Terminal Call Processing in Esterel

Gary J. Murakami
Ravi Sethi

AT&T Bell Laboratories
Murray Hill, New Jersey 07974[1]

ABSTRACT

Each physical device attached to a node in a data network has corresponding call processes that run within the node; specifically, within a control computer in the node. A call process is responsible for the set-up and take-down of calls to and from a device. Call processes are typically complex state machines that react to hardware signals and user input. This paper describes an implementation of a terminal call process in Esterel, a special language designed for programming reactive systems. We conclude that Esterel allows clear and concise code specifications for terminal call processes; furthermore, the specifications compile into implementations.

1 INTRODUCTION

In the TDK system [5], a typical call from a terminal to a host computer proceeds as follows:

The terminal supplies the name of the host to be called.
TDK establishes a connection.
The terminal and the host use the connection to exchange data.
The host terminates the call by hanging up.

TDK, from Terminal Datakit, is an experimental software controller for a Datakit [3] switch.[2] The controller has a process per ‘‘line’’ or ‘‘device’’ attached to the switch. Such processes are referred to as *call processes*. This paper deals with call processes for terminals; hence, call process will refer to *terminal call process* unless otherwise specified. A terminal call process implements a protocol for set-up and take-down of calls from a terminal. Such protocols will be referred to as *terminal protocols*.

Although automata are natural and convenient for conveying the basic idea behind terminal protocols, their readability suffers when we use them to specify realistic protocols. A limitation of automata is that succinctly stated requirements can lead to widespread changes in an automaton. The changes examined in this paper include those resulting from a terminal being turned off prematurely, from the baud rate changing before the complete host name is supplied, and from a terminal temporarily ‘‘breaking away’’ to query the control software in the underlying network node.

Esterel is a language designed specifically for programming *reactive* systems — ‘‘systems which maintain a permanent interaction with their environment: real-time process controllers, communication protocols, man-machine interface drivers, etc. [1].’’ The output of an Esterel compiler is an automaton, which can be readily transcribed into a traditional sequential language, such as C [4]. Nevertheless, at the source level, Esterel supports synchronous threads and modules, which facilitate the development of programs that can be viewed as automata.

Section 2 outlines an experiment in which an existing terminal call process, written in C, was reimplemented using Esterel. The compiled object code for the Esterel version is comparable in time and space to the original. Section 3 examines a module from the Esterel version to illustrate the programming style and to introduce some key programming constructs. The rest of the paper contains detailed observations about Esterel and TDK.

[1] Murakami’s present address: Department of Computer Science, University of Illinois, Champaign-Urbana, Illinois 61801.

[2] This paper deals not with the AT&T product called Datakit VCS — a registered tradmark of AT&T — but with the experimental research version described in [3] and [5].

2 WHAT WE DID

Since several versions of a terminal protocol will be mentioned in this paper, it will be convenient to name them:

NAME	DATE	REMARK
Otermp		The original terminal protocol, in C.
Stermp	Oct. 1987	A simplified version, in Esterel.
Etermp	Sep. 1988	A complete version, in Esterel.
Ftermp	Nov. 1989	The final structured version, in Esterel.

This section briefly describes these versions.

Some Common Signals

State changes in a terminal protocol will be said to occur in response to input *signals*. Thus, the set of input signals corresponds to the input alphabet of the automaton for a protocol.[3]

The role of some common signals can be illustrated with respect the automaton in Fig. 1. A terminal sends an `OFFHK` signal to indicate readiness to communicate. TDK then prompts with

`Number please`

Prompts are in italics. The names of host computers are referred to as *numbers*.

The protocol in Fig. 1 expects the name of the host computer to be carried along with a signal `DATAIN`. The host name in the figure is `coma`. The connection to the host is either rejected with a `NAK` signal (due to, say, a busy or incorrect number) or accepted with an `ANSWER` signal. Acceptance means that the connection is established.

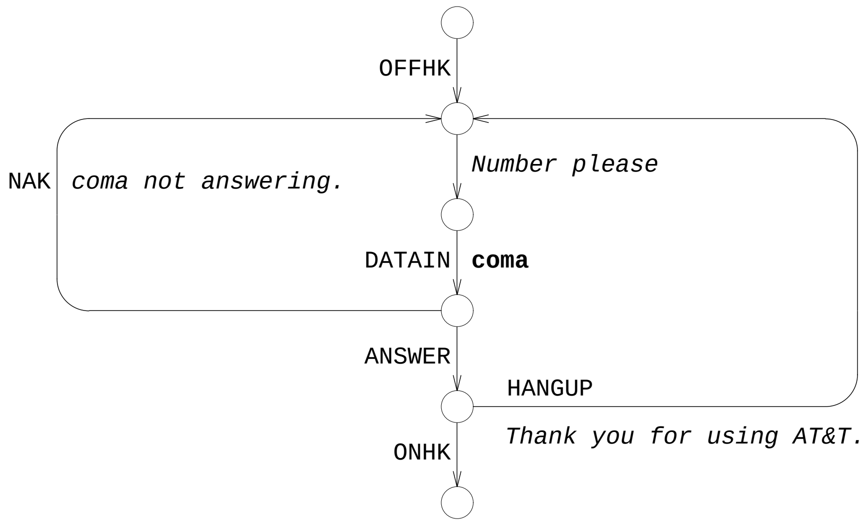

Fig. 1. A trivial terminal protocol.

[3] This notion of input signals includes some ‘‘supervision messages’’ in addition to ‘‘signals’’ in telephony. Call-processing terminology dates back to the earliest telephone networks. The signal names `ONHK` (read ‘‘on-hook’’) and `OFFHK` (read ‘‘off-hook’’) recall hooks for holding telephone receivers. When not in use, a receiver rested on a hook; it had to be taken off the hook before communication could occur. Hooks were invented in 1877. ‘‘Before 1878, there was no central-office switching and telephony was confined to private lines with a telephone at each end. The signaling problem was simple · · · the caller simply shouted into the mouthpiece using words with long vowel sounds such as ‘ahoy’ and ‘hello’ [2].’’

The call ends either with the signal `ONHK` due to the terminal being turned off, or with the `HANGUP` signal from the host computer.

Otermp: The Original Protocol

The original protocol *Otermp*, written in C, is much more complex, with escape states and special cases that obscure the essentially linear flow of control through Fig. 1. Additional features result in many more signal types and states. The states in Fig. 2 correspond to labels in the C code; the transitions correspond to goto statements. The signals under each state name correspond to transitions from a state to itself. The code is difficult to understand even after weeks of study.

Stermp: A Simplified Protocol

In October 1987, we used Esterel to specify a simplified terminal protocol, *Stermp*, which was simulated, but never fully implemented; that is, it was not run inside a switch. The code for *Stermp* consists of several modules, each less than half a page long:

termp	The main module.
session	Call processing from `OFFHK` to `ONHK`. Several calls can be placed during a session.
getString	Collect incoming characters into a string; a similar module appears in Section 3.
getDline	Translate a string into a destination.
talk	Set-up and take-down a connection.

Stermp implements a number of features from Fig. 2, yet its control flow structure is close to that of the trivial protocol in Fig. 1. In particular, *Stermp* spports the following: the baud rate can be changed during the collection of an incoming string; the terminal can send an `ONHK` signal at any time to terminate a session; a call process can ask to be suspended when a clock timeout occurs. These features benefit directly from built-in Esterel constructs.

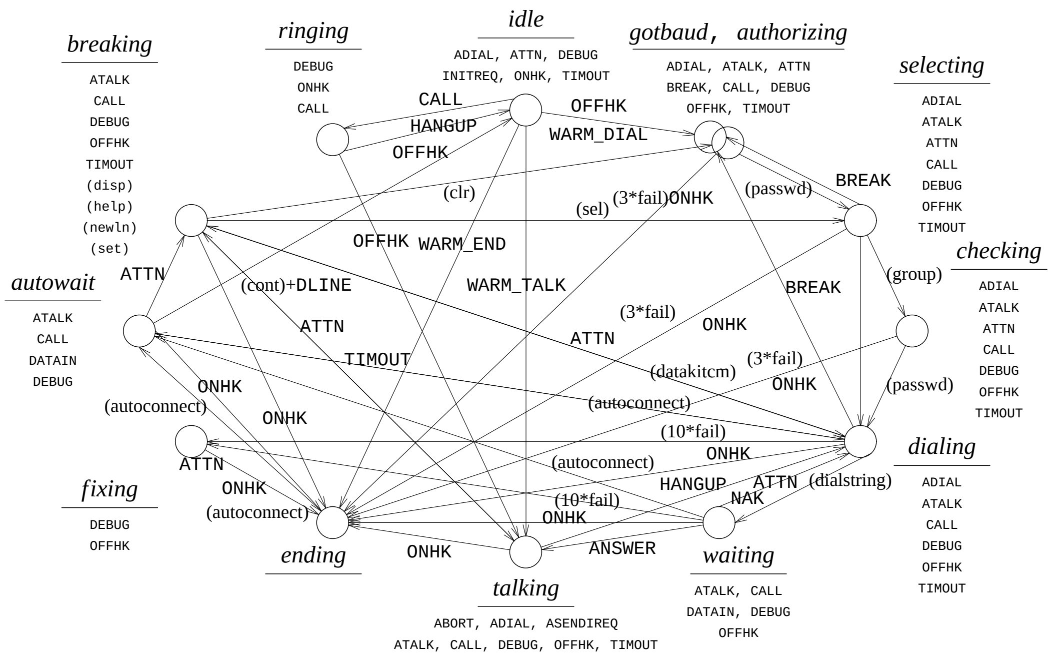

Fig. 2. The original terminal protocol.

Program		Otermp	Etermp	Ftermp
Source, lines	C	1267	469	469
	Esterel	0	576	547
Binary, bytes	Text	15498	13760	13056
	Data	5234	31066	6166
	BSS	0	0	0
	Total	**20732**	**44826**	**19222**

Fig. 3. The size of the source and object code for the executable versions.

Version	Date	States	Actions	Size
Stermp	Oct. 1987	8	51	**305**
Etermp	Sep. 1988	23	269	**13112**
Ftermp	Nov. 1989	26	139	**2176**

Fig. 4. A comparison of the Esterel versions.

The readability of *Stermp* encouraged us implement the complete terminal protocol.

***Etermp*: A Complete Version in Esterel**

The Esterel version, *Etermp* is a full implementation, consistent with *Otermp*. The C code compiled from *Etermp* was dropped into a switch and used to switch calls. *Etermp* has fewer source lines than *Otermp*; see Fig. 3. Of the 576 lines of Esterel, 450 lines specify the behavior of the reactive system. The rest of the code consists of definitions and C interface code.

Control flow in Esterel is structured. Control flow in Fig. 2 is not. *Etermp* therefore benefited from Esterel primarily at the level of the individual ‘‘states’’ in Fig. 2, which correspond to specific labels in the C code for *Otermp*. Each ‘‘state’’ begins with a C switch statement, which has a case for each signal. Control flow within the cases produces substates within each ‘‘state’’. The number of substates in *Otermp* is comparable to the 23 states in the code compiled from *Etermp*; see Fig. 4.

Etermp, has a module per ‘‘state’’ in Fig. 2. The individual modules mention only the signals that affect them. Other default actions are factored out and mentioned once in a separate module. The modules are all in parallel, awaiting a signal to begin execution. Control flow between modules is implemented by emitting a signal that ‘‘fires’’ the next module.

Although the number of states in *Etermp* is only three times the number of states in the simplified version *Stermp*, its intermediate code has over forty times the size of that for *Stermp*. This blowup led us to reimplement the terminal protocol.

***Ftermp*: The Final Esterel Version**

Figure 5 represents our present understanding of the original terminal protocol; it is based on the same code as Fig. 2, but was drawn several months later.

We draw the readers attention, not to the details of the two diagrams, but to the more uniform appearance of Fig. 5 compared to Fig. 2. Esterel provides constructs (reviewed in Section 3) that allow the merged arrows in Fig. 5 to be implemented by single statements — for example, see the arrows into *ending* on `ONHK` or the arrows into *gotbaud* on `BREAK`. We therefore believe that the cleaner appearance of Fig. 5 is

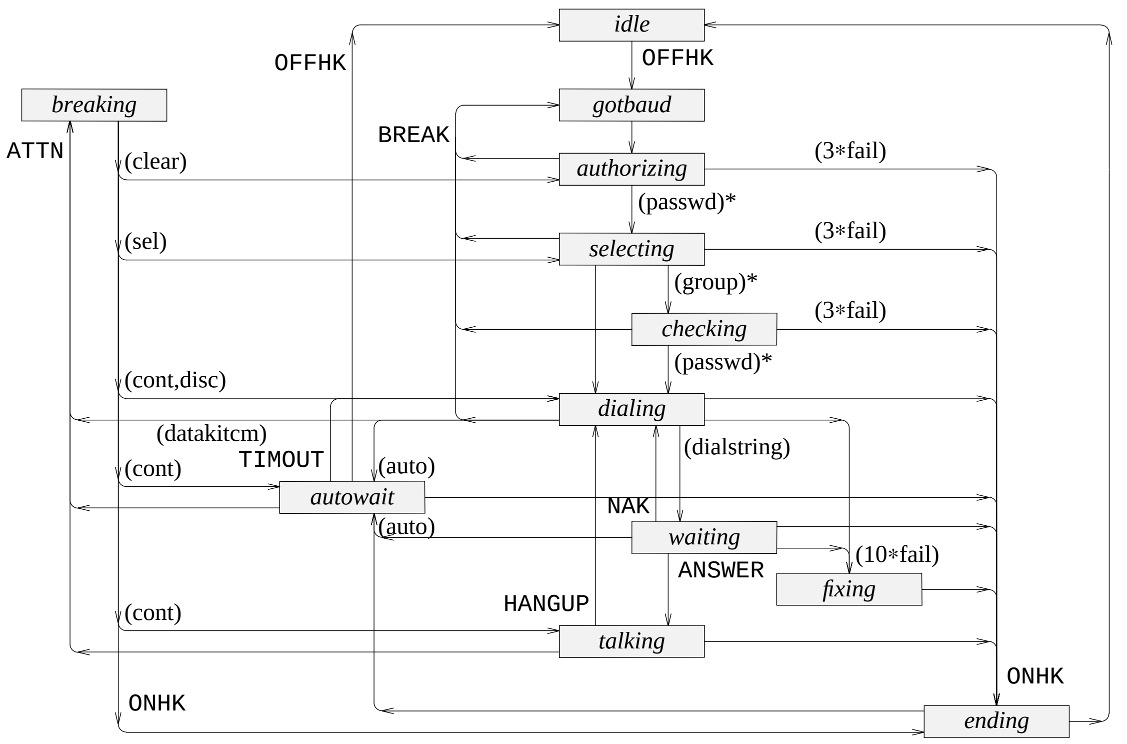

Fig. 5. The original terminal protocol, revisited.

suggestive of an Esterel implementation, just as the individual transitions in Fig. 2 are suggestive of gotos in a C program.

The transitions to and from the *breaking* state in Fig. 5 merit a comment, because they do not fit well with single-entry/single-exit control flow constructs. As an aside, from the *breaking* state, the terminal can query administrative date stored within the switch controller. From this state, the terminal can ask for an ongoing call to be continued or disconnected. In *Ftermp*, the *breaking* state is implemented by a module in parallel with the rest of the code.

3 A MODULE IN ESTEREL

This section examines a building block written in Esterel to collect an incoming string from the terminal. Incoming strings include passwords, destination names, and commands (in the *breaking* state). The discussion of the `getString` module introduces several of the Esterel constructs that we found useful for implementing terminal protocols.

Esterel: Introductory Remarks

Esterel is designed for implementing embedded systems, which interact with their environment through input and output signals. Conceptually, some input signals arrive, the system responds, and then settles down to wait for the next set of input signals. The system response is theoretically instantaneous — in practice, the underlying machine is assumed to be fast enough that the system settles down before the next set of input signals arrive. The instantaneous response is a consequence of the ‘‘synchrony hypothesis,’’ which makes Esterel programs deterministic. A key component of the Esterel system is a compiler, which translates Esterel programs into deterministic automata. As a final introductory remark, Esterel has no built-in notion of clock time. Applications that need to keep track of time must do so by interacting with an external timer, as we shall see.

Signals and Control Flow

The complete Esterel program in Fig. 6 consists of a single module `getString`. An input signal `DATAIN` is declared on line 2. The type `string`, within parentheses, refers to the type of the values carried by `DATAIN`. The executable statements of the program are in a bold font. The program waits for signal `DATAIN` to arrive. It then assigns the value carried by the signal, denoted by `?DATAIN`, to the variable `aString`. The program then terminates.

Now suppose that the string arrives in pieces. The bold statements in Fig. 7 use a variable `aString` to accumulate an incoming string. Esterel concentrates on the sequencing aspects of reactive systems. A *host* programming language, say C, is needed to handle all other aspects of a computation. The following syntax is used to call host language procedures from an Esterel program:

`call` ⟨*procedure-name*⟩ `(`⟨*reference-parameters*⟩`)` `(`⟨*value-parameters*⟩`)`

In Fig. 7, procedure `build` is called with a single value parameter `?DATAIN` and two reference parameters `aString` and `done`. The declaration of the boolean `done` is not shown. In addition to accumulating the incoming string in `aString`, procedure `build` uses the variable `done` to indicate whether a complete string has been collected. In our application, a newline marks the end of an incoming string.

An exit statement must be enclosed within a trap statement. Execution of

`exit` ⟨*trap-name*⟩

send control to the end of the enclosing trap named ⟨*trap-name*⟩. The exit statement in Fig. 7 sends control to the end of the trap statement named `GET_STRING`; in this case, to the end of the program.

```
module getString:
input DATAIN(string);

var aString : string in
   await DATAIN;
   aString := ?DATAIN
end
.
```

Fig. 6. A complete Esterel program.

```
trap GET_STRING in
  aString := "";
  loop
    await DATAIN;
    call build(aString,done)(?DATAIN);
    if done then
      exit GET_STRING
    end
  end
end
```

→ Esterel compiler →

```
State 0
aString := "";
nextstate 1

State 1
if DATAIN then
  build(aString,done)(?DATAIN);
  if done then nextstate 2 end;
  nextstate 1
end;
nextstate 1

State 2
```

Fig. 7. The Esterel compiler translates source programs into automata. Here, the source program fragment accumulates an incoming string, using a procedure `build`. A pretty-printed version of the compiled automaton appears on the right.

```
trap GET_STRING in
   loop
      aString := "";
      do
         loop
            emit TIMER(N);
            await DATAIN;
            call build(aString, done)(?DATAIN);
            if done then exit GET_STRING end
         end
      watching ALARM timeout
         if aString <> "" then
            call wString()(WARNING)
         end
      end % do
   end % loop
end % trap
```

Fig. 8. Use of the `do-watching` construct.

Watchdogs: The `do-watching` Construct

The watchdog or `do-watching` construct sets Esterel apart from a language like C. The normal action of

do $\langle instructions \rangle_1$ **watching** $\langle signal \rangle$ **timeout** $\langle instructions \rangle_1$ **end**

is to execute $\langle instructions \rangle_1$ and terminate. If, however, $\langle signal \rangle$ arrives in the interim, then control is taken away from $\langle instructions \rangle_1$, and given to $\langle instructions \rangle_2$.

The semantics of Esterel are carefully defined to ensure determinacy. Only at certain program points can control be taken away from an instruction.

Keyword `timeout` simply refers to the occurrence of $\langle signal \rangle$ before $\langle instructions \rangle_1$ terminates. ‘‘Time’’ in Esterel refers to the relative occurrence of signals and has no connection with clock time.

The watchdog in Fig. 8 watches for `ALARM` while the enclosed loop collects an incoming string. The loop body emits a signal `TIMER`, which asks the environment (TDK) to start a clock timer. The environment emits an `ALARM` if the clock timer goes off. TDK resets the timer whenever a signal arrives. Thus, arrival of `DATAIN` automatically resets the timer. If `ALARM` arrives before a complete string is collected in the variable `aString`, procedure `wstring` is called to warn that `aString` is being emptied.

Communication between Parallel Statements

Esterel has a parallel construct, which is compiled away. The source code in Fig. 9 has the statement

every BREAK do `call cycleBaud()(); emit RESTART` **end**

in parallel with a statement to collect an incoming string. The procedure `cycleBaud` changes the incoming baud rate from the terminal. The code for the automaton, however, is sequential. Paralellism in the source has been eliminated by interleaving the code from the two parallel branches.

A statement of the form

`signal RESTART in` $\langle instructions \rangle$ `end`

declares `RESTART` to be a signal local to $\langle instructions \rangle$. Local signals are compiled away.

In words, one parallel branch in Fig. 9 waits for signal `BREAK`. If this signal arrives, it changes the baud rate and emits the local signal `RESTART`. The other parallel branch has the form

```
loop ⟨instructions⟩ each RESTART
```

This statement corresponds to a looping watchdog; each arrival of RESTART terminates the enclosed ⟨*instructions*⟩ and sends control back to the beginning of the loop. Thus, upon RESTART, collection of the incoming string is interrupted.

Parallelism that can be compiled away is very useful for partitioning code. A separate parallel branch was used in Fig. 9 to wait for BREAK because the subprotocol for changing the baud rate is hardware device dependent, whereas the collection of an incoming string is independent of the hardware.

4 EXPERIENCE WITH ESTEREL

Modules, watchdogs, and source-level parallelism help to make the Esterel versions of the terminal protocol readable and manageable, in our opinion. Examples illustrating the benefits of watchdogs and parallelism have already appeared in Section 3, so this section contains some minor criticism of the language. Overall, Esterel is a very promising language.

Emits are Unordered

Esterel programs are assumed to respond instantaneously to a set of input signals. Signals emitted during an instant are assumed to occur together, so their order is not significant. The following sequence of emit statements is a reaction to an OFFHK input signal:

```
emit AOFFHK;
emit TALK;
```

The desired behavior is that the AOFFHK acknowledgement and the TALK signal will be emitted in sequential order as they appear in the source code. The language does not define the output signal order, however,

```
trap GET_STRING in
  signal RESTART in
    every BREAK do
      call cycleBaud()();
      emit RESTART
    end
  ||
    loop
      aString := "";
      do
        loop
          emit TIMER(N);
          await DATAIN;
          call build(···)(···);
          if done then
            exit GET_STRING
          end
        end
      watching ALARM
    each RESTART
  end % signal
end % trap
```

(a) Parallel modules.

```
State 0
aString := ""; TIMER(N);
nextstate 1
State 1
if BREAK then
   cycleBaud()();
   aString := ""; TIMER(N);
   nextstate 1
end;
if ALARM then nextstate 2 end;
if DATAIN then
   build(···)(···);
   if done then nextstate 3 end;
   TIMER(N);
   nextstate 1
end;
nextstate 1
State 2
if BREAK then
   cycleBaud()();
   aString := ""; TIMER(N);
   nextstate 1
end;
nextstate 2
```

(b) Compiled automaton, pretty-printed.

Fig. 9. Parallel source code, sequential automaton.

and the compiled code reversed their order. When `TALK` was emitted first, the hardware module responded by repeating `OFFHK` continuously (while it looked for an immediate `AOFFHK`).

The solution is to define a new signal which encapsulates a sequence of output signals. The order is specified in the host language interface function for the new signal. However, there may be sequencing requirements involving emissions and procedure calls. In this case, the emit statement should be changed to a procedure call. *Etermp* replaces the two emit statements by a sequence of two procedure calls.

The Host Language Interface

Only local variables are permitted in Esterel. There are no global variables even for submodules. This preserves the rigorous semantics for control flow in Esterel, but it complicates the interface with global data such as system data and hardware registers. One solution is to use signals as global variables. However, this results in duplicate data and is more likely to generate signal collisions. Signals used for control flow can cause causality cycles, but signals used for global variables should not be involved in control flow. Another solution is to use constants to read global variables and to write them using procedure calls.

Esterel provides only primitive types and type name definitions. All other type manipulation, such as access to structure elements and bit manipulation, must be done in the host language by encapsulating them in function or procedure calls. An alternative is to define Esterel constants that are actually expanded into C code expressions via `#define` macros in the C interface.

Language Layers

Some secondary language related effects are as follows. The separation of code into Esterel and host-language layers exacts a penalty. Definitions must be repeated — both a C definition and an Esterel definition are required for each constant, function, procedure, and signal in the program interface. Signals often have values, only easily accessible in the Esterel code, which are duplicated in the C code. The language layering forces the program to encapsulate even simple C statements into functions or procedures. The Esterel compiler produces C code with actions encapsulated in C functions. The combination results in highly nested function calls just to execute a simple C statement like an assignment. A partial solution is to define preprocessor macros for C functions. If C++ is used as a host language, then inline functions could reduce the levels of indirection at run time.

The procedure call syntax

`call` ⟨*procedure-name*⟩`(`⟨*reference-parameters*⟩`)(`⟨*value-parameters*⟩`)`

often forces an extra C function call just to re-arrange the parameters into the correct order for library or system calls. Perhaps an attribute keyword could be used to specify reference parameters, as in Pascal. This would also enable reference parameters for Esterel functions. Note that using C as a host language results in hidden reference parameters for arrays (the anomalous type in C, e.g. strings).

Other Concerns

The compiled code presently contains a lot of duplicated code. We could have used better error messages.

5 EXPERIENCE WITH TDK

Several challenges were encountered in the actual implementation of the Esterel versions, *Etermp* and *Ftermp*.

One good reason for using one large main function for *Otermp* is that the operating system in TDK does not provide read/write global data for each process. The C code generated by the Esterel compiler forces the use of many function calls. One solution is to pass modifiable objects as parameters to each C function. This is cumbersome and difficult to implement.

The Esterel compile code required global data. A little-known facility for allocating uninitialized global data (BSS) came in handy. The Esterel output and C interface code was processed to force all read/write data to BSS (by moving initialization to `main()`). The end result is that Esterel and C interface code can be programmed normally except that read/write data must be uninitialized at load time and explicitly initialized at run time.

The operating system in the controller provides a process suspension facility to allow processes to free up memory resources. When a process suspends, the process stack and the global memory are released. All state information saved in the stack and the global memory is lost. Only 4 very precious words are available to the programmer to save state information for all of the processing that has been completed. Fortunately, the current position in the Esterel automaton is available in a variable that can be saved in one of the 4 precious words. Other variables are explicitly re-initialized to a known value when the process is awakened after suspension.

Restructuring or respecifying the terminal protocol helps to reduce complexity. Most notably, some states can be merged to provide a cleaner design, simpler specification, and easier implementation. The *selecting* and *checking* states were combined. If the user selects a new security group, the password is always prompted for even if there is no password. The *autowait* state for predefined destinations waits for a timeout analogous to waiting for dial-string input from the user. The timeout is minimal for the first call attempt and jumps up to a long timeout after call failures. The *fixing* and *ending* states are co-located as exception handling for the inner part of the *Etermp* state machine.

6 DISCUSSION

The Esterel language enables the natural coding of reactive systems. Parallelism naturally expresses many reactions, and parallelism in Esterel is inexpensive at run-time. The do-watching statement clearly expresses watchdog constructs. The trap-handle statement provides powerful exception handling facilities while maintaining logical control flow.

In the original *Otermp*, each action must be explicitly coded in each state. This results in considerable duplication of code. In the Esterel versions, several signals have common actions that are expressed in one line of Esterel code. These default actions are mentioned once in the main Esterel module. The individual states encode only their primary actions.

7 ACKNOWLEDGMENTS

Initial exploration of terminal protocols began with the Steve Mahaney, with lots of discussions with Lee McMahon. The simplified version *Stermp* was done in October 1987 with the help of Gerard Berry. George Gonthier provided invaluable expertise on Esterel. Bill Marshall helped with the interface to the research controller. Ce-Kuen Shieh made the November 89 version *Ftermp* run in the switch.

8 REFERENCES

1. Berry, G., Couronné, P., and Gonthier, G. Synchronous programming in Esterel. Rapport Recherche 647, INRIA, 1987.

2. Fagen, M.D., Ed. *A History of Engineering and Science in the Bell System: The Early Years (1875-1925)*. AT&T Bell Laboratories, Available from AT&T Technologies, Indianapolis (1-800-432-6600), 1975.

3. Fraser, A.G. Towards a universal data transport system. *IEEE J. Selected Areas in Communications SAC-1*, 5 (1983), 803-816.

4. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*, 2nd Ed. Prentice-Hall, Englewood Cliffs, N. J, 1988.

5. McMahon, L.E. An experimental software organization for a laboratory data switch. In *ICC '81, IEEE Intl. Conference on Communications*, Vol. 2, 1981, pp. 25.4.1-25.4.4.