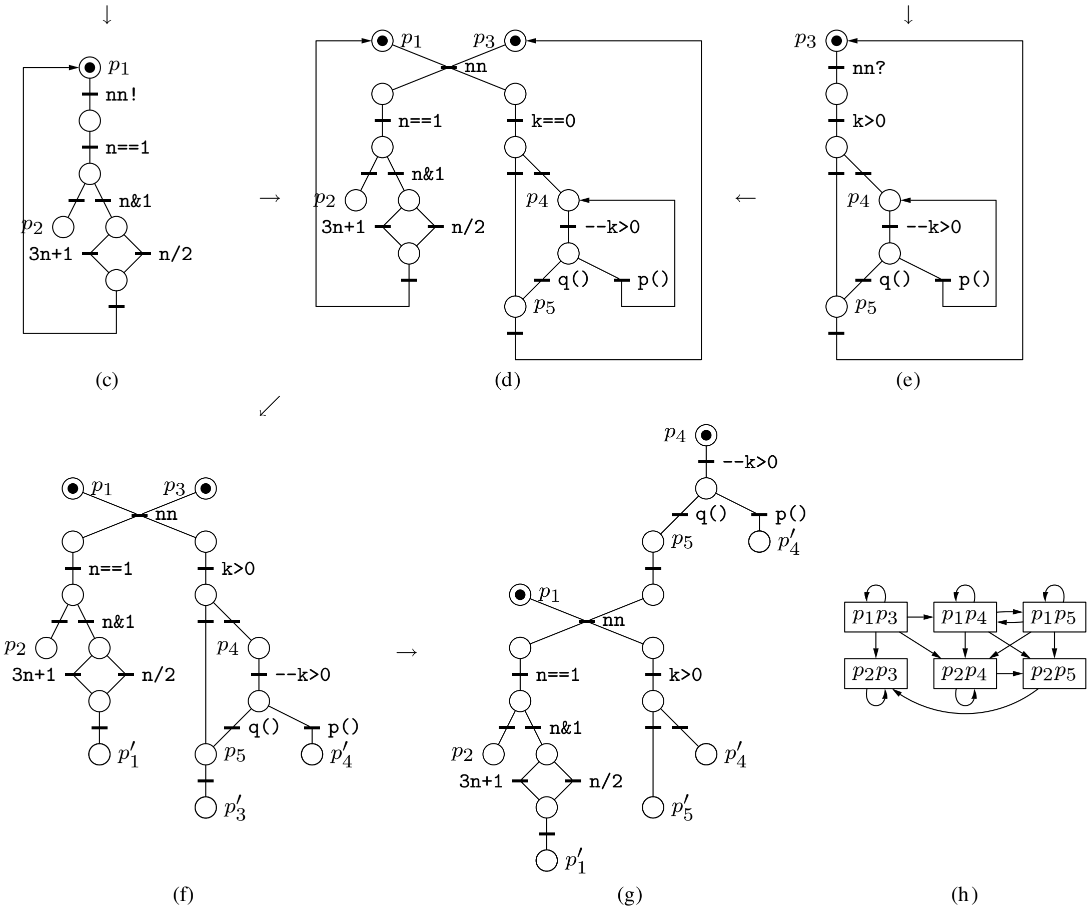


Fig. 25. Building a finite-state model of a system in Lin's language. Processes (a) and (b) are transformed into Petri nets (c) and (e) and fused to form (d). The maximal expansion segment (f) is generated by unrolling (d) starting at the initial marking p1p3. The ME (g) is generated by unrolling (d)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

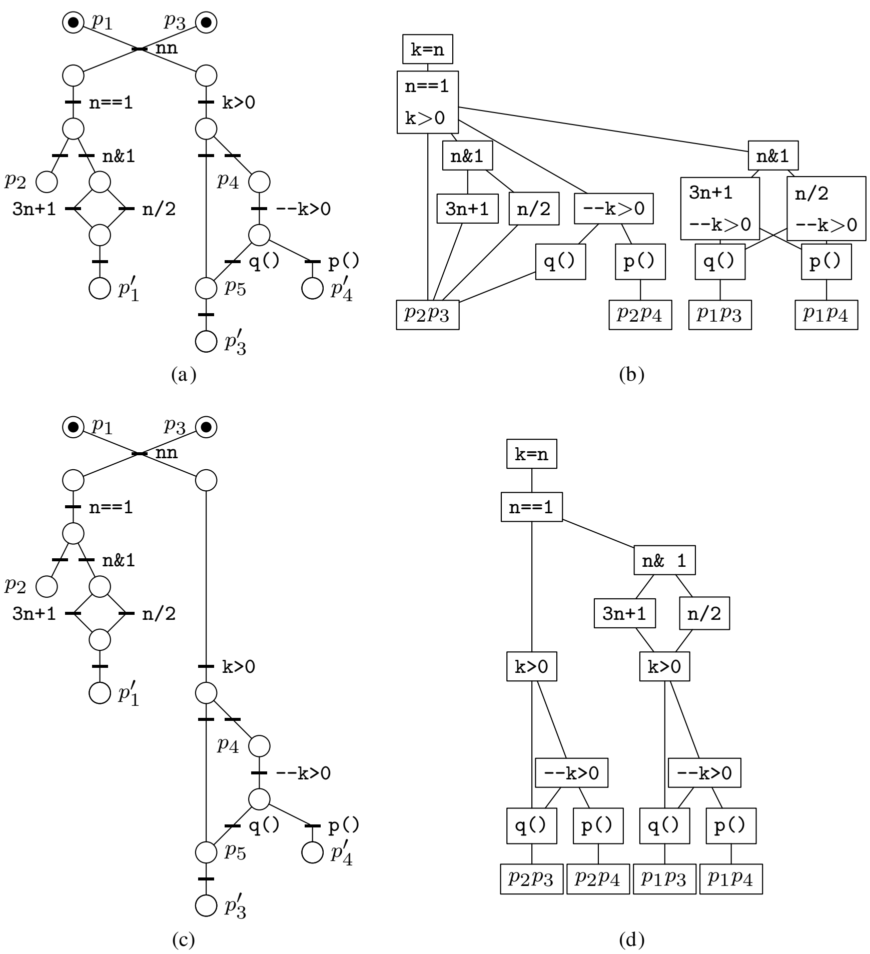


Fig. 26. Synthesizing sequential code from maximal expansions. (a) A scheduled ME. (b) The code generated from it. (c) A different schedule: one which delays the execution of one of the processes. (d) The resulting code.

For example, Figure 26a is an as-soon-as-possible schedule that leads to large, fast code since it fuses decisions, forcing code to be duplicated. By contrast, Figure 26b 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 Section 3.3, uses a different algorithm for sequentializing such graphs that avoids the exponential blow-up at the expense of additional 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 24 and a partial interpreter that builds automata. Since his language is not synchronous like Lustre or Esterel, Lin uses a different criteria—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.

Lin's formalism uses rendezvous communication, which forces communicating processes to synchronize. By contrast, the compiler in the next section uses buffered communication. This makes the scheduling problem harder, however, since it must consider and avoid over- and under-flowing buffers. The main difference is that Lin is able to consider the entire state space of the system, while the FlowC compiler must choose a subset carefully.

5.2 Compiling FlowC

Cortadella et al. [1999] [2000] propose a system for generating code from 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 FlowC, their 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

```
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, In-
Port 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;
    }
  }
}
```

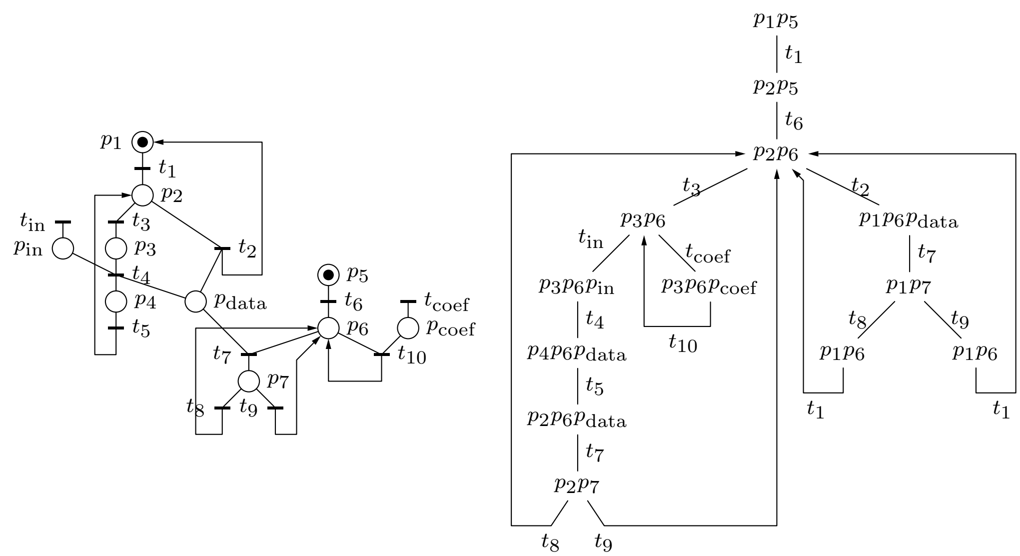
Fig. 27. A pair of communicating processes in FlowC.

of data tokens in inter-process 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 hard as finding an optimized SDF schedule.

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 $p_1 p_5$, both t_1 and t_6 are enabled, but only one needs to be fired since firing the other would only produce a different interleaving. By contrast, at the marking $p_2 p_6$, the left process is making a decision so both t_2 and t_3 need to be fired, producing the two successor states $p_3 p_6$ and $p_1 p_6 p_{\mathrm{data}}$. 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 28a, $\{t_1\}$, $\{t_2, t_3\}$, $\{t_5\}$, and $\{t_8, t_9\}$ are 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



(a) (b)

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

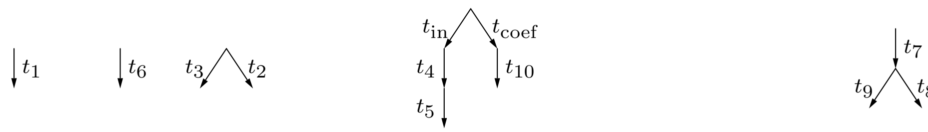
after firing each transition in the first ECS, it considers the transitions in the second ECS and so forth. The algorithm returns "no schedule" if the number of tokens in a buffer place exceeds its bound.

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 since 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. 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., $\{t_{\rm in}\}$, $\{t_{\rm coef}\}$) are considered last since 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 $p_2p_6p_{\rm data}$ and $p_1p_6p_{\rm data}$ 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 since many ECSs always appear in the same sequence (for example, t_7 is always followed by t_8 and t_9 in Figure 28b). Combining ECSs that always run together into a larger




```
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 28b (above) and the fragment of code generated for each (below).

code segments produces faster code by reducing the need for inter-segment run-time scheduling. Their algorithm (described in a tech 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;` in Figure 27), and branching in a code segment corresponds to a conditional test: either an if-then-else or a C switch statement. Figure 29 shows the segments their algorithm generates along with the fragments of code each implies.

Cortadella et al.'s technique, like most in this paper, 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, ten 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.

6. CONCLUSIONS

This paper 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. I classified the techniques 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.

I proposed a simple framework for comparing these different techniques based on the idea of partial evaluation. Each compiler can be thought of 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. With few exceptions, each technique evolved independently and this is the first time they have been discussed together.

Due to space, I have not includes some other noteworthy techniques. For example, the standard subset construction technique used to convert nondeterministic finite automata into deterministic automata (see, for example, 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 evey toward reducing the number of intermediate variables that resembles the same problem in Lustre. Compiling the multi-rate 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 seem 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 *International Workshop on Distributed and Parallel Embedded Systems (DIPES)* (Paderborn University, Germany, Oct. 1998).

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 Transactions on Computer-Aided Design of Integrated Circuits and Systems 18*, 6 (June), 834–849.

BERRY, G. 1992. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A 339*, 87–104.

BERRY, G. 1999. The constructive semantics of pure Esterel. Book in preparation available at http://www.esterel.org.

BERRY, G. 2000. *The Esterel v5 Language Primer*. Centre de Mathématiques Appliquées. Part of the Esterel compiler distribution from http://www.esterel.org.

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

BERRY, G. AND GONTHIER, G. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 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, March 1999).

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* (Veldhoven, The Netherlands, Oct. 1993). pp. 188–196.

BHATTACHARYYA, S. S., LEUPERS, R., AND MARWEDEL, P. 2000. Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing 47*, 9 (Sept.), 849–875.

BHATTACHARYYA, S. S., MURTHY, P. K., AND LEE, E. A. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer, Boston, Massachusetts.

BHATTACHARYYA, S. S., MURTHY, P. K., AND LEE, E. A. 1999. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems 21*, 2 (June), 151–166.

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.

BRYANT, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (Aug.), 677–691.

BRZOZOWSKI, J. A. 1964. Derivates of regular expressions. *Journal of the Association for Computing Machinery 11*, 4 (Oct.), 481–494.

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)* (1985).

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, Jan. 1987).

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, California, June 1995). pp. 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. Technical Report LSI-99-47-R (Nov.), Department of Software, Universitat Politècnica de Catalunya.

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, California, 2000). pp. 489–494.

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

EDWARDS, S. A. 2001. An Esterel compiler for large control-dominated systems. Submitted to IEEE Transactions on Computer Aided Design.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 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, California, June 1995). pp. 151–156.

FUJIMOTO, R. M. 1980. Parallel discrete event simulation. *Communications of the 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], Université d'Orsay.

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

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

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

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (June), 231–274.

HAREL, D. AND NAAMAD, A. 1996. The Statemate semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology 5*, 4 (Oct.), 293–333.

HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey.

HOPCROFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts.

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

IEEE Computer Society. 1996. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (1364-1995)*. 345 East 47th Street, New York, New York: IEEE Computer Society.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Upper Saddle River, New Jersey.

LABROSSE, J. 1998. *MicroC/OS-II*. CMP Books, Lawrence, Kansas.

LE GUERNIC, P., GAUTIER, T., LE BORGNE, M., AND LE MAIRE, C. 1991. Programming real-time applications with SIGNAL. *Proceedings of the 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 Transactions on Computers C-36*, 1 (Jan.), 24–35.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987b. Synchronous data flow. *Proceedings of the IEEE 75*, 9 (Sept.), 1235–1245.

LENGAUER, T. AND TARJAN, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems 1*, 1 (July), 121–141.

LIN, B. 1998. Efficient compilation of process-based concurrent programs without runtime scheduling. In *Proceedings of Design, Automation, and Test in Europe (DATE)* (Paris, France, Feb. 1998). pp. 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 Transactions on Modeling and Computer Simulation 1*, 3 (July), 244–274.

MURATA, T. 1989. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE 77*, 4 (April), 541–580.

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

PLOTKIN, G. D. 1981. A structural approach to operational semantics. Technical Report 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, California, Nov. 1993). pp. 42–47.

SHIPLE, T. R., BERRY, G., AND TOUATI, H. 1996. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Test Conference* (Paris, France, March 1996). pp. 328–333.

SILBERSCHATZ, A. AND GALVIN, P. B. 1998. *Operating System Concepts* (Fifth ed.). Addison-Wesley, Reading, Massachusetts.

SIMONS, B. AND FERRANTE, J. 1993. An efficient algorithm for constructing a control flow graph for parallel code. Technical Report TR–03.465 (Feb.), IBM, Santa Teresa Laboratory, San Jose, California.

STEENSGAARD, B. 1993. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14 (Oct.), Microsoft.

TANENBAUM, A. S. 1992. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, New Jersey.

THIBAULT, S. A., MARLET, R., AND CONSEL, C. 1999. Domain-specific languages: from design to implementation application to video device drivers generation. *IEEE Transactions on Software Engineering 25*, 3 (May), 363–377.

THOMAS, D. E. AND MOORBY, P. R. 1998. *The Verilog Hardware Description Language* (Fourth ed.). Kluwer, Boston, Massachusetts.

ULRICH, E. G. 1965. Time-sequenced logic simulation based on circuit delay and selective tracing of active network paths. In *Proceedings of the 20th ACM National Conference* (1965). pp. 437–448.

ULRICH, E. G. 1969. Exclusive simulation of activity in digital networks. *Communications of the 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, California, Nov. 2000). pp. 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, Texas, Oct. 1999). pp. 646–651.