An Esterel Compiler for a Synchronous/Reactive Development System

Stephen Edwards[1]

June 16, 1994

[0]This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Additional support provided by Interval Research Corporation, Digital Equipment Corporation, and the Semiconductor Research Corporation under grant number 94–DC–008

[1]email: sedwards@alumni.caltech.edu

Abstract

The objective of this project was to create a different scheme for compiling the Esterel synchronous reactive programming language [6, 5, 4, 9] which could handle larger programs, facilitates debugging, and could be easily retargeted toward different architectures. The approach presented here uses an intermediate representation which is somewhere between a high-level reactive language like Esterel and assembly code for a traditional processor. This is similar to the `ic` format used in the Esterel V3 compiler [8] and Baker's NDAM[2, 3]. Compilation proceeds by translating this into assembly code for a SPARC processor.

This differs significantly from the scheme used in the V3 compiler, which derives a single finite-state machine representing the behavior of the program. The FSM approach offers fast executables and exact causality checking, but suffers from exponential growth of compile times and object code sizes.

This report describes the Esterel language, the intermediate representation used by this new compiler, and how the format is translated into executable SPARC assembly code. Its intended audience are those who wish to understand the workings of this Esterel compiler and those simply curious about the Esterel language.

Contents

Chapter 1

Introduction

The synchronous, reactive programming language Esterel was devised by Berry and Cosserat [6] to describe controllers for real-time systems. Esterel resembles many high-level languages, but incorporates a model of time.

This document describes a compiler that translates Esterel into an intermediate form which is an assembly language for an ideal synchronous, reactive machine, and then translates this into SPARC assembly code. This scheme avoids the problems of rapidly-growing object code size and compilation times in the Esterel V3 compiler supported by CISI Ingenierie [8].

Figure 1.1 illustrates where the compiler presented here fits into the synchronous, reactive development environment being developed at the University of California, Berkeley. Baker [2] uses a similar intermediate format to define a synchronous, reactive subset of the VHDL language, which is compiled into Esterel and run with the Esterel V3 compiler. In addition, Baker [3] has also shown that the intermediate representation can be compiled into finite state machines which can then be forwarded to a model checking or language containment verification system for further analysis.

In all development systems, the ability to simulate the system under development is important. Often, this is done exclusively to catch bugs, but it can be used for other purposes. For example, the mock-up of the digital watch presented in Appendix C could be used to evaluate the user interface of the watch. However, since this is a reactive system, the utility of a simulator would drop rapidly if the simulation was too slow.

The path used in [2] (S-VHDL $\rightarrow$ NDAM $\rightarrow$ Esterel) facilitates such simulation, but the Esterel compiler used in that study can have prohibitively long compilation times and large executables. The compiler presented here compiles quickly and produces a fast, small executable.

This document is arranged in roughly the order in which the compiler performs its tasks. Chapter 2 presents the Esterel language in some detail. Chapter 3 contains a description of the representation used as an intermediate be-

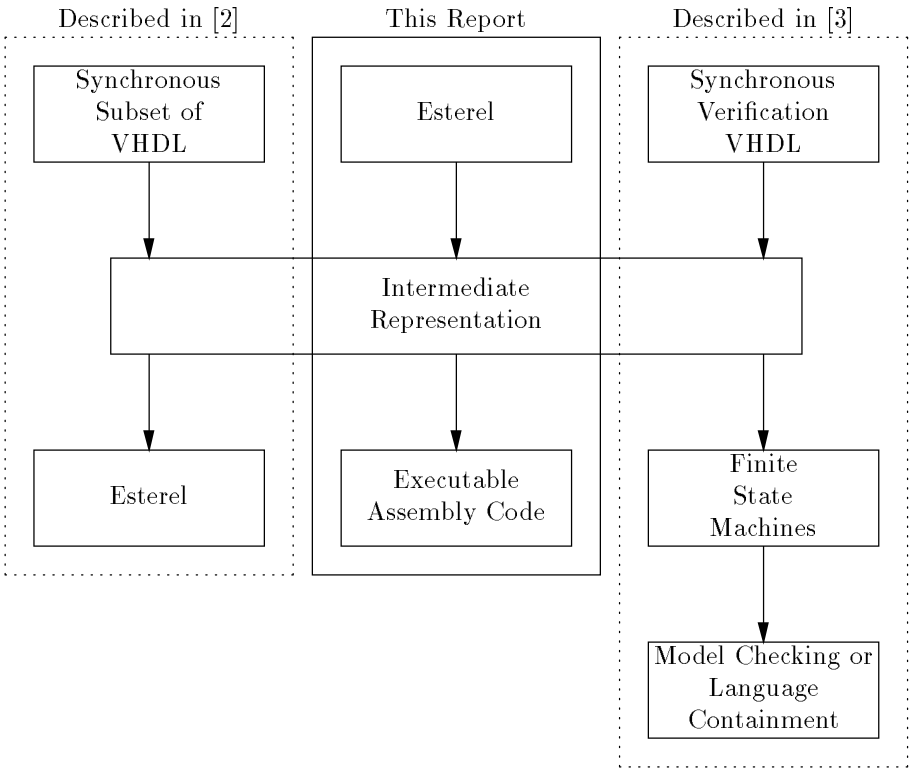


Figure 1.1: Where this compiler fits in the synchronous/reactive environment.

tween Esterel source and assembly code. Chapter 4 discusses the issues in the run-time system. Chapter 5 discusses the important notion of causality in Esterel. Finally, Chapter 6 presents some experimental results and raises some questions about the language. Appendices A and B describe the lexical aspects of Estereland a BNF grammar. Appendix C describes a large Esterel program—a digital watch with five functions.

1.1 Reactive Systems

Reactive systems respond continuously to their environment at a speed determined by their environment. These differ from transformational systems which have all input available at the beginning of execution and produce all output by the end. Between these two extremes are interactive systems, which also respond to their environment continuously, but do so at the system's rate, not the environment's.

The C programming language is well-suited to constructing transformational systems. The event model employed in the X Window System supports interactive systems. Neither of these, however, directly supports the strict time requirements imposed by reactive systems.

Many embedded systems are required to be reactive. For example, an anti-lock braking system in a modern automobile would be of little use if it required anywhere between a second and a minute to detect and react to a wheel skidding. An elevator controller which occasionally ignores a floor request would quickly annoy its users.

1.2 Synchrony

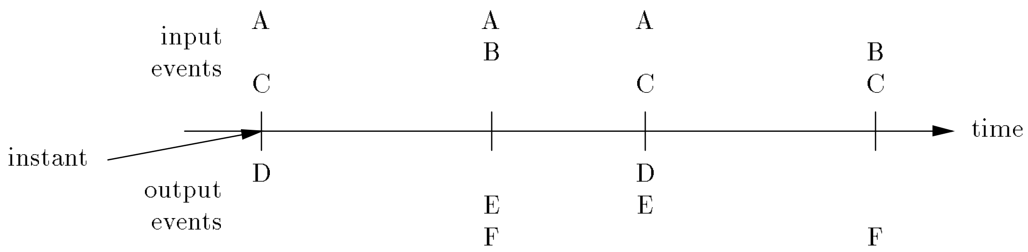
To support reactive programming, Esterel adopts the *strong synchrony model*:[1]

> The program reacts *instantly* to external events. Most instructions take no time, including control structures. Instructions which do take time, such as delay instructions, do so explicitly.

Adopting this model leads to time being treated as a sequence of discrete *instants* between which nothing of interest happens. Events, such as a button being depressed, happen in a particular instant. In the same instant, the program computes and presents its reaction to the event.

This leads to a straightforward notion of concurrency. When two events occur, either they occur in exactly the same instant (are concurrent), or in different instants.

[1] called the *strong synchrony hypothesis* elsewhere




```
module simple:
input A, B, C;
output D, E, F;
  loop
    await A ; emit D ; await A
  end
||
  every B do
    emit F
  end
||
  await B ; emit E ; await C ; emit E
.
```

Figure 1.2: Esterel's model of time.
Presented the input events shown above the time line, the Esterel program shown produces the events listed below the time line.

Figure 1.2 on page 7 illustrates these concepts. An instant is denoted by a vertical tick on the time line. The input events for that instant are listed above this and the events these produce are shown below.

Real machines are not, of course, infinitely fast. But if the machine always has enough time to compute its reaction before the next event arrives, then the perfect synchrony model is satisfied.

1.3 Signals

A signal is a channel on which events occur. For example, a digital stopwatch with a button labeled start/stop might have an input signal called `START/STOP` which has an event each time the button is pressed. It might also employ signals called `SECOND` and `MINUTE` which have events every second and minute respectively. It would be natural to synchronize these so that an event on `MINUTE` occured every sixty seconds. Such synchronization can be specified *exactly* in Esterel:

```
every 60 SECOND do
  emit MINUTE
end
```

Esterel supports two types of signal. A *pure* signal is only ever present or absent. A *valued* signal is either absent, or present with some value. For example, in an elevator controller, there might be a pure signal `DOORCLOSED` indicating that the door has closed, and a valued signal called `FLOOR` indicating on which floor the car has just arrived.

1.4 Previous Work

Berry and Cosserat, the designers of Esterel, write [6]

> The goal of the ESTEREL project is to develop a real-time language based on a *rigorous formal model,* and actually to develop simultaneously the language, its semantics and its implementation.

They give the semantics [6] through a set of rewrite rules. These take an Esterel program, a set of input events, and a memory state to produce a set of output events, a new memory state, and a new Esterel program which does in its first instant what the old program would do in its second instant.

With a program to perform such a rewrite (originally implemented in a LISP-like language), it is a straightforward task to build an interpreter, albeit a slow one.

Shortly after the interpreter was developed, it was discovered that the rules could also produce an Esterel compiler. Because Esterel has no dynamic data allocation (in particular it contains no recursion), every Esterel program can be treated as a finite-state machine. This is made easier when the data portion (separate from the signal portion) is abstracted away.

To compile an Esterel program, an FSM is formed whose states are labeled with complete Esterel programs and whose transitions are labeled with sets of input events. The reset state is labeled with the program to be compiled. The rewrite rules are applied to this program to find the program which results from *every* possible set of input events. Each of these is a potential new state, which is then rewritten with every possible set of input events which may form new states. This process continues until all states, when rewritten with every possible set of input signals, take transitions to other established states. When this process is completed, the state labels can be discarded. At each step, in effect, the *derivative* of the state machine with respect to some input symbols is taken[7].

To keep the number of states within reason, all data-dependent actions are treated separately. When a transition (the execution of a program in an instant) affects memory, perhaps by evaluating an expression, the transition is labeled with that expression. At run-time, the expression is evaluated and the result stored when that transition is taken.

Data-dependent conditional statements (i.e., `if` statements) complicate things. Every time an `if` statement is encountered in a transition, it effectively splits that transition into two branches. At run-time, the `if` condition is tested and the appropriate branch is taken. Each `if` statement can, at worst, double the size of the state machine.

This compilation scheme was used in an earlier Esterel compiler. The latest, the Esterel V3 compiler currently supported by CISI INGENIERIE [8] takes a similar approach: it translates the pure Esterel source into an intermediate representation (called `ic`) which is then used to form an FSM (represented in the `oc` file format) which captures all behavior of the program.

This approach has nice theoretical properties, but has a few shortcomings. In particular, the number of states is potentially exponential in the size of the program, and compilation (i.e., determining the FSM) takes time proportional to the product of the number of states and the number of possible input signal combinations, which is potentially exponential in the number of input signals. For an example of how bad this can be, see Table 6.1 on page 47.

To partially alleviate the explosion in the number of input signal combinations, the keyword `relation` was included, which allows the programmer to reduce the number of possible input combinations by placing constraints on input signals. For example, signals can be marked as mutually exclusive.

The FSM approach allows for the possibility of many instructions being compiled away so that they take *no* time during execution, leading to a program with a constant response time. Manipulation of pure internal signals can be treated in this manner, but data manipulation cannot, making truly constant response time unlikely.

The main advantage of compiling Esterel source into an FSM is that it ensures the program is *causal* and actually makes sense as a specification. It is comparatively easy in Esterel to specify a program which is a paradox, usually of the form “if this happened, then it did not.” If a FSM can be found, then the program is guaranteed, at least, to run. For further discussion of this, see Chapter 5.

The V3 compiler attempts to address the problems of large programs with the `-cascade` option, but this is limited to cases where the program can be broken up into modules which have no feedback, i.e., there exists an ordering of the modules where module i only depends on the actions of modules $0, \ldots, i-1$. However, the correctness of this decomposition is only ensured if the program can be compiled without the option, which can be simply impossible due to memory/disk space constraints.

The other problem with the FSM approach is debugging. Since a particular Esterel instruction does not usually map directly to a specific section of code in the executable, it's difficult to say which instructions were executed. In particular, "single-stepping" is not practical.

1.5 An Alternative

The approach to compiling Esterel presented here is far more traditional. The scheme first mechanically translates the source text into an intermediate representation similar to three-address code used in modern optimizing compilers, then translates the intermediate representation directly into assembly code for a processor. Currently, this compiler produces code for the SPARC environment. This target was chosen mostly out of convenience—the compiler could easily be adapted to another assembly language, or to produce C code.

Calculating the response for each instant is performed as a fixed-point computation, effectively breaking each instant into a series of steps, as shown in Figure 1.3 on page 11. The compiler produces a routine which, when called, takes one of these steps. This is called by a common outer loop which handles signals from the environment and a few housekeeping chores. The issue of convergence is subtle, but in practice the number of iterations required is small (< 20) and fairly constant during the execution of a program.

This compilation approach works well with larger programs. The effort required by the compiler, the size of the executable, and its execution time are all approximately linear in the size of the original Esterel source. This advantage becomes very clear for large programs, where there can be over two orders of magnitude difference in times/sizes compared to the V3 compiler (see Table 6.2 on page 47).

Another advantage to this approach is that it is more easily debugged. Since most instructions in the source program have a direct manifestation in the final executable, stepping through the program and observing the effects of each instruction is feasible.

Finally, a slight variant of the intermediate representation used here has been used to compile a synchronous, reactive subset of VHDL (see [2, 3]). This suggests that a more general synchronous-reactive compiler could be built by adding additional front ends which use the same intermediate representation.

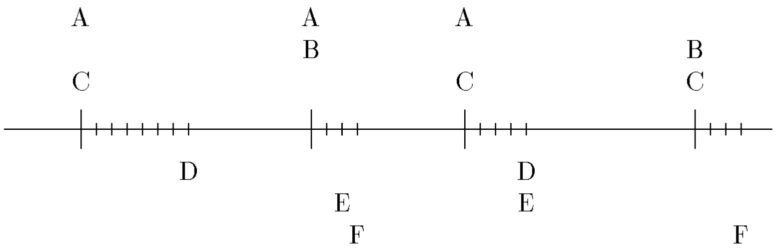


Figure 1.3: Computing the reaction of an Esterel program.
As in Figure 1.2, the input events are shown above the time line. The program computes its reaction in a series of steps following each instant. As long as these steps are completed before the next instant, the perfect synchrony model is satisfied.
A programmer need not and should not consider this—the program will behave as if the signals appear as shown in Figure 1.2.

Chapter 2

The Esterel Language

Esterel is a block-structured textual language with a syntax similar to many high-level languages. It has nested if-then-else statements, loops, and the familiar infix expression syntax. It also has constructs for parallel execution, preemption, and exception handling.

Esterel's semantics are defined by a set of core instructions. The remaining instructions are convenient shorthands for combinations of these. This scheme is attractive because it simplifies formal treatments of the language, yet allows a programmer to write programs whose behavior is much clearer to the human reader.

2.1 Esterel's Model of Time

Esterel's model of time is fundamental to the definition of the language. As described earlier, Esterel invokes the strong synchrony model and assumes the program reacts instantly to stimulus. In this framework, the execution of the program is divided up into discrete instants. In each instant, some set of input events is presented and the program computes a set of output events.

2.2 Signals, Sensors, and Variables

A signal is a broadcast channel for events. Esterel supports two varieties

- Pure signals are either present or absent in an instant, but never both.
- Valued signals are pure signals with an associated value which only changes on event boundaries, but may be read at any time. The `?` operator returns the value of a signal in an expression. For example, `?A` refers to the value of signal `A`.

Sensors are used to represent continuously-varying environmental inputs. Their values are read in the same manner as valued signals, but there is never an event on a sensor, even when the value changes. Sensors may only be inputs from the environment.

Esterel has local variables, but no global variables. These may take boolean or integer values.[1] The value of a variable is set by assignment, and may be tested by an if-then construct.

Signals are used copiously throughout Esterel programs, both for communication with the environment and for internal communication. For most applications, signals are preferred over variables because of their synchronizing ability—and instruction which requires the value of a signal is suspended until the signal has been emitted by another part of the program. Shared variables (written in one part of the program, read in another section executing in parallel in the same instant) are not guaranteed to contain the correct values.

2.3 Occurrences

An *occurrence* describes an instant (the instant in which the occurrence is said to *elapse*) in terms of one or more signal events. Occurrences are used throughout Esterel for synchronization between signals and instructions. For example, the `await` instruction, which simply delays until its occurrence has elapsed, provides a simple form of synchronization.

Occurrences take one of three forms:

- **Simple**

  The occurrence elapses in the instant the given signal has an event, excluding any in the current instant.

- **Immediate**

  The occurrence elapses in the instant the given signal has an event, including any in the current instant.

- **Counted**

  The occurrence elapses in the same instant as the nth event on the given signal, excluding any in the current instant.

The three types of occurrences are depicted in Figure 2.1.

[1] The Esterel V3 compiler supports the importation of variables with more complex types from a host language (such as C), which may be used as arguments to functions from that language. The compiler described herein does not support this.

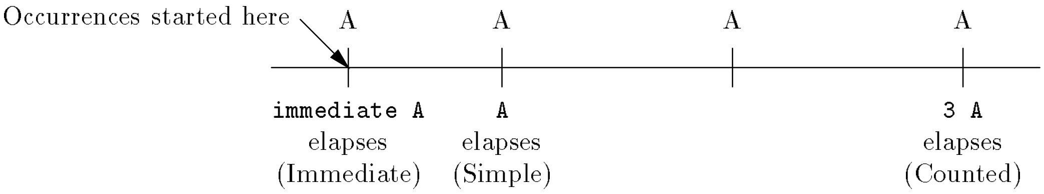


Figure 2.1: Illustration of the three types of occurrences.

2.4 Structure

An Esterel program is broken into a number of modules that execute in parallel. Each module definition contains an interface portion and a body composed of instructions.

Instructions may be composed in sequence or in parallel. Any instruction may be such a composition.

- i_1 `;` i_2 `;` $\cdots$ `;` i_n

  Instructions in a semicolon-delimited sequence are executed in order. First, i_1 is executed. When i_1 terminates, i_2 is executed, and so on. When i_n terminates, the sequence itself terminates.

  These instructions may terminate instantly, leading to an ordering of instructions *within* an instant. However, something like `emit A ; emit B` is equivalent to `emit B ; emit A` since there is no ordering of *events* within an instant.

- i_1 `||` i_2 `||` $\cdots$ `||` i_n

  Instructions delimited by double vertical bars are executed in parallel. i_1 through i_n are executed immediately. When all instructions have terminated, then the parallel construct terminates.

2.5 Core Instructions

2.5.1 Signal

A local signal s is introduced an instruction i by the following construct

```
signal s in
  i
end
```

The signal s can only be used within the instruction i (which may be a composition of instructions), and is not visible outside of the `signal` construct.

2.5.2 Var

A local variable v with type t (`integer` or `boolean`) is introduced into an instruction i with

```
var v :  t in
  i
end
```

An initializing expression e may be included. This is evaluated when the `var` construct is first entered.

```
var v := e :  t in
  i
end
```

Like the `signal` construct, the variable v can only be used within the instruction i, and is not visible elsewhere.

2.5.3 Emit

The `emit` instruction places an event on a signal s in the current instant and terminates instantly.

```
emit s
```

If the signal s is valued, then the `emit` instruction includes an expression which is also evaluated instantly.

```
emit s( e )
```

2.5.4 Halt

The `halt` instruction does nothing and never terminates. This is the fundamental time consumer in Esterel. It appears in many composite instructions, often in situations where it can be preempted.

2.5.5 Preemption and Exceptions

The core preemption construct takes the form

```
do
  i1
watching o
timeout i2 end
```

The instruction i_1 is executed while occurrence o is watched. If i_1 terminates before o elapses, then the whole `do` construct terminates. If o does elapse, i_1 *is terminated before it has a chance to execute for that the instant* and i_2 is executed.

Esterel's exception construct is

```
trap E in
  i1
handle E do i2
end
```

The instruction i_1, which somewhere contains the instruction `exit` E, is executed while exception E is watched. If i_1 terminates without E being raised by the `exit` instruction, then the `trap` construct terminates. If i_1 raises E, then i_1 is terminated and i_2 is executed. However, when E is raised, i_1 *is allowed to finish for the instant.*

Figure 2.2 on page 17 illustrates the differences between `do` and `trap`. It employs the composite instruction `await` o which is a shorthand for `do halt watching` o, which simply waits for the given occurrence to elapse before terminating.

An exception may be given a value that can be read within the instruction i_2 with the `??` operator. For example, `??F` refers to the value of exception `F`, given by the expression e in `exit F(`e`)`.

2.5.6 Conditionals

The `if` statement in Esterel has the familiar form

```
if e then i1
else i2
end
```

The boolean expression e is evaluated instantly. If true, then instruction i_1 is executed, otherwise, i_2 is executed.

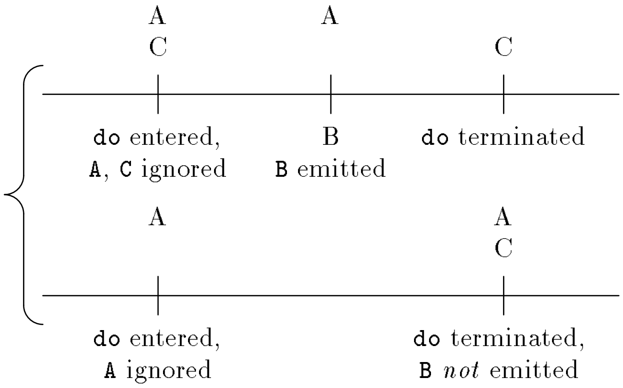
The other conditional statement in Esterel checks for the presence of a signal.

```
present s then i1
else i2
end
```

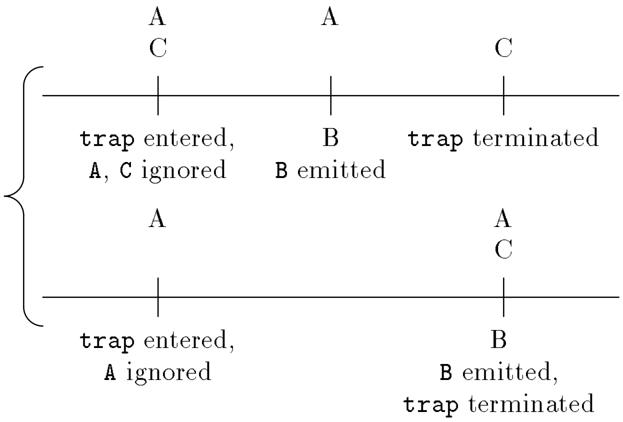
Here, instruction i_1 is executed if signal s is present in the current instant. Otherwise, i_2 is executed.

For both conditional statements, either the `then` i_1 or the `else` i_2 clause may be omitted.

```
do
 await A ; emit B ; halt
watching C
```




```
trap E in
 await A ; emit B ; halt
||
 await C ; exit E
end
```




```
do
 await immediate A ; emit B ; halt
watching immediate C
```

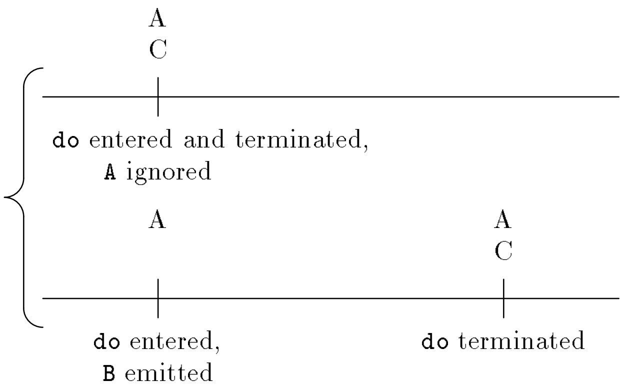


Figure 2.2: An illustration of the differences between `do` and `trap` and the effect of the `immediate` keyword.
On the left are the responses of the fragments on the right to two different sequences of input events.

2.5.7 Loop

Esterel includes an infinite loop instruction:

$$\texttt{loop}\quad i\quad \texttt{end}$$

This executes the instruction i, waits for it to finish, and executes it again. True infinite loops are often desired in controllers, but when necessary, these can be terminated through preemption or an exception. It is an error for the instruction i to take no time. For example, `loop emit S end`. This corresponds to the program doing an infinite amount of work in zero time.

2.6 Composite Instructions

2.6.1 await

The `await` instruction is one of the most common. It waits for its occurrence to elapse and then terminates.

`await` o	translates to	`do`
		`  halt`
		`watching` o

`await` also comes in the following elaborate form:

```
await
 case o1 do i1
 case o2 do i2
 ⋮
 case on do in
end
```

This waits for one of the occurrences $o_1, \ldots, o_n$ to elapse, say o_j. Then, instruction i_j is executed and the `await` terminates. If two occurrences elapse in the same instant, the one *listed first* takes precedence.

This can be built using preemption and exceptions:

`await`
` case` o_1 `do` i_1
` case` o_2 `do` i_2
`end`
	translates to	`trap E1 in`
		`  trap E2 in`
		`    do halt watching` o_1
		`    timeout exit E1 end`
		`  ||`
		`    do halt watching` o_2
		`    timeout exit E2 end`
		`  handle E2 do` i_2 `end`
		`handle E1 do` i_1 `end`

2.6.2 do ... upto

`do`
  i
`upto` o
	translates to	`do`
		  i `; halt`
		`watching` o

In effect, this forces the duration of instruction i to be exactly the length of occurrence o, terminating i early if it does not finish before o elapses.

2.6.3 loop ... each

This modification of the `loop` instruction restarts itself whenever its occurrence elapses:

`loop`
  i
`each` o
	translates to	`loop`
		 `do`
		  i `; halt`
		 `watching` o
		`end`

2.6.4 every ... do

This repeatedly synchronizes its instruction with its occurrence:

`every` o `do`
 i
`end`
	translates to	`await` o
		`loop`
		 `do`
		  i `; halt`
		 `watching` o
		`end`

2.6.5 sustain

Although signals are often thought of as events, the `sustain` instruction allows them to be used as flags. It uses a special signal, `TICK`, which is present in every instant by definition. Thus, `sustain` forces another signal to be present in every instant.

Typically, `sustain` is used with a `do` or a `trap` that defines the length of time that it is executing, and hence, the length of time that its flag is asserted.

`sustain` s	translates to	`every TICK do`
		 `emit` s
		`end`

2.7 A Stopwatch Controller in Esterel

Figure 2.3 on page 22 depicts a simple stopwatch controller illustrating many common characteristics of Esterel programs. The module consists of four (elaborate) instructions running in parallel, each responsible for some part of the stopwatch. All communication, both with the outside world and between different parts of the program, is done through signals. Preemption and exceptions are used liberally.

2.7.1 Signals

A typical digital stopwatch has two buttons marked start/stop and lap/reset, here conveyed through the `SS` and `LR` input signals respectively. `SECOND` is a periodic signal assumed to be generated by an external oscillator once a second.

When in lap mode, the stopwatch continues to measure time, but the display does not change. The `FROZEN` output is present when the display is in this mode. The integer-valued output `TIME` is the value for the display, and does not change when the stopwatch is lap mode.

The internal signal `RESET` resets the counter. `LAP` indicates a switch between lap and normal mode. `RUN` is used as a flag to indicate that the stopwatch is running, and is present in every such instant.

2.7.2 The Start/Stop Button Handler

The first process is responsible for the action of the start/stop button, which is a simple toggle. When the stopwatch is running, the `RUN` signal present in every instant, which is enforced by the `sustain` instruction.

This illustrates how simply state information can be incorporated into Esterel code. Here, there are two states: one which waits for `SS`, and one which sustains the `RUN` signal until the next `SS` signal.

2.7.3 The Lap/Reset Button Handler

The second process is responsible for decoding the action of the lap/reset button. When lap/reset is pressed and the the stopwatch is running, the stopwatch switches into or out of lap mode. If the stopwatch is not running and not in lap mode, then the action is to reset. The `every` construct ensures that these actions are taken exactly when the lap/reset button is pressed.

This behavior could be described by the following boolean equations

$$\begin{aligned} \mathtt{LAP} &= \mathtt{B2} \cdot \mathtt{RUN} \\ \mathtt{RESET} &= \mathtt{B2} \cdot \overline{\mathtt{RUN}} \cdot \overline{\mathtt{FROZEN}} \end{aligned}$$

Esterel's representation is more flexible than this. Currently this process contains no state information, but in Esterel, it can be added without a significant change to the code. The precedence of the operations is made explicit by the familiar form of the `present` instruction.

2.7.4 The Frozen Display Handler

The third process keeps track of the lap mode and generates the `FROZEN` signal accordingly. Similar to the first process, it implements a toggle which sustains the `FROZEN` signal after an odd number of `LAP` events.

This uses the `trap` construct to ensure that `FROZEN` is sustained up to and including the instant `LAP` appears. This is in contrast with the first process which does not emit `RUN` when the start/stop button is pressed the second time.

2.7.5 The Counter

The fourth process, the counter, illustrates the use of a local variable and how the explicit initialization feature can be used. By enclosing the `var` declaration and its initialization expression in a `loop`...`upto` preemption construct, the reset behavior is automatic and straightforward. The way to think about it is this: A counter is something that starts at zero and goes up every second that the stopwatch is running. If the display is not frozen, then this count should be broadcast to the display. After every reset, this process is restarted.

```
module STOPWATCH:

input SS, LR, SECOND;
output TIME(integer);

signal RESET, LAP, RUN, FROZEN in

% Start/Stop Button Handler

  loop
    await SS;
    do
      sustain RUN
    upto SS
  end

||

% Lap/Reset Button Handler

  every LR do
    present RUN then emit LAP
      else present FROZEN
        else emit RESET
      end
    end
  end

||

% Frozen Display Handler

  loop
    await LAP ;
    trap T in
      sustain FROZEN
    ||
      await LAP ; exit T
    end
  end

||

% Counter

  loop
    var second := 0 : integer in
      emit TIME(second) ;
      every SECOND do
        present RUN then
          second := second + 1
        end ;
        present FROZEN else
          emit TIME(second)
        end
      end
    end
  upto RESET

end
.
```

Figure 2.3: A stopwatch controller written in Esterel.

Chapter 3

The Intermediate Representation

Like many compilers, this compiler translates source code into an intermediate representation before generating assembly language for the target processor. The intermediate representation used here was chosen using the following criteria:

- Completeness—Every construct in Esterel must have a correct translation into the intermediate representation. In particular, it must support the parallel execution and preemption semantics of Esterel.

- Generality—A similar representation has been used in a compilation scheme for a synchronous subset of VHDL [2, 3]. Mimicking this work ensures that this intermediate representation could be used in other situations, allowing the reuse of the code generator.

- Simplicity—Keeping the intermediate representation as simple as possible simplifies the final code generation phase and makes optimization much easier.

The result of balancing these sometimes conflicting requirements is presented below. The intermediate representation chosen is very close to the three-address code used in modern optimizing compilers [1]. All but one of the instructions are "simple" in some sense—they translate into only a few assembly-language instructions. The remaining instruction, `try`, is responsible for parallel execution and preemption and is really the workhorse of the language.

The notion of time employed in the intermediate representation is the same synchronous/reactive one used by Esterel. Only the `halt` instruction takes any time—the rest happen instantly, but in an order.

The following presentation introduces an informal syntax which corresponds directly with the data structures used inside the compiler[1]. This syntax is intended to be illustrative rather than machine-readable, although the compiler can produce it.

3.1 Data Objects

Unlike a typical assembly language, the intermediate representation manipulates objects at a higher level than memory and registers. In particular, valued and pure signals identical to Esterel's are included, as well as valued and pure exceptions.

All objects in the intermediate representation are globally accessible—the correct scoping is imposed by the structure of the Esterel program.

3.1.1 Signals

`s0`, `s1`, `s2`, ...

The intermediate representation deals with both pure and valued (with an integer value) signals. At any time during the execution of the program, each signal is in one of the following three states:

- **present**: The signal is present in the current instant.
- **absent**: The signal is absent in the current instant.
- **unknown**: The presence or absence of the signal in the current instant is unknown.

All the signals used in the program are listed in the `Signals` block. Each is listed along with its name from the Esterel program, and its type. For example, `s1: RING` specifies that pure signal `s1` represents the signal called `RING` in the Esterel source program. `s5: A(int)` indicates that signal `A` in the Esterel source has been assigned to signal `s5`.

3.1.2 Variables

`v0`, `v1`, `v2`, ...

Variables take integer values. Esterel's booleans are implemented using the integer values 1 and 0 for true and false respectively. Variable are listed in the `Local Variables` block along with their names from the Esterel program. No type is specified—all are integers.

[1]Processes are classes with variable-sized arrays of instruction objects, an integer denoting the process's number, and an integer denoting its program counter. Each instruction is a subclass of a general instruction class. For example, the `try` class contains an array of pointers to the subprocesses it calls, an array of watch clauses, and an array of handle clauses. Information about data objects is also stored in arrays.

3.1.3 Registers

`r0`, `r1`, `r2`, ...
Like variables, these take integer values, but their values are only guaranteed to persist until the next non-assignment statement. Registers are used primarily to store intermediate results in evaluating expressions.

3.1.4 Exceptions

`e0`, `e1`, `e2`, ...
The intermediate representation contains both pure and valued exceptions. At any time during execution, each exception is either

- **raised**: An `exit` statement has raised the exception
- **lowered**: The exception is being observed, but no corresponding `exit` has been executed.

Like signals, each exception is listed in the `Exceptions` block along with an `(int)` designation when the exception is valued, and its name from the Esterel program.

3.1.5 Counters

`c0`, `c1`, `c2`, ...
Counters are used in the intermediate representation to keep track of the number of events that have been observed on a particular signal in a counted occurrence. Like signals, each counter is listed along with its name (the name of the signal being counted in that occurrence) in the `Counters` block.

3.2 Processes

The program of the intermediate format is build from a hierarchically-arranged group of *processes*—sequences of instructions. A single process behaves like a program on a standard processor—each has a program counter pointing to the instruction currently being executed. Once that instruction has been executed, the program counter is moved to the next instruction to be executed (usually the next in the sequence, but branches are allowed) and execution continues. Each instruction in a process is assigned a small integer label used for branch targets.

Execution of a process may not "fall off the end." The last instruction of a process is a `halt`, which makes the process's execution cease but does not terminate it, an `exit` which terminates the process, or an unconditional `goto`.

Each process is introduced with a line giving its unique name (`P0`, `P1`, etc.) and which program counter it uses, e.g., `PC1`, `PC5`, etc. Each instruction in a process is given an index. (`0:`, `1:`, etc.) These labels are used as branch destinations and simplifies correlating assembly language instructions with those in the intermediate format.

3.3 Simple Instructions

3.3.1 Assignment Statments

The intermediate representation has one all-encompassing assignment statement which handles most data manipulation. It may have two or three arguments, each of which may be one of the data objects described above or a constant integer. Many combinations of these are unused—for example, the destination of an assignment may not be constant, and only the simple assignment form is allowed to have non-register operands. These policies of use were imposed to simplify the translation of these instructions into assembly code for the SPARC RISC processor, and would simplify the translation for other processors.

- d `:=` s

  Simple assignment. The value of the source is written into the destination.

- d `:=` $op\ s$

  Unary operation. The operator is applied to the value of the source and the result written into the destination. The unary operators are integer negate, binary NOT, and decrement.

- d `:=` $s_1\ op\ s_2$

  Binary operation. The operator is applied to the values of the two sources and the result written into the destination. The binary operators are integer add, subtract, multiply, divide, and modulus, binary AND and OR, integer equality, integer less than, and integer less than or equal to.

When a signal or exception is referenced, the value returned is the value of the signal in the current instant, and not presence/absence or raised/lowered information. For such accesses to be legal, the presence/absence of a signal must be established in an instant or the raised status of an exception must be known, either from context (The code run by a `handle` clause of a `try` obviously knows that the exception has been raised.) or through force (The `require` statement ensures that the value of a signal is known correctly.).

3.3.2 Flow-of-Control Statements

The intermediate representation has a general branch instruction which either unconditionally branches to an instruction, checks for the zero/non-zero status of a register or variable, the presence/absence of a signal, or the raised/lowered status of an exception. It takes the following forms

`goto` l	Unconditional branch to instruction l
`if` s `goto` l	If status of s is "true," branch to instruction l
`if not` s `goto` l	If status of s is "false," branch to instruction l

3.3.3 `emit`

The `emit` statement is very similar to its Esterel counterpart. It makes the given signal present in the current instant, and may set its value, if any. The two forms are

`emit` s	Emit the signal s
`emit` s r	Emit the signal s setting its value to register r

3.3.4 `exit`

The `exit` statement, like its Esterel counterpart, can raise an exception and set its value. In the intermediate representation, it also serve to terminate a process (In Esterel, this action was implicit.) The three forms are

`exit`	Terminate the process
`exit` e	Terminate the process and raise exception e
`exit` e r	Terminate the process and raise exception e, assigning it the value in register r.

3.3.5 `halt`

The `halt` statment, like its Esterel counterpart, prevents further execution of the process, but does not allow it to terminate. This is the only mechanism in the intermediate representation which consumes time. Between instants, each active process is either stopped at a `halt` or waiting on a halted subprocess.

3.3.6 `require`

`require` s_1 s_2 $\ldots s_k$

When the value of a signal is needed to evaluate an expression, the newest value of that signal is needed. Similarly, a conditional branch which depends on the presence or absence of a signal must know whether the signal is present or absent in an instant before proceeding. The `require` instruction ensures that the named signals are known before execution may proceed.

3.4 The `try` Instruction

The `try` instruction is responsible for parallel execution of processes and both kinds of preemption (signal- and exception-prompted). Its most general form is

$$
\begin{array}{l}
\texttt{try} \\
\quad \texttt{call}\ \ p_1 \\
\quad \texttt{call}\ \ p_2 \\
\quad \vdots \\
\quad \texttt{call}\ \ p_n \\
\texttt{watching}\ s_1\ c_1\ \texttt{goto}\ wl_1 \\
\texttt{watching}\ s_2\ c_2\ \texttt{goto}\ wl_2 \\
\vdots \\
\texttt{watching}\ s_m\ c_m\ \texttt{goto}\ wl_m \\
\texttt{handle}\ e_1\ \texttt{goto}\ el_1 \\
\texttt{handle}\ e_2\ \texttt{goto}\ el_2 \\
\vdots \\
\texttt{handle}\ e_p\ \texttt{goto}\ el_p
\end{array}
$$

This calls subprocesses p_1 through p_n while monitoring signals s_1 through s_m and exceptions e_1 through e_p. If one of these signals is present or an exception is raised, then the subprocesses are terminated and the handler pointed to by the `goto` is executed, otherwise the instruction terminates when all of its subprocesses have.

More specifically, if exception e_i is raised and exceptions e_1 through e_{i-1} are not, then the subprocesses are terminated *after they have completed for the instant* and execution proceeds with instruction el_i. If signal s_i is present, then counter c_i is decremented. If counter c_i becomes zero and counters c_1 through c_{i-1} are non-zero, then the subprocesses are terminated and execution proceeds with instruction wl_i. The signals are checked before any subprocesses execute in every instant *except the first in which the* `try` *is executed.*

The semantics of this instruction were chosen to capture Esterel's most difficult instruction, `await...case`. This can have a mixture of counted, uncounted, and immediate occurrences. Counters were introduced to handle the counted occurrences. Another approach would have been to introduce a separate process to count the signals. However, since the semantics require that when a preempting signal occurs, *none* of the processes will be executed in that instant, it was not clear how to do this.

In a previous version of this compiler, there were two varieties of the `try` instruction, one that dealt only with signals, and one that dealt only with exceptions. All instructions in Esterel can be represented by a nesting of such constructs, but since much of the code was the same (program counter initialization, subprocess calling, potential set calculation, and so forth), the two varieties were merged.

3.5 Translating Esterel

The translation of a small Esterel program into the intermediate format is shown in Figure 3.1 on page 30. The `await A` instruction turns into a `try` which calls a process containing a single `halt` while watching the awaited signal. The `present` instruction, since it needs to know about a signal to correctly check its condition, is composed of a `require` statement followed by a conditional branch.

In this simple example, there are three expressions: "`3`" in the initialization of variable `C`, "`C + 3`" in the first `emit`, and "`C - 2`" in the second. These expressions produce code in a RISC-like load-store manner. Values which reside in "memory" (i.e., non-registers) are loaded into registers before being manipulated.

A more elaborate program and its translation are shown in Figure 3.2 on page 31. This example contains an `await...case` statement, which is the most complex in Esterel, employing the three types of occurrences (simple, counted, and immediate). The counted occurrence requires a counter to be loaded with the iteration count before the `try` instruction is entered (instructions 2 and 3 of process `P0`). The immediate occurrence is similar to a simple occurrence, but the presence of the signal is checked before the `try` instruction is entered (instructions 0 and 1 of process `P0`).

This example also illustrates the use of an exception. The statements which are observed for the exception are placed in a separate process (`P3`) and run by a `try` (instruction 14 of process `P0`) with a `handle` clause.

The value of signal `B` is read by the `?` operator in the `exit` instruction. The `exit` instruction (instruction 5 of process `P3`) raises the exception and sets its value to the contents of register `r0` which contains the value of signal `s2`. Note that this signal was required (in instruction 3) before the expression was evaluated. The value of the exception is loaded in the handler routine and emitted through signal `F` (instructions 12 and 13 of process `P0`).

```
module example1:
input A,D;
output B(integer);

  loop
    var C := 3 :  integer in
      await A ;
      present D then
        emit B(C+3)
      else
        emit B(C-2)
      end
    end
  end
.
```

```
Local variables
v0:  C

Signals 4( 1 valued )
s0:  TICK
s1:  A
s2:  D
s3:  B(int)

Process P0 uses PC0

0:  r0 := 3
1:  v0 := r0

2:  try
      call P1
    watching s1 goto 3

3:  require s2
4:  if not s2 goto 10

5:  r0 := v0
6:  r1 := 3
7:  r0 := r0 + r1
8:  emit s3 r0
9:  goto 14

10:  r0 := v0
11:  r1 := 2
12:  r0 := r0 - r1
13:  emit s3 r0

14:  goto 0

Process P1 of process P0 uses PC1
0:  halt
```

Figure 3.1: A small example and its translation into the intermediate format. The local variables and signals are listed above the processes. Each instruction in each process is labeled with a small integer, and each process is introduced with a name and which program counter it uses.

```
module example2:
input A, B(integer), C;
output D, E, F(integer);

  await
    case A do emit D
    case 3 B do
      emit E ;
      await A ;
      emit D
    case immediate C do
      emit F(0)
  end
  ;
  trap e(integer) in
    loop
      await A ;
      present B then
        exit e(?B)
      end
    end
  handle e do
    emit F(??e)
  end
.
```

```
Signals 7 ( 2 valued )
s0:  TICK
s1:  A s2:  B(int) s3:  C
s4:  D s5:  E s6:  F(int)

Counters 1
c0:  B

Exceptions 1 ( 1 valued )
e0:  e(int)

Process P0 uses PC0
0:  require s3
1:  if s3 goto 12
2:  r0 := 3
3:  c0 := r0
4:  try
      call P1
    watching s1 goto 6
    watching s2 c0 goto 8
    watching s3 goto 12
5:  goto 14

6:  emit s4
7:  goto 14

8:  emit s5
9:  try
      call P2
    watching s1 goto 10
10:  emit s4
11:  goto 14

12:  r0 := 0
13:  emit s6 r0

14:  try
      call P3
    handle e0 goto 16
15:  goto 18

16:  r0 := e0
17:  emit s6 r0
18:  exit

Process P1 of process P0 uses PC1
0:  halt

Process P2 of process P0 uses PC1
0:  halt

Process P3 of process P0 uses PC1
0:  try
      call P4
    watching s1 goto 1

1:  require s2
2:  if not s2 goto 7

3:  require s2
4:  r0 := s2
5:  exit e0 r0
6:  goto 0

Process P4 of process P3 uses PC2
0:  halt
```


Figure 3.2: A further example illustrating the translation from Esterel into the intermediate format.

Chapter 4

Execution

The intermediate format has been chosen so that virtually all instructions are easy to execute on a traditional processor. This compiler generates code for the SPARC architecture, although it could easily be retargeted to another. Arithmetic instructions translate, for the most part, into single assembly language instructions. `emit` simply stores a value in an array in memory, and a conditional branch consists of a test followed by an assembly-language branch.

The difficult part of executing the intermediate format (and hence, Esterel) is ensuring that the correct instructions execute in each instant in the right order. For example, the subprocesses of a `try` must not be executed if any of the watched signals are present in that instant. Also, if an expression reads the value of a signal, and if some other part of the program is going to emit that signal, the expression should get the new value, and not the old.

4.1 Causal Interleaving

The approach taken here turns the execution of the intermediate format into a fixed-point computation on the set of signals, both internal and external. The code generator produces a routine, hereafter referred to as the main process routine, that takes incomplete information about the presence or absence of signals and emits what signals it can, changing program counters, variables, counters, exceptions, and so forth in the process. Execution for each instant consists setting all signals except those global inputs that are known to be present or absent to unknown, and calling the routine repeatedly until the presence or absence of each signal is known.

There is no instruction in the intermediate representation that "unemits" a signal, so something else must establish the absence of a signal. A conservative rule is used to decide which signals are absent:

If an internal or output signal could not be emitted in the current instant, then is marked as absent.

Much of the code produced is responsible for establishing which signals could still be emitted in the current instant. In particular, when the main process has executed what it can using the information it has, it returns the address of a routine that marks all the signals that still have the potential to be emitted in the current instant. The outer loop then marks all unknown signals which are *not* in this potential set as absent.

This technique causes the set of present and absent signals to grow monotonically and the set of unknown signals to shrink monotonically since once a signal is marked as present, it cannot be marked as absent (only unknown signals are marked as absent), and once it is marked as absent, it could not thereafter be emitted since all signals that could possibly be emitted are placed in the potential set.

4.2 Process Routines

Each process in the the intermediate representation is translated into a sequence of assembly-language instructions, which compose its process routine.

Each process routine, when called, executes as many instructions as it can, depending on the state of various signals, and returns with

- A program counter that points to the instruction to be executed the next time the process routine is called;
- A potential set calculator that points to a routine that marks those signals which could still be emitted in the current instant; and
- A status, one of

Terminated (0) The process has hit an `exit` instruction and has terminated. In this case, the returned program counter points to this same `exit` instruction and the returned potential set calculator points to a null routine (this process could not emit any more signals this instant).

Halted (1) The process has hit a `halt` instruction. In this case, the returned program counter points to this same `halt` and the returned potential set calculator points to a null routine.

Waiting (−1) This indicates that the process needs information about more signals before it may proceed, but that it has more to do in the current instant. The returned potential set calculator points to a routine that marks the appropriate signals.

If the process exits or halts, the returned program counter points to the same `exit` or `halt` instruction. This simplifies the code that calls the process: rather than having to test a flag or the program counter against something that indicates the terminated or halted status of the process, it can always call the process regardless of whether it has terminated.

When a process routine contains a `try` instruction, it calls other process routines (i.e., those in `call` clauses of the `try` instruction). Potential set calculators also may call other potential set calculators with the relationships imposed by the `try` instruction.

Each process is assigned a unique program counter. In any program, it may not be possible for every process to be active simultaneously, so each program counter may not have a unique process. In particular, if a process contains two `try` instructions, the processes called by those `try` instructions may share program counters. The assignment of program counters is performed at compile time with a simple recursive rule which makes a worst-case estimate of the number of simultaneously-active processes.

4.3 Processor Registers

Processor registers are used to return the new program counter, the new potential set calculation routine, and the return status.

Intermediate representation registers are mapped directly onto processor registers.

One processor register stores the base address of all the arrays for addressing purposes.

Register use is shown in Table 4.2.

4.4 Simple Instructions

The compiler currently produces assembly code for the SPARC architecture. This is a modern RISC processor with 32 32-bit registers. Register `%g0` is special—it always returns a zero, and may be used as the destination for a result which is ignored.

The branch instructions on the SPARC have a single shadowed instruction following. This instruction is not executed only if the annul flag (part of the branch opcode) is set and the branch fails.

The one addressing mode used by the compiler adds a 12-bit immediate value to a register to form a full 32-bit address. By putting the high-order bits of the base address of the arrays in a register, this mode facilitates quick access to all the run-time data. The syntax

```
[%i5+%lo(_V+12)]
```

Description	C definition	Multiplicity
Program Counters point to process routines. These are function pointers.	`void (*PC[])()`	one per active process
Potential Function Pointers point to potential set calculation routines. These are function pointers.	`void (*PO[])()`	one per active process
Halted Flags A process sets its halted flag to 1 when it is finished for the instant. At the beginning of each instant, these flags are all cleared to 0 to restart the processes for the instant. These flags are used by both the process routines and the potential set calculators.	`char H[]`	one per active process
Signal Presence Flags Each signal is either present (1), absent (-1), or unknown (0). At the beginning of each instant, all but the input signals are set to unknown.	`char S[]`	one per signal
Signal Potential Flags The potential set routines set each signal with the potential to be emitted to 1. Those signals which have no potential (0), and are unknown are marked as absent.	`char P[]`	one per signal
Signal Values	`int SV[]`	one per valued signal
Variable Values	`int V[]`	one per variable
Counter Values	`int C[]`	one per counted occurrence
Exception Presence Flags	`char E[]`	one per exception
Exception Values	`int EV[]`	one per valued exception

Table 4.1: List of all runtime data structures.
The size of each of these is determined at compile time.

name	use
%g0	Always zero
%o0	Temporary
%o1	Temporary
%l0	r0
%l1	r1
⋮	⋮
%l7	r7
%i2	Returned program counter
%i3	Returned potential set calculation routine
%i4	Returned status (−1,0,1)
%i5	Base address of arrays

Table 4.2: Register usage for the SPARC processor.

means "take the low-order bits of the address of _V, an array, add twelve, and add this to the register %i5". %i5 contains the base address of the arrays, so this refers to the 12th byte of the _V array. Such a scheme allows for a small, limited amount of data, but this has not presented a problem thus far.

4.4.1 Assignment Statements

Simple assignment instructions translate into single instructions:

- A memory load instruction (array to register). e.g.,

  `r0 := v1` → `ld [%i5+%lo(_V+4)],%l0`

- A memory store instruction (register to array)

  `v3 := r1` → `st %l1,[%i5+%lo(_V+12)]`

- A constant load (constant to register)

  `r2 := 5` → `mov 5,%l2`

Assignment instructions with unary and binary operands only reference registers, so most translate to single instructions:

`r0 := not r1` → `xnor %g0,%l1,%l0`

`r1 := r2 + r3` → `add %l2,%l3,%l1`

The comparison operators use the SPARC's annul flag, which cancels the execution of the instruction in the branch delay slot if the branch is not taken. If the branch is taken, 1 is loaded, otherwise 0 is loaded.

```
r1 := r2 < r3          →    cmp %l2,%l3
                            bl,a LL1
                            mov 1,%r1
                            mov 0,%r1
                           LL1:
```

4.4.2 Flow-of-Control Statements

Each instruction is given a label like `P5I3`, which refers to instruction 3 of process 5.

- Unconditional branches translate directly.

```
goto 1              →    ba P0I1
                         nop
```

- A register test translates to

```
if r1 goto 5        →    tst %l1
                         bne P3I5
                         nop
```

- A signal presence test translates to

```
if not s1 goto 3    →    ldsb [%i5+%lo(_S+1)],%o0
                         tst %o0
                         bneg P2I3
                         nop
```

4.4.3 `emit`

`emit` simply stores the signal value, if any, and sets the signal to present:

```
emit s2 r1          →    st %l1,[%i5+%lo(_SV+8)]  ! store value
                         mov 1,%o0
                         stb %o0,[%i5+%lo(_S+2)]  ! set to present
```

4.4.4 `exit`

An `exit` with an exception stores the exception value, if any, raises its exception, sets the return PC to branch to the same `exit` instruction, and sets the return potential set routine to null. It then "returns" to the `try` instruction which called it, returning the halted status.

Returning the halted status in this case is done to get around a technical point. If all other processes were also to terminate, the innermost enclosing `try` would execute first. Since this `try` may *not* be watching for the given exception (i.e., another further out would be), it may erroneously execute further instructions. The halted status prevents this possibility and since we can be assured that some enclosing `try` is watching for the exception, it will be handled.

```
exit e1 r0        →   st %l0,[%i5+%lo(_EV+4)] ! store value
                      mov 1,%o0 ! set to raised
                      stb %o0,[%i5+%lo(_E+1)]
                     LL1:
                      set LL1, %i2 ! Return PC
                      set PO1, %i3 ! Return PO
                      ba PR1
                      or %i4, 1 , %i4 ! return HALTED
```

When the `exit` does not refer to an exception, the generated code is similar, except that the terminated status is returned (implicitly) instead of the halted status.

```
exit              →  LL1:
                      set LL1, %i2 ! Return PC
                      set PO1, %i3 ! Return PO
                      ba PR1
                      nop
```

4.4.5 `halt`

`halt` is much like an `exit`. Naturally, the halted status is returned.

```
halt              →  LL2:
                      set LL2, %i2 ! Return PC
                      set PO2, %i3 ! Return PO
                      ba PR2
                      or %i4, 1, %i4 ! return HALTED
```

4.4.6 Require

`require` may force a process to return with the waiting status. First, it checks its signals and if none are unknown, it branches to the next instruction. Otherwise, it returns with the waiting status and returns a potential set calculation routine that marks those signals can be emitted.

```
require s2 s3   →  ! check signals s2 and s3
emit s0            POI1:
emit s1             ldsb [%i5+%lo(_S+2)],%o0
                    tst %o0
                    be LL0
                    ldsb [%i5+%lo(_S+3)],%o0
                    tst %o0
                    be POI2
                    nop
                   ! Return waiting
                   LL0:
                    set POI1,%i2 ! PC
                    set LL1,%i3 ! potential
                    ba PR0
                    or %i4,-1,%i4 ! return waiting
                   !
                   ! Potential set calculator
                   ! marks s0 and s1 as having potential
                   LL1:
                    stb %g0,[%i5+%lo(_P+0)]
                    stb %g0,[%i5+%lo(_P+1)]
                    ba PO0
                    nop
                   !
                   ! Code for emit
                   POI2:
                    ⋮
```

The potential set calculation for the `require` instruction is static, since we know which segments of code (in particular, the *first* instructions of the subprocesses of every `try`) may be executed in the current instant.

4.5 The `try` Instruction

`try` is the most complex of all the intermediate instructions, being responsible for parallel execution, preemption, and exception handling. Because of this, it also has significant responsibilities related to the potential set.

A `try` instruction's behavior changes with time. For example, it examines any watched signals only after the first instant. The change is accomplished by returning different program counters (i.e., not always pointing to the beginning of the code for the `try`) as appropriate, and through the use of the halt array.

Most of the information a `try` instruction needs is available at compile time. For example, it will always check the same signals and exceptions. This makes for very simple, loop-free code in the executable.

A `try` instruction has the following structure:

```
PCi ... PCj = 0                               initialize subprocess program counters
Ek ... El = 0                                 lower our exceptions
repeat
    A:                                        waiting for processes to run
    call PCi ... PCj                          call each of the subprocesses
    if any processes returned waiting         a process has not completed
        return waiting, PC = A
    if any exception was raised               check the exceptions after completion
        branch to its handler
    if any process returned halted            process is done for this instant,
        return halted, PC = B                 but is still active
    branch to the next instruction            all subprocesses have terminated
    B:                                        waiting for watching signals
    if this process has been halted           halted in this instant
        return halted, PC = B
    if any watched signals are unknown        need to know about all
        return waiting, PC = B                watched signals before proceeding
    decrement the counter of any counted
       occurrence whose signal is present
    if any occurrence elapsed                 counter became zero, or
        branch to its handler                 simple occurrence's signal present
end repeat
```

The full potential-set calculator routine for a `try` is much more elaborate than that used by `require`:

```
if process is not halted
    call POi ... POj                          potential set routines of subprocesses
    mark potential set of each watch handler
    mark potential set of each exception handler
    mark potential set of next instruction
return
```

4.6 Example

The SPARC translation of the program of Figure 3.1 on page 30 is shown in Figure 4.1 on page 41. The assembly code for the `try` instruction has been placed in the second column for clarity.

Each instruction is given a label such as `P0I1`, which indicates process zero, instruction one.

```
Local variables
v0:  C

Signals 4( 1 valued )
s0:  TICK
s1:  A
s2:  D
s3:  B(int)

Process P0 uses PC0
0:  r0 := 3
1:  v0 := r0
2:  try
      call P1
    watching s1 goto 3
3:  require s2
4:  if not s2 goto 10
5:  r0 := v0
6:  r1 := 3
7:  r0 := r0 + r1
8:  emit s3 r0
9:  goto 14
10:  r0 := v0
11:  r1 := 2
12:  r0 := r0 - r1
13:  emit s3 r0
14:  goto 0

Process P1
  of process P0 uses PC1
0:  halt
```

```
!  Process P0 uses PC0
!
P0I0:
    mov 3,%l0
P0I1:
    st %l0,[%i5+%lo(_V+0)]
P0I2:
P0I3:
!  Potential signals:  3
    ldsb [%i5+%lo(_S+2)],%o0
    tst %o0
    bne P0I4
    nop
LL7:
    set P0I3,%i2
    set LL8,%i3
    ba PR0
    or %i4,-1,%i4
LL8:
    stb %g0,[%i5+%lo(_P0+12)]
    ba P00
    nop
P0I4:
    ldsb [%i5+%lo(_S+2)],%o0
    tst %o0
    bneg P0I10
    nop
P0I5:
    ld [%i5+%lo(_V+0)],%l0
P0I6:
    mov 3,%l1
P0I7:
    add %l0,%l1,%l0
P0I8:
    st %l0,[%i5+%lo(_SV+0)]
    mov 1,%o0
    stb %o0,[%i5+%lo(_S+3)]
P0I9:
    ba P0I14
    nop
P0I10:
    ld [%i5+%lo(_V+0)],%l0
P0I11:
    mov 2,%l1
P0I12:
    sub %l0,%l1,%l0
P0I13:
    st %l0,[%i5+%lo(_SV+0)]
    mov 1,%o0
    stb %o0,[%i5+%lo(_S+3)]
P0I14:
    ba P0I0
    nop

P1I0:
LL9:
    set LL9,%i2
    set P01,%i3
    ba PR1
    or %i4,1,%i4
```

```
!  Initializing all program counters
    set P1I0,%o0
    st %o0,[%i5+%lo(_PC+4)]
!  Lowering all exceptions
    ba LL0
    nop
LL1:
!  Checking the halted status of this
    ldsb [%i5+%lo(_H+0)],%o0
    tst %o0
    be LL3
    nop
!  Returning the halted status
    set LL1,%i2
    set LL5,%i3
    ba PR0
    or %i4,1,%i4
LL3:
!  Checking watched signals
    ldsb [%i5+%lo(_S+1)],%o0
    tst %o0
    be LL2
    nop
    bneg LL6
    nop
    ba P0I3
    nop
LL6:
LL0:
!  Executing all subprocesses
    stb %i4,[%i5+%lo(_H+0)]
    mov %g0,%i4
    ld [%i5+%lo(_PC+4)],%o0
    call %o0,0
    nop
PR1:
    st %i2,[%i5+%lo(_PC+4)]
    st %i3,[%i5+%lo(_P0+4)]
    subcc %i4,%g0,%o0
    bpos LL4
    nop
!  Return with the waiting status
    set LL0,%i2
    set LL5,%i3
    ba PR0
    stb %g0,[%i5+%lo(_H+0)]
LL4:
    ldsb [%i5+%lo(_H+0)],%i4
!  Checking all exceptions
    tst %o0
    be P0I3
    nop
!  Halting
    set LL1,%i2
    set LL5,%i3
    or %o0,%i4,%i4
    ba PR0
    stb %o0,[%i5+%lo(_H+0)]
!  Returning with waiting status
LL2:
    set LL1,%i2
    set LL5,%i3
    ba PR0
    or %i4,-1,%i4
LL5:
!  Calculating potential set
    ldsb [%i5+%lo(_H)],%o0
    tst %o0
    bne P00
    nop
    ld [%i5+%lo(_P0+4)],%o0
    call %o0,0
    nop
P01:
!  Potential signals:    3
    stb %g0,[%i5+%lo(_P+3)]
    ba P00
    nop
```

Figure 4.1: The translation of the Esterel program of Figure 3.1 from page 30.

The potential-set calculator routine for the `try` starting at `LL5` first checks the status of the halted flag for the process, returning immediately if it is set, indicating that the process has halted. In this case, since the process has halted, it cannot emit any more signals in the current instant. However, at the beginning of the next instant, the halted flag will be cleared and when the potential set calculator is called again (even before it is known that the process can be executed), the potential set will be calcuated assuming that the process could be executed.

Because each subprocess can be called from exactly one `try` instruction, the return address of each subprocess is known at compile time. In this example, for instance, process `P1` is called from the `try` in process `P0`. The return address of process `P1` is given the label `PR1`. Similarly, the potential set calculator for process `P1` is called from exactly one point, so its return address is explicit: `PO1`.

4.7 Outer Loop

The outer loop calls the outermost process routine and its potential set calculator repeatedly to compute the program's response for an instant. The outer loop performs the following actions:

$\mathtt{H}_0 \ldots \mathtt{H}_n := 0$ *clear all halted flags*
$\mathtt{S}_0 \ldots \mathtt{S}_m := 0$ *mark all signals as unknown*
$\mathtt{S}_i \ldots \mathtt{S}_j := 1$ or -1 as appropriate *mark all input signals as present or absent*
repeat
    $\mathtt{P}_0 \ldots \mathtt{P}_l = 0$ *mark signals as having no potential*
    call `PC0` *outermost process routine*
    foreach signal i
        if $\mathtt{S}_i = 0$ and $\mathtt{P}_i = 0$ *signal is unknown and has no potential*
            $\mathtt{S}_i = -1$ *mark signal as absent*
until outermost process routine returned halted

Chapter 5

Causality

In Esterel, as in any language, it is possible to write something nonsensical. Possible errors include the simple syntax error (misspelling a keyword, for example), and more elaborate semantic errors (e.g., trying to add an integer or a boolean, emitting an input-only signal, etc.). In Esterel, there is the fairly subtle concept of causality which can be violated.

A simple example of a causality error is the following paradox

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

Because of the instantaneous semantics of Esterel, this fragment means that if the signal `A` is absent in an instant, then it should be emitted in that instant, which is clearly nonsense since a signal is either present or absent, never both. Another, more common mistake is to make a `do`...`watching` preempt itself. For example,

```
do
  await A ; emit B ; emit C
watching C
```

When `A` arrives, `B` and `C` are emitted in the same instant. However, when `C` is emitted, the enclosed instruction is not executed (the semantics of the `do`), so `C` could not have been emitted. (A way to get around this particular problem is to replace the `do`...`watching` with a `trap` instruction.)

These two errors would be fairly easy to catch at compile time—the rule is that any instruction whose execution depends on a signal cannot emit that signal. However, manifestation of these sorts of errors can be arbitrarily subtle. Consider the following sequence

```
await B ; present A else emit C end
||
await C ; emit D
||
await D ; emit E
||
await E ; emit A
```

Here, when `B` arrives, `C` is emitted if `A` is present. However, `A` is emitted if `E` is present, and `E` is emitted if `D` is present, and `D` is emitted if `C` is present. It is subtle, but this is the same sort of paradox as `present A else emit A end`.

To illustrate how subtle such violations can be, consider the following variation on this code:

```
await B ; present A else emit C end
||
await B ; await C ; emit D
||
await D ; emit E
||
await E ; emit A
```

This does *not* constitute a causality violation. Assuming `B` is the only input signal, when `B` occurs, `D` cannot be emitted because the `await C` will only start looking for `C` in the *next* instant. Thus, `D` is not emitted, so `E` and `A` are not emitted, so `C` is.

However, by adding one keyword, the fragment again becomes non-causal:

```
await B ; present A else emit C end
||
await B ; await immediate C ; emit D
||
await D ; emit E
||
await E ; emit A
```

Here is another paradox:

```
every A do
  present C then emit D end
end
||
every A do
  present D else emit C end
end
```

Here, when `A` is present, the first line says that `C` implies `D`, but the second says `D` implies not `C`, which is paradoxical.

However, the data-dependent actions make the problem even more subtle. Consider

```
var i := 0 : integer in
  every A do
    i := i + 1 ;
    emit B(i)
  end
  ||
  every B do
    if ?B mod 2 = 1 then
      present C then emit D end
    end
  end
  ||
  every B do
    if ?B mod 2 = 0 then
      present D else emit C end
    end
  end
end
```

This is actually causal since it is impossible for both `?B mod 2 = 0` and `?B mod 2 = 1` to be true in the same instant (the value returned by `?B` is unique to an instant).[1]

So in general, exact causality checking is impractical. The Esterel V3 compiler simulates only the signal portion of the program as part of the compilation process (i.e., not the data portion), which accounts for its rapidly-growing compilation times. It can detect causality violations, but it is conservative and it requires excessively long compilation times.

Some causality checking is necessary, but the conservative approach taken by the Esterel V3 compiler requires too much time to perform.

[1]Unsurprisingly, the Esterel V3 compiler flags this example as noncausal.

Chapter 6

Results and Conclusions

The compiler presented in this report was tested on the lengthy example (and variations thereon) presented in appendix C. For comparison, it was also tested with the production Esterel V3 compiler supported by CISI INGENIERIE[8]. These results are shown in Tables 6.1 and 6.2 on pages 47 and 47.

The examples watch1, watch2, watch3, and watch4 are stripped-down versions of the complete watch example, containing the first one, two, three, and four submodules respectively.

6.1 Results for The Esterel V3 Compiler

Please refer to Table 6.1. With the Esterel V3 compiler, the number of states in the finite-state machine starts small, but grows quickly with the size of the input file. The length of the `ic` file, which contains an intermediate representation similar to the one used here, is growing roughly linearly with the length of the input file, as is to be expected. However, the length of the `oc` file, which contains a description of the state machine used to produce the C source, is growing very rapidly—exponentially for this example.

The C source file produced is roughly the same size as the `oc` file, so the system's C compiler is presented with a challenge: C source files no smaller than 980K. Not surprisingly, the time required to produce the C source file is rapidly getting out of hand, starting at a minute and increasing by about a factor of four for each 200-line increase in the length of the source file.

The size of the executable produced by this compiler tracks the size of the C source file, and is also growing very rapidly. The one consolation is that the time required to simulate one thousand clock ticks is both small and growing slowly.

	watch1	watch2	watch3	watch4	watch
lines in source file	297	467	619	823	998
number of states	7	22	32	128	>206
length of `ic` file (kilobytes)	16	25	35	45	52
length of `oc` file (megabytes)	0.98	5.42	18.7	190	>236
time to create C source file (mm:ss)	0:52	4:43	15:57	>37:00†	
time to compile C source (mm:ss)	1:50	15:30	18:43†		
size of MIPS executable (megabytes)	0.87	3.7	12.2		
time to simulate 1000 clock ticks (seconds)	2.8	4.8	6.6		

† Times on a machine roughly 3× faster

Table 6.1: The Esterel V3 compiler used on the watch example from appendix C

	watch1	watch2	watch3	watch4	watch
lines in source file	297	467	619	823	998
number of processes	45	83	110	150	178
number of program counters	34	50	58	78	97
number of signals	40	45	54	62	70
length of translation file (kilobytes)	10	17	24	31	36
lines of assembly code (thousands)	5.6	9.9	13	18	21
time to produce assembly code (seconds)	1.7	2.5	3.1	3.7	4.5
time to assemble (seconds)	3	4	5.5	7.8	8.5
size of SPARC process code (kilobytes)	14	26	35	48	56
size of SPARC executable (kilobytes)	64	80	96	112	128
time to simulate 1000 clock ticks (seconds)	2.3	2.6	3.2	3.8	4.2
iterations per clock tick	9	9	9	9	9

Table 6.2: This compilation scheme used on the watch example from appendix C

6.2 Results for This Compiler

Please refer to Table 6.2. With the compilation scheme presented in this report, the number of processes and program counters are both growing roughly linearly with the size of the source file. Moreover, the length of the translation (roughly equivalent to the `ic` file—a textual listing of the intermediate representation) is growing linearly. The really encouraging result is that the number of assembly code lines is also growing linearly with the size of the source file, as expected.

The time required to produce this assembly code is very small indeed (this time includes parsing the Esterel source file, converting it to the intermediate representation, and translating this into SPARC assembly code), and also appears to be linear in the length of the Esterel source file, very much unlike the Esterel V3 compiler.

It takes roughly twice as long to run the assembly code through the assembler[1] as it does to compile the Esterel source code.

Not surprisingly, the executable produced (which includes a simple command-line interface, so comparison with the MIPS executable is reasonable), is much smaller. For example, the 600-line source file watch3 produces a 12 megabyte executable with thee Esterel V3 compiler, whereas this approach produces a 128 kilobyte executable, nearly two orders of magnitude smaller!

Finally, the time to simulate 1000 clock ticks is roughly comparable to the Esterel V3 compiler (the two machines used were roughly the same speed: a SPARCstation IPC and a DECstation 5000). The number of calls to the process code for the signal information to converge is between one and ten for all five examples, with nine being the overwhelming median for all.

6.3 Comments

It is not entirely fair to compare the compilation times for two compilers, as the Esterel V3 compiler is doing full causality checking. The compiler presented here does effectively none, instead leaving the checking to runtime. However, the times and file sizes for the V3 compiler become prohibitive very quickly. For example, I was unable to find a machine with the 400 MB of free disk space required produce the C source code for the watch4 example. Even for the smaller watch2 example, waiting twenty minutes for the program to compile would have made debugging agonizing, to say the least. The multi-megabyte executables are also infeasible for embedded systems. Although the amount of memory available in such systems has been growing rapidly over time, these seem excessive.

So the question of whether this language is practical without full compile-time causality checking arises. In the process of writing and debugging the watch example of appendix C, it turned out not to be a major issue. If a

[1] Sun's standard SPARC assembler shipped with SunOS 4.1.1 for these examples

causality violation was introduced (and quite a number were over the course of development), it would become clear fairly quickly—either the whole watch would fail, or the module where the violation was introduced would fail in an obvious manner. The Tcl/Tk mock-up described in appendix C was an invaluable debugging aid. Instead of entering test vectors in a textual manner and observing the results, I simply used the mock-up like I used the watch on my wrist—the bugs became apparent fairly quickly.

Esterel is a good language for describing very heterogeneous systems. In such cases, little code could possibly be reused—every problem has to be solved in a different way. However, when there is some uniformity involved, which is often the case with human-interface code, Esterel is not completely effective. For example, most of the "adjust" modes of the watch are very similar—`MODE` advances the field being adjusted, `FORWARD` and `REVERSE` adjust that field. This was implemented with a valued signal, but that technique isn't quite right. A signal with a notion of an ordered "one-hot" encoding would be better.

The designers of Esterel included the `copymodule` keyword in the language (not present in the compiler presented here) which is little more than a simple macro expansion. The watch example of appendix C used the `m4` macro preprocessor for this function, which was nearly as effective. Such preprocessors can improve the readability of the code, and somewhat simplify the programmer's task, but much more is needed to capture the similarity in typical control-dominated systems.

A number of important features were omitted from the example of appendix C which were present in the real watch on which it was based. Why they were omitted sheds some light on Esterel's shortcomings.

- The true watch has a "telephone book" mode which stores about 24 names (8 alphanumeric characters each) and phone numbers (12 digits each).
- The true watch calculated the day-of-the-week from the year, month and date. Moreover, it knows about the number of days in each month and leap years.
- The true watch has five alarms and a hourly signal.

The telephone book mode was omitted primarily because storing the data would have been difficult in Esterel. The Esterel V3 compiler has the ability to import complex types from a host language (e.g., C) and attach them to signals, local variables, and whatnot. These can then be manipulated with functions in the host language called from Esterel. These facilities are not present in the compiler presented here, but could be included. But this seems to be avoiding the problem by letting a "real" language handle the messy data manipulation.

Calculating the day-of-the-week from the year, month, and date is a fairly complex arithmetic operation which would benefit from array lookups, something not present in Esterel.

The five alarms could have been implemented with a number of instantiations of the alarm module presented here, but this produces an excessive amount of code. Since all five are identical, there should be some way to reuse the code more effectively.

6.4 Conclusions

The objective stated in the abstract has been achieved: a compiler for the Esterel language has been produced which quickly produces an efficient executable. For large programs, its performance greatly eclipses the existing compilation scheme, and allows such programs to be compiled at all. The increase in speed has come at the expense of compile-time causality checking.

There are many ways to proceed from here. Retargeting the backend to produce code for other processors is one simple modification. There are probably many more simple checks which could be performed at compile time in the hopes of catching an erroneous (i.e., non-causal) program. Also, this approach lends itself to symbolic debugging, which has not currently been implemented beyond a simple mechanism to report the locations of the program counters.

A hope for this work is for it to be used for other tasks. For example, it appears that the scheme presented here for producing assembly code from the intermediate representation could also be used to produce code from a synchronous subset of the VHDL language [2, 3]. The intermediate format also lends itself to taking event derivatives and forming an FSM. While this has shown to be potentially explosive, there may be a way of effectively partitioning a program. An FSM so generated could be sent to a formal verification system which could then prove properties about the system.

Appendix A

Lexical Aspects of Esterel

IDENTIFIERS

An identifier is a sequence of letters, digits, and underscores starting with a letter. Case is significant. There is no limit on the length of identifiers.

KEYWORDS

All keywords are lowercase.

`and`	`exit`	`mod`	`sustain`
`await`	`false`	`module`	`then`
`call`	`halt`	`not`	`timeout`
`case`	`handle`	`nothing`	`times`
`do`	`if`	`or`	`trap`
`each`	`immediate`	`output`	`true`
`else`	`in`	`present`	`upto`
`emit`	`input`	`repeat`	`var`
`end`	`inputoutput`	`sensor`	`watching`
`every`	`loop`	`signal`	`with`

INTEGER LITERALS

An integer literal is a string of digits `0`–`9`. Leading zeros are and `-` signs are allowed. Leading `+` signs are disallowed.

COMMENTS

Comments begin with a percent sign (`%`) and continue to the end of the line.

WHITESPACE

Whitespace includes comments, spaces, tabs, newlines, and form feeds, and serves to delimit identifiers, keywords, and integer literals.

Appendix B

Syntax of Esterel

In the following, items in braces ({ }) are optional. The notation { *A* }* means "zero or more occurrences of *A*," and the notation { *B* }+ means "one or more occurrences of *B*." γ-*identifier* is an identifier of type γ. Keywords are in a `typewriter` typeface.

FILES

The source file is composed of one or more *module*s.

file →

{ *module* }+

MODULES

A module contains zero or more declarations and a single instruction terminated by a period (`.`).

module →

`module` *module-identifier* :
{ *declaration* }*
instruction
`.`

DECLARATIONS

declaration →

`input` *signal-declaration-list* ;
| `output` *signal-declaration-list* ;
| `inputoutput` *signal-declaration-list* ;
| `sensor` *sensor-declaration-list* ;

Signal Declaration Lists

Signals are declared with comma-separated lists of one or more signals, each with an optional type.

signal-declaration-list →
    signal-declaration { `,` *signal-declaration* }*

signal-declaration →
    signal-identifier
  | *signal-identifier* `(` *type-identifier* `)`

type-identifier →
    `integer`
  | `boolean`

sensor-declaration-list →
    sensor-declaration { `,` *sensor-declaration* }*

sensor-declaration →
    signal-identifier `(` *type-identifier* `)`

Instructions

instruction →
    `var` *variable-declaration-list* `in` *instruction* `end`
  | `signal` *signal-declaration-list* `in` *instruction* `end`
  | `[` *instruction* `]`
  | { *instruction* `;` }+ { *instruction* }
  | *instruction* { `||` *instruction* }+
  | `nothing`
  | `halt`
  | `exit` *exception-identifier* { `(` *expression* `)` }
  | *variable-identifier* `:=` *expression*
  | `if` *expression* { `then` *instruction* }
    { `else` *instruction* } `end`
  | `loop` *instruction* `end`
  | `repeat` *expression* `times` *instruction* `end`
  | `emit` *signal-identifier* { `(` *expression* `)` }
  | `sustain` *signal-identifier* { `(` *expression* `)` }
  | `present` *signal-identifier* { `then` *instruction* }
    { `else` *instruction* } `end`
  | `do` *instruction* `watching` *occurrence*
    { `timeout` *instruction* `end` }

| `await` *occurrence* { `do` *instruction* `end` }
| `await` { `case` *occurrence* { `do` *instruction*} }$^+$ `end`
| `loop` *instruction* `each` *occurrence*
| `do` *instruction* `upto` *occurrence*
| `every` *occurrence* `do` *instruction* `end`
| `trap` *exception-declaration* `in` *instruction*
{ `handle` *exception-identifier* `do` *instruction* } `end`

exception-declaration →
exception-identifier
| *exception-identifier* `(` *type-identifier* `)`

VARIABLE DECLARATION LISTS

Variables are declared in comma-separated lists with optional initialization expressions and type specifications.

variable-declaration-list →
variable-declaration { `,` *variable-declaration* }*

variable-declaration →
variable-identifier { `:=` *expression* }
{ `:` *type-identifier* }

EXPRESSIONS

expression →
integer-literal
| `true`
| `false`
| *variable-identifier*
| `?`*signal-identifier*
| `?`*sensor-identifier*
| `??`*exception-identifier*
| `(` *expression* `)`
| `-` *expression*
| *expression* `*` *expression*
| *expression* `/` *expression*
| *expression* `mod` *expression*
| *expression* `+` *expression*
| *expression* `-` *expression*
| *expression* `<` *expression*
| *expression* `<=` *expression*
| *expression* `>` *expression*

| *expression* `>=` *expression*
| *expression* `=` *expression*
| *expression* `<>` *expression*
| `not` *expression*
| *expression* `and` *expression*
| *expression* `or` *expression*

Occurrences

occurrence →
signal-identifier
| `immediate` *signal-identifier*
| *expression* *signal-identifier*

Appendix C

A Large Example

To illustrate the effectiveness of the compiler on a large program, the following program was created. It describes the functionality of a fairly elaborate digital wristwatch.

The watch has five main features, shown in Figure C.1.

- Time keeping including the day, date, month and year
- An alarm which may be set to particular time and date
- A dual timezone mode
- A settable countdown timer
- A stopwatch

These features are controlled through four buttons:

- MODE, which switches between the above-listed modes, or between fields when setting the watch. If no other buttons have been pressed, MODE switches to the next mode, otherwise, it returns to the timekeeping mode.
- ADJUST, which toggles between setting and running for the above modes
- FORWARD, used to increase the value of a field when setting
- REVERSE, used to decrease the value of a field when setting

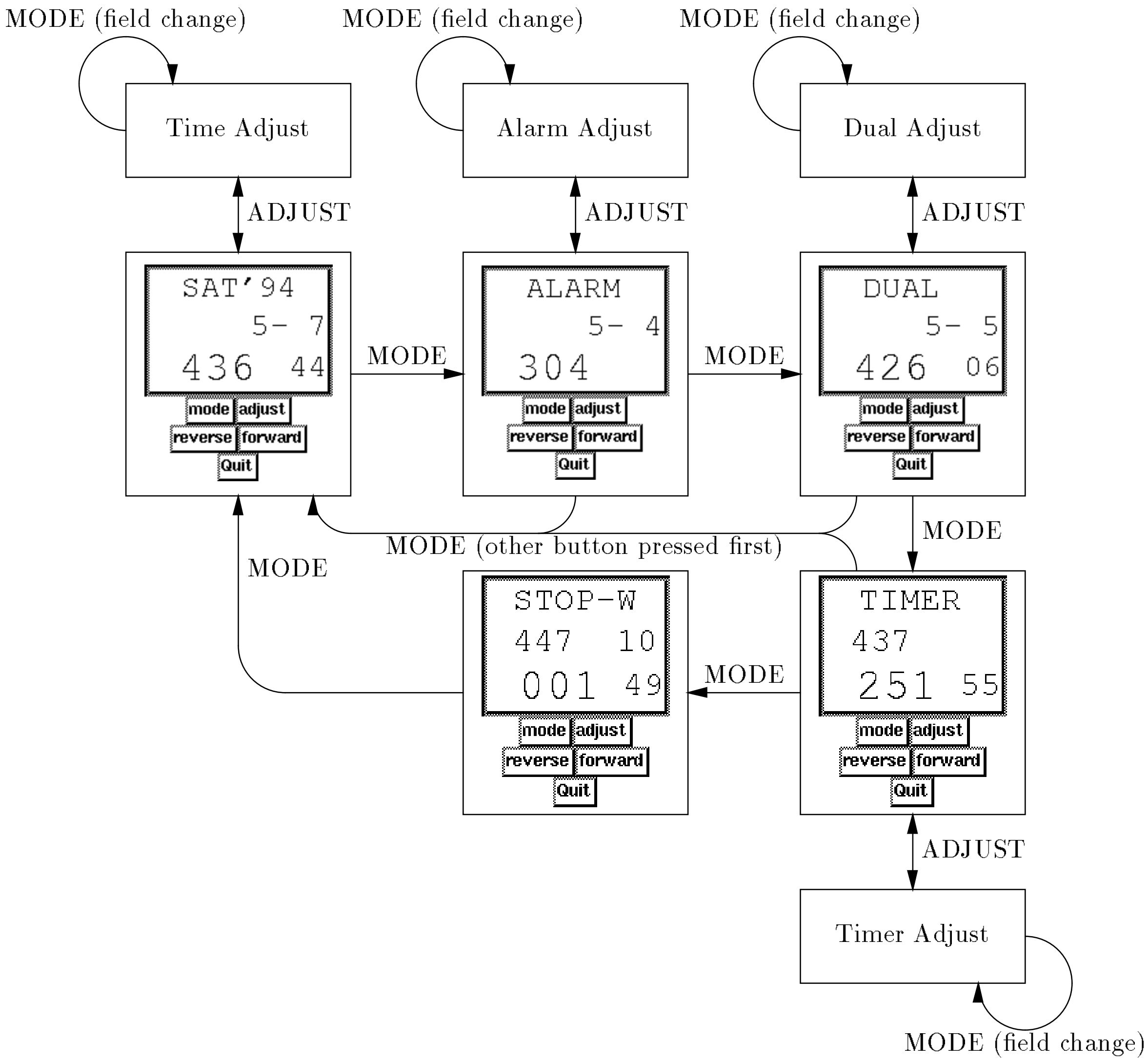


Figure C.1: The watch's modes.

C.1 Testing Scheme

A mock-up of an actual watch was created using the Tcl/Tk system [10] for building user interfaces. A short ($\approx$ 40 line) `tcl` script describes a window with four text items and five buttons. Four of these buttons call simple routines in a small C program ($\approx$ 400 lines) when pressed, the other quits the program. Each of these simple routines set the given button to present and calls the `tick` routine, which calculates the Esterel program's response for the next instant. The C program then examines the emitted signals and adjusts the display accordingly through `tcl` commands.

A facility for creating periodic events in Tk is used to call a C routine approximately ten times a second. This C routine sets the `TENTHSECOND` signal present and calls `tick`. Not surprisingly, since this timekeeping mechanism is fairly inexact, the watch keeps less-than-perfect time. Nevertheless, this scheme made the debugging process much more efficient than had it been attempted using just a command-line-based simulator.

C.2 The Main Module

The watch receives its controls through five signals, one which occurs ten times a second, and one for each button.

```
module watch:

input TENTHSECOND;

input MODE, ADJUST, REVERSE, FORWARD;
```

The display is controlled through two valued signals, one which indicates which major mode is being displayed (time, time adjust, alarm, alarm adjust, etc), and one which, in the adjust modes, indicates which field is being changed.

```
output  DISPLAY_MODE(integer),
        DISPLAY_FIELD(integer);
```

A separate valued signal is used for each field from the five major components. In effect, the C program acts as a multiplexer which selects which of these fields to copy to the display based on `DISPLAY_MODE` and `DISPLAY_FIELD`.

These are integer-valued signals, but it is useful to think of them as taking on the following symbolic values. The various `_modes` define the values `DISPLAY_MODE` takes on and the various `_adjusts` define values for `DISPLAY_FIELD`.

```
define(`Time_mode',`0')
define(`Time_adjust_mode',`1')
```

```
        define(`Time_adjust_seconds',`0')
        define(`Time_adjust_minutes',`2')
        define(`Time_adjust_hours',`1')
        define(`Time_adjust_days',`6')
        define(`Time_adjust_dayofweek',`3')
        define(`Time_adjust_months',`5')
        define(`Time_adjust_years',`4')
define(`Alarm_mode',`2')
define(`Alarm_adjust_mode',`3')
        define(`Alarm_adjust_minutes',`1')
        define(`Alarm_adjust_hours',`0')
        define(`Alarm_adjust_days',`3')
        define(`Alarm_adjust_months',`2')
define(`Dual_mode',`4')
define(`Dual_adjust_mode',`5')
        define(`Dual_adjust_minutes',`1')
        define(`Dual_adjust_hours',`0')
        define(`Dual_adjust_days',`3')
        define(`Dual_adjust_months',`2')
define(`Timer_mode',`6')
define(`Timer_adjust_mode',`7')
        define(`Timer_adjust_minutes',`1')
        define(`Timer_adjust_hours',`0')
define(`Stopwatch_mode',`8')

output  MAIN_SECOND(integer),
        MAIN_MINUTE(integer),
        MAIN_HOUR(integer),
        MAIN_DAY(integer),
        MAIN_DATE(integer),
        MAIN_MONTH(integer),
        MAIN_YEAR(integer);

output  DUAL_MINUTE(integer),
        DUAL_HOUR(integer),
        DUAL_DAY(integer),
        DUAL_DATE(integer),
        DUAL_MONTH(integer);

output  TIMER_HOUR(integer),
        TIMER_MINUTE(integer),
        TIMER_SECOND(integer);

output  ALARM_MINUTE(integer),
```

```
        ALARM_HOUR(integer),
        ALARM_DATE(integer),
        ALARM_MONTH(integer);

output  STOPWATCH_TENTHSECOND(integer),
        STOPWATCH_SECOND(integer),
        STOPWATCH_MINUTE(integer),
        STOPWATCH_HOUR(integer);
```

Four other signals perform miscellaneous tasks.

`UPDATE` is present whenever the display needs to be updated, e.g., when the mode changes, when a value changes, etc.

`ALARM` is present whenever the alarm is going off.

`FLASH` is present whenever the watch is in an adjust mode and the requested field should be blanked. This is used to indicate which field is currently being changed.

`STOPWATCH_FROZEN` is present whenever the stopwatch is in lap mode (keeping time, but display holds previous value). When in stopwatch mode, the display indicates when it is in lap mode.

```
output  UPDATE;
output  ALARM;
output  FLASH;
output  STOPWATCH_FROZEN;
```

The main module is composed of five processes executing in parallel, one for each major mode (the time and dual modes are combined here), and one which controls the operation of the display. The `m4` macro preprocessor was used to break the program into modules. Thus, each of the `_module`s and `_watchdog`s here actually expand into a number of instructions which will be described in later sections.

```
  signal MAIN_REVERSE, MAIN_FORWARD, DUAL_REVERSE, DUAL_FORWARD,
         ALARM_REVERSE, ALARM_FORWARD, TIMER_STARTSTOP, TIMER_RUNNING,
         TIMER_REVERSE, TIMER_FORWARD, TIMER_RESET,
         STOPWATCH_FREEZETHAW,
         STOPWATCH_STARTSTOP, STOPWATCH_RUNNING,
         STOPWATCH_RESET in

    time_module
    ||
    alarm_module
```

```
    ||
    timer_module
    ||
    stopwatch_module
    ||
    loop
      trap Return_to_time in
        time_control_module ;
        alarm_control_module ;
        dual_control_module ;
        timer_control_module ;
        stopwatch_control_module
      end
      ;
      await TICK
    end

  end
.
```

The main loop is a sequence of control modules (instructions) enclosed by a `trap` instruction. If one of the control modules terminates, it starts the next module. However, if it issues the `Return_to_time` exception, the watch returns to timekeeping mode.

The local signals are used to adjust the various timekeeping mechanisms and adjust their modes. These form the communication path between the control modules and the modules which actually keep the time.

C.3 The Time Module

The time module, responsible for keeping track of the main time as well as the time for the dual-timezone mode, is composed of a number of sub-modules executing in parallel, one for each unit of time.

Again, local signals are used for communcation between the various modules.

```
define(`time_module',`
  signal NEXT_SECOND, NEXT_MINUTE,
        MAIN_NEXT_HOUR, MAIN_NEXT_DAY, MAIN_NEXT_MONTH, MAIN_NEXT_YEAR,
        DUAL_NEXT_HOUR, DUAL_NEXT_DAY, DUAL_NEXT_MONTH, DUAL_NEXT_YEAR,
        MAIN_RESET_SECOND in

    every MAIN_FORWARD do
      if ?DISPLAY_FIELD = Time_adjust_seconds then emit MAIN_RESET_SECOND end
    end
```

```
    ||
    tenthsecond_module
    ||
    second_module
    ||
    minute_module(`MAIN',`Time_adjust')
    ||
    minute_module(`DUAL',`Dual_adjust')
    ||
    hour_module(`MAIN',`Time_adjust')
    ||
    hour_module(`DUAL',`Dual_adjust')
    ||
    day_module(`MAIN',`Time_adjust')
    ||
    day_module(`DUAL',`Dual_adjust')
    ||
    dayofweek_module(`MAIN',`Time_adjust')
    ||
    month_module(`MAIN',`Time_adjust')
    ||
    month_module(`DUAL',`Dual_adjust')
    ||
    year_module(`MAIN',`Time_adjust')

  end
')
```

The tenthsecond module is simple. It emits the `NEXT_SECOND` signal every ten tenths of a second, or resets itself when necessary.

```
define(`tenthsecond_module',`
  loop
    every 10 TENTHSECOND do
      emit NEXT_SECOND
    end
  each MAIN_RESET_SECOND
')
```

The second module is slightly more complex because the second count must be visible to the outside world (through the `MAIN_SECOND` signal).

```
define(`second_module',`
  loop
    var second := 0 : integer in
```

```
      emit MAIN_SECOND(second);

      every NEXT_SECOND do
        second := second + 1;
        if second = 60 then second := 0; emit NEXT_MINUTE end;
        emit UPDATE ; emit MAIN_SECOND(second)
      end

    end
  each MAIN_RESET_SECOND
')
```

The minute module is more complex still, since the minute must be able to be increased and decreased. The `await...case` instruction is used to arbitrate between which of the three actions to take.

Using the `?DISPLAY_FIELD` signal in this manner is somewhat unsatisfactory—there is a slight possibility that some other field is adjusted just when `NEXT_MINUTE` signal is present, causing the minute to be lost. One solution would be to "fan out" the `MAIN_FORWARD` signals based on the value of `DISPLAY_FIELD` in a manner similar to `MAIN_RESET_SECOND`. But this is not particularly elegant. What is needed is a more sophisticated notion of a signal, one which has a "one-hot" notion associated with it.

Two instantiations of the minute module are used, one for the main timekeeper and one for the dual-timezone mode. `$1` and `$2` are replaced with `MAIN` and `Time_adjust`, or `DUAL` and `Dual_adjust` respectively.

```
define(`minute_module',`
  var minute := 0 : integer in

    emit $1_MINUTE(minute);

    loop
      await
        case $1_FORWARD do
          if ?DISPLAY_FIELD = $2_minutes then minute := minute + 1 end
        case $1_REVERSE do
          if ?DISPLAY_FIELD = $2_minutes then minute := minute - 1 end
        case NEXT_MINUTE do
          minute := minute + 1 ; if minute = 60 then emit $1_NEXT_HOUR end
      end ;
      minute := (minute + 60) mod 60;
      emit UPDATE ; emit $1_MINUTE(minute)
    end
```

```
  end
')
```

The hour module is almost identical to the minute module, and the pair could probably have been written as a more elaborate macro.

```
define(`hour_module',`
  var hour := 0 : integer in

    emit $1_HOUR(hour);

    loop
      await
        case $1_FORWARD do
          if ?DISPLAY_FIELD = $2_hours then hour := hour + 1 end
        case $1_REVERSE do
          if ?DISPLAY_FIELD = $2_hours then hour := hour - 1 end
        case $1_NEXT_HOUR do
          hour := hour + 1 ; if hour = 24 then emit $1_NEXT_DAY end
      end ;
      hour := (hour + 24) mod 24;
      emit UPDATE ; emit $1_HOUR(hour)
    end

  end
')
```

The day module is slightly different. It wraps around at one instead of zero. This is fairly simple-minded since all months are assumed to have 31 days.

```
define(`day_module',`
  var date := 1 : integer in

    emit $1_DATE(date);

    loop
      await
        case $1_FORWARD do
          if ?DISPLAY_FIELD = $2_days then date := date + 1 end
        case $1_REVERSE do
          if ?DISPLAY_FIELD = $2_days then date := date - 1 end
        case $1_NEXT_DAY do
          date := date + 1 ; if date = 32 then emit $1_NEXT_MONTH end
      end ;
```

```
      date := (date + 30) mod 31 + 1;
      emit UPDATE ; emit $1_DATE(date)
    end

  end
')
```

The day-of-the-week, month, and year modules are all very similar:

```
define(`dayofweek_module',`
  var day := 0 : integer in

    emit $1_DAY(day);

    loop
      await
        case $1_FORWARD do
          if ?DISPLAY_FIELD = $2_dayofweek then day := day + 1 end
        case $1_REVERSE do
          if ?DISPLAY_FIELD = $2_dayofweek then day := day - 1 end
        case $1_NEXT_DAY do
          day := day + 1
      end ;
      day := (day + 7) mod 7;
      emit UPDATE ; emit $1_DAY(day)
    end

  end
')

define(`month_module',`
  var month := 1 : integer in

    emit $1_MONTH(month);

    loop
      await
        case $1_FORWARD do
          if ?DISPLAY_FIELD = $2_months then month := month + 1 end
        case $1_REVERSE do
          if ?DISPLAY_FIELD = $2_months then month := month - 1 end
        case $1_NEXT_MONTH do
          month := month + 1 ; if month = 13 then emit $1_NEXT_YEAR end
      end ;
```

```
      month := (month + 11) mod 12 + 1;
      emit UPDATE ; emit $1_MONTH(month)
    end

  end
')

define(`year_module',`
  var year := 94 : integer in

    emit $1_YEAR(year);

    loop
      await
        case $1_FORWARD do
          if ?DISPLAY_FIELD = $2_years then year := year + 1 end
        case $1_REVERSE do
          if ?DISPLAY_FIELD = $2_years then year := year - 1 end
        case $1_NEXT_YEAR do
          year := year + 1
      end ;
      year := (year + 100) mod 100;
      emit UPDATE ; emit $1_YEAR(year)
    end

  end
')
```

C.4 The Time Control Module

The time control module is responsible for handling the four buttons when the watch is in the main timekeeping mode.

```
define(`time_control_module',`
  trap Leave_time in
    loop

      % Time display

      emit DISPLAY_MODE(Time_mode) ; emit UPDATE ;

      do
       await MODE; exit Leave_time
```

```
      watching ADJUST

      ;

      emit DISPLAY_MODE(Time_adjust_mode) ; emit UPDATE ;

      do
        time_adjust_module
      watching ADJUST

    end % time display/adjust loop

  end % Leave_time
')
```

An exception is used to detect when this module should terminate, i.e., when `MODE` is presessed from within. `ADJUST` toggles between the main time display mode and the time adjust mode.

The time adjust module cycles through the fields (using the variable **`setfield`**) when `MODE` is pressed, converts the `REVERSE` and `FORWARD` buttons to their time-adjusting counterparts, and flashes the display field.

```
define(`time_adjust_module',`
  var setfield := 0 : integer in

    emit DISPLAY_FIELD(setfield) ; emit UPDATE ;
    [
    every MODE do
      setfield := setfield + 1;
      if setfield = 7 then setfield := 0; end ;
      emit DISPLAY_FIELD(setfield) ; emit UPDATE
    end
    ||
    every REVERSE do
      emit MAIN_REVERSE
    end
    ||
    every FORWARD do
      emit MAIN_FORWARD
    end
    ||
    flash_enable_module
    ]
```

```
  end % var setfield
')
```

The flash module is simple—it emits the **FLASH** signal on and off, synchronized with the TENTHSECOND signal.

```
define(`flash_enable_module',`
  loop
    emit UPDATE ; await 2 TENTHSECOND ;
    emit FLASH ; emit UPDATE ; await TENTHSECOND ;
    emit FLASH ; await TENTHSECOND
  end
')
```

C.5 The Alarm Module

Much like the time module, the alarm module is composed of a number of submodules executing in parallel. Every minute, the alarm is checked against the main clock to see if the alarm should be started. If so, it is started and sustained until one of the main buttons is pressed.

```
define(`alarm_module',`
  signal STARTALARM in

    alarm_minute_module
    ||
    alarm_hour_module
    ||
    alarm_date_module
    ||
    alarm_month_module
    ||
    every MAIN_MINUTE do
      if ( (?MAIN_MINUTE = ?ALARM_MINUTE) and
           (?MAIN_HOUR = ?ALARM_HOUR) and
           (?ALARM_DATE = 0 or (?ALARM_DATE = ?MAIN_DATE)) and
           (?ALARM_MONTH = 0 or (?ALARM_MONTH = ?MAIN_MONTH)) ) then
        emit STARTALARM
      end
    end
    ||
    loop
      await STARTALARM ;
      do
```

```
        do
          do
           sustain ALARM
          watching MODE
        watching FORWARD
      watching REVERSE
    end

  end
')
```

The minute, hour, date, and month modules are all fairly similar. Each uses a local variable to remember their settings, and each waits until an ALARM_FORWARD or ALARM_REVERSE signal instructs them to change.

```
define(`alarm_minute_module',`
  var minute := 0 : integer in

    emit ALARM_MINUTE(minute);

    loop
      await
        case ALARM_FORWARD do
          if ?DISPLAY_FIELD = Alarm_adjust_minutes then
           minute := minute + 1 end
        case ALARM_REVERSE do
          if ?DISPLAY_FIELD = Alarm_adjust_minutes then
          minute := minute - 1 end
      end ;
      minute := (minute + 60) mod 60;
      emit UPDATE ; emit ALARM_MINUTE(minute)
    end

  end
')

define(`alarm_hour_module',`
  var hour := 0 : integer in

    emit ALARM_HOUR(hour);

    loop
      await
        case ALARM_FORWARD do
```

```
          if ?DISPLAY_FIELD = Alarm_adjust_hours then hour := hour + 1 end
        case ALARM_REVERSE do
          if ?DISPLAY_FIELD = Alarm_adjust_hours then hour := hour - 1 end
      end ;
      hour := (hour + 24) mod 24;
      emit UPDATE ; emit ALARM_HOUR(hour)
    end

  end
')

define(`alarm_date_module',`
  var date := 0 : integer in

    emit ALARM_DATE(date);

    loop
      await
        case ALARM_FORWARD do
          if ?DISPLAY_FIELD = Alarm_adjust_days then date := date + 1 end
        case ALARM_REVERSE do
          if ?DISPLAY_FIELD = Alarm_adjust_days then date := date - 1 end
      end ;
      date := (date + 32) mod 32 ;
      emit UPDATE ; emit ALARM_DATE(date)
    end

  end
')

define(`alarm_month_module',`
  var month := 0 : integer in

    emit ALARM_MONTH(month);

    loop
      await
        case ALARM_FORWARD do
          if ?DISPLAY_FIELD = Alarm_adjust_months then month := month + 1 end
        case ALARM_REVERSE do
          if ?DISPLAY_FIELD = Alarm_adjust_months then month := month - 1 end
      end ;
      month := (month + 13) mod 13 ;
      emit UPDATE ; emit ALARM_MONTH(month)
```

```
    end

  end
')
```

C.6 The Alarm Control Module

Like the time control module, the alarm display and alarm set modes are split into two modules. One difference is that the `MODE` button has two different effects depending on whether `ADJUST` is pressed before `MODE`. If the user adjusts the alarm, `MODE` returns the watch to the main time display mode, otherwise, it advances it to the next main mode.

```
define(`alarm_control_module',`
  trap Leave_Alarm in
    emit DISPLAY_MODE(Alarm_mode); emit UPDATE ;

    do
      await MODE; exit Leave_Alarm
    watching ADJUST

    ;

    loop

      emit DISPLAY_MODE(Alarm_adjust_mode) ; emit UPDATE ;

      do
        alarm_adjust_module
      watching ADJUST

      ;

      emit DISPLAY_MODE(Alarm_mode); emit UPDATE ;

      do
        await MODE; exit Return_to_time
      watching ADJUST

    end % alarm adjust/display mode

  end % Leave_Alarm
')
```

```
define(`alarm_adjust_module',`
  var setfield := 0 : integer in

    emit DISPLAY_FIELD(setfield) ; emit UPDATE ;

    [
    every MODE do
      setfield := setfield + 1;
      if setfield = 4 then setfield := 0 end;
      emit DISPLAY_FIELD(setfield) ; emit UPDATE ;
    end
    ||
    every REVERSE do
      emit ALARM_REVERSE
    end
    ||
    every FORWARD do
      emit ALARM_FORWARD
    end
    ||
    flash_enable_module
    ]

  end

')
```

C.7 The Timer Module

The countdown timer has a structure similar to the other modules. The main module is composed of set of sub-modules executing in parallel, one for each unit of time. The alarm for the countdown timer works in much the same way as it does for the alarm module.

```
define(`timer_module',`
  signal STARTALARM, SECOND, MINUTE, HOUR, RESET in

    timer_tenthsecond_module
    ||
    timer_second_module
    ||
    timer_minute_module
```

```
    ||
    timer_hour_module
    ||
    loop
      await STARTALARM ;
      do
        do
          do
            sustain ALARM
          watching MODE
        watching FORWARD
      watching REVERSE
    end

  end
')
```

The tenthsecond module is a little more elaborate for the countdown timer because it must monitor the state of the timer and halt when the count reaches zero, which is performed with the exception `Halt_Timer`. This functionality would be difficult with a signal used with preemption—the `do`...`watching` which monitors the signal would be controlling the count, introducing a casality violation.

```
define(`timer_tenthsecond_module',`
  var tenthsecond := 9 : integer in

    loop

      do
        every TIMER_RESET do tenthsecond := 9; emit RESET end
      watching TIMER_STARTSTOP ;

      trap Halt_Timer in
        every TENTHSECOND do
          tenthsecond := tenthsecond - 1;
          if tenthsecond = -1 then tenthsecond := 9; emit SECOND end
        end
        ||
        every SECOND do
          if ?TIMER_MINUTE = 0 and ?TIMER_HOUR = 0 and
             ?TIMER_SECOND = 0 then
           emit STARTALARM; exit Halt_Timer
          end
```

```
        end
        ||
        await TIMER_STARTSTOP ; exit Halt_Timer
      end

    end
  end
')
```

The second module is much simpler since it relies on the tenthsecond module to halt the timer when appropriate. The requirements are that the seconds be both resettable and externally-visible.

```
define(`timer_second_module',`
  var second := 0 : integer in

    emit TIMER_SECOND(second);

    loop
      await
        case RESET do
          second := 0
        case SECOND do
          second := second - 1;
          if second = -1 then second := 59; emit MINUTE; end
      end ;

      emit UPDATE ; emit TIMER_SECOND(second)

    end

  end
')
```

The minute module handles both the actual number of minutes remaining and the number of minutes which the user has requested. The reset operation is to load the number of minutes remaining with the number of minutes requested.

```
define(`timer_minute_module',`
  var minute := 0, setminute := 0 : integer in

    emit TIMER_MINUTE(minute);

    loop
      await
```

```
        case RESET do
          minute := setminute
        case TIMER_FORWARD do
          if ?DISPLAY_FIELD = Timer_adjust_minutes then
            setminute := (setminute + 61) mod 60 ; minute := setminute end
        case TIMER_REVERSE do
          if ?DISPLAY_FIELD = Timer_adjust_minutes then
            setminute := (setminute + 59) mod 60 ; minute := setminute end
        case MINUTE do
          minute := minute - 1; if minute = -1 then minute := 59; emit HOUR end
      end ;

      emit UPDATE ; emit TIMER_MINUTE(minute)

    end

  end
')
```

The hour module is similar to the minute module. Both the number of hours remaining and the number of hours requested are the responsibility of this module.

```
define(`timer_hour_module',`
  var hour := 0, sethour := 0 : integer in

    emit TIMER_HOUR(hour);

    loop
      await
        case RESET do
          hour := sethour
        case TIMER_FORWARD do
          if ?DISPLAY_FIELD = Timer_adjust_hours then
            sethour := (sethour + 61) mod 60 ; hour := sethour end
        case TIMER_REVERSE do
          if ?DISPLAY_FIELD = Timer_adjust_hours then
            sethour := (sethour + 59) mod 60 ; hour := sethour end
        case HOUR do
          hour := hour - 1
      end ;

      emit UPDATE ; emit TIMER_HOUR(hour)
```

```
    end

  end
')
```

C.8 The Timer Control Module

Because the timer has both a complex run behavior (it can be started, stopped, and reset), and a complex adjustment behavior (the hours and minutes can be set independently, and the timer is implicitly reset before adjustment), the control module is broken into three pieces, one for running, one for adjustment, and one which calls both of these, presented below. The required functionality is that after a button other than MODE has been pressed, MODE returns to the timekeeping mode, rather than the next (the stopwatch).

```
define(`timer_control_module',`
  trap Leave_Timer in
    emit DISPLAY_MODE(Timer_mode); emit UPDATE ;

    do
      timer_run_module(`Leave_Timer')
      ;
      loop
        timer_run_module(`Return_to_time')
      end

    watching ADJUST

    ;

    loop

      emit DISPLAY_MODE(Timer_adjust_mode); emit UPDATE ; emit TIMER_RESET ;

      do
       timer_adjust_module
      watching ADJUST

      ;

      emit DISPLAY_MODE(Timer_mode); emit UPDATE ;

      do
```

```
      loop
        timer_run_module(`Return_to_time')
      end
    watching ADJUST

  end  % run/adjust loop

 end   % Leave_Timer
')
```

The timer run module handles the three buttons MODE, which either returns the watch to the timekeeping mode, or sends it to the next mode (the stopwatch), FORWARD, which starts and stops the timer, and REVERSE, which stops and resets the timer.

```
define(`timer_run_module',`
  await
    case MODE do exit $1
    case FORWARD do emit TIMER_STARTSTOP
    case REVERSE do emit TIMER_RESET
  end
')
```

The timer adjust module cycles between the two fields (hour and minute), and sends `TIMER_REVERSE` and `TIMER_FORWARD` to the other timer modules. The field being adjusted is set to flash by the flash enable module, described in section C.4 on page 68.

```
define(`timer_adjust_module',`
  var setfield := 0 : integer in

    loop
      await
        case MODE do
          setfield := (setfield + 1) mod 2;
          emit DISPLAY_FIELD(setfield) ; emit UPDATE ;
        case REVERSE do
          emit TIMER_REVERSE
        case FORWARD do
          emit TIMER_FORWARD
      end
    end
    ||
    flash_enable_module
```

```
  end
')
```

C.9 The Stopwatch Module

The stopwatch module consists of parallel-executing modules for each of the units of time and two toggles which control whether the stopwatch is running and whether it is in lap mode (the display is not advancing, but the time elapsed is kept).

An exception is used to detect `STOPWATCH_STARTSTOP`. This ensures that `STOPWATCH_RUNNING` is present up to and including the instant in which `STOPWATCH_STARTSTOP` occurs.

```
define(`stopwatch_module',`
  signal SECOND, MINUTE, HOUR, RESET in

    loop
      await STOPWATCH_STARTSTOP;

      trap Stopwatch_stop in
        sustain STOPWATCH_RUNNING
        ||
        await STOPWATCH_STARTSTOP ; exit Stopwatch_stop
      end

    end
    ||
    loop
      await STOPWATCH_FREEZETHAW ; emit STOPWATCH_FROZEN ;
      do
        sustain STOPWATCH_FROZEN
      watching STOPWATCH_FREEZETHAW
    end
    ||
    stopwatch_tenthsecond_module
    ||
    stopwatch_second_module
    ||
    stopwatch_minute_module
    ||
    stopwatch_hour_module

  end
')
```

The tenthsecond module for the stopwatch is even more elaborate than that for the countdown timer. In addition to start, stop, and reset functionalities, it much also handle its display, which is affected by the lap mode.

```
define(`stopwatch_tenthsecond_module',`
  var tenthsecond := 0 : integer in

    emit STOPWATCH_TENTHSECOND(tenthsecond);

    loop

      do
        every STOPWATCH_RESET do
          tenthsecond := 0;
          emit UPDATE ; emit STOPWATCH_TENTHSECOND(tenthsecond)
        end
      watching STOPWATCH_STARTSTOP ;

      do
        loop
          await
            case STOPWATCH_RESET do
              tenthsecond := 0
            case TENTHSECOND do
              tenthsecond := tenthsecond + 1;
              if tenthsecond = 10 then tenthsecond := 0; emit SECOND end
            case STOPWATCH_FREEZETHAW
          end
          ;
          emit UPDATE ;
          present STOPWATCH_FROZEN else
            emit STOPWATCH_TENTHSECOND(tenthsecond)
          end
        end
      watching STOPWATCH_STARTSTOP

    end

  end
')
```

The stopwatch second, minute, and hour modules are comparatively simple. Each must handle reset, increasing, and the display freezing effects of the lap mode.

```
define(`stopwatch_second_module',`
  var second := 0 : integer in

    emit STOPWATCH_SECOND(second);

    loop
      await
        case STOPWATCH_RESET do
          second := 0
        case SECOND do
          second := second + 1;
          if second = 60 then second := 0; emit MINUTE; end
        case STOPWATCH_FREEZETHAW
      end ;

      present STOPWATCH_FROZEN else
        emit UPDATE ; emit STOPWATCH_SECOND(second)
      end

    end

  end
')

define(`stopwatch_minute_module',`
  var minute := 0 : integer in

    emit STOPWATCH_MINUTE(minute);

    loop
      await
        case STOPWATCH_RESET do
          minute := 0
        case MINUTE do
          minute := minute + 1;
          if minute = 60 then minute := 0; emit HOUR; end
        case STOPWATCH_FREEZETHAW
      end
      ;
      present STOPWATCH_FROZEN else
        emit UPDATE ; emit STOPWATCH_MINUTE(minute)
      end
    end
```

```
  end
')

define(`stopwatch_hour_module',`
  var hour := 0 : integer in

    emit STOPWATCH_HOUR(hour);

    loop
      await
        case STOPWATCH_RESET do
          hour := 0
        case HOUR do
          hour := hour + 1;
          if hour = 24 then hour := 0; end
        case STOPWATCH_FREEZETHAW
      end
      ;
      present STOPWATCH_FROZEN else
        emit UPDATE ; emit STOPWATCH_HOUR(hour)
      end
    end

  end
')
```

C.10 The Stopwatch Control Module

The stopwatch control module is simplified because, unlike all the other modes, it has no adjust mode. Moreover, since it is the last in the chain, it does not need to worry about the dual functionality of the MODE button, which always returns the watch to the timekeeping mode.

The `STOPWATCH_RUNNING` signal is used as a flag to distinguish between when the REVERSE button resets the stopwatch and when it toggles lap mode.

```
define(`stopwatch_control_module',`
  do

    emit DISPLAY_MODE(Stopwatch_mode); emit UPDATE;

    loop

      await
```

```
        case FORWARD do
          emit STOPWATCH_STARTSTOP
        case REVERSE do
          present STOPWATCH_RUNNING
            then emit STOPWATCH_FREEZETHAW
            else emit STOPWATCH_RESET
          end
     end

    end

  watching MODE
')
```

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley series in Computer Science. Addison-Wesley, 1988.

[2] W. Baker. An application of a synchronous reactive semantics to the VHDL language. Technical Report UCB/ERL M93/10, Unversity of California, Berkeley, 1993.

[3] W. Baker. On interfacing existing hardware description languages to state-space exploration-based verification. unpublished, June 1993.

[4] G. Berry. A hardware implementation of pure esterel. In *1991 International Workshop on Formal Methods in VLSI Design*. ACM SIG DA, January 1991.

[5] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems. In *France-Japan Artificial Intelligence and Computer Science Symposium*, 1986.

[6] Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brooks, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer-Verlag, 1984.

[7] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the Association for Computing Machinery*, 11(4):481–494, October 1964.

[8] CISI INGENIERIE. *The Esterel V3 Language Reference Manual*, 1988.

[9] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[10] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1993.