Safety Property Verification of ESTEREL Programs and Applications to Telecommunications Software

Lalita Jategaonkar Jagadeesan
Carlos Puchol
James Von Olnhausen

July 1995

In *Proceedings of the 7th International Conference on ComputerAided Verification*, Volume 939 of the Lecture Notes in Computer Science,pages 127-140, July 1995.

Safety Property Verification of ESTEREL Programs and Applications to Telecommunications Software

(Appeared in the Proceedings of Conference on Computer-Aided Verification
LNCS Volume 939, pages 127–140, 1995)

Lalita Jategaonkar Jagadeesan[1], Carlos Puchol[2⋆], and James E. Von Olnhausen[3]

[1] Software Production Research Dept., AT&T Bell Laboratories, Naperville, IL 60566 (USA)
[2] Dept. of Computer Sciences, The University of Texas at Austin, Austin, TX 78712 (USA)
[3] Global Software Platform Lab, AT&T Bell Laboratories, Naperville, IL 60566 (USA)

Abstract. We present a technique for automatically verifying linear-time temporal logic safety properties of programs written in ESTEREL, a formally-defined language for programming reactive systems. In our approach, linear-time temporal logic safety properties are first translated into ESTEREL programs that model these properties. Using the ESTEREL compiler, the translations are compiled in parallel with the ESTEREL program to be verified. A trivial reachability analysis of the output of the compiler then indicates whether or not the safety property is satisfied by the program. We describe two real-world software problems — ESTEREL versions of two features of the AT&T 5ESS® switching system — and one well-known benchmark problem — the generalized railroad crossing problem — that we have verified using our technique and associated tool set.

1 Introduction

The ESTEREL programming language[5] is a formally-defined, high-level language designed specifically for programming reactive systems. It is based on the "synchrony hypothesis," which states that every reaction of a system to a set of inputs is theoretically instantaneous. In practice, this amounts to requiring that the environment of the reactive system be invariant during every reaction. ESTEREL is especially well suited for programming real-time reactive systems that are embedded inside larger applications.

Safety properties are typically sufficient to describe most intended properties of real-time systems, since responses are required within bounded intervals. One of the more widely accepted languages for specifying temporal behavior and safety properties of reactive systems is linear-time temporal logic [16] . We present a technique for automatically verifying a large class of propositional linear-time temporal logic safety properties of ESTEREL programs. We then describe two real-world software problems and one benchmark problem that we have verified using our technique and associated tools. In particular, we have performed a case study [13] to assess the suitability of ESTEREL to switching software by writing ESTEREL versions of two features of the AT&T 5ESS® telephone switching system — a reactive real-time system which provides

⋆ The author is currently supported by a Fulbright fellowship. The work described here was performed while the author was visiting AT&T Bell Laboratories.

telecommunications services. As part of this assessment, we have used our verification tools to prove that our ESTEREL versions satisfy some safety properties required for the existing software. The executable code generated by our ESTEREL version of the more complex of these features has been successfully tested in the 5ESS switch environments; a detailed presentation appears in [13] . We have also verified an ESTEREL solution of the generalized railroad crossing problem [12] using our technique and tools.

Many traditional verification techniques allow properties to be proved only of models of programs rather than program texts themselves; these models are written by hand and require a detailed understanding of the program and the target language, and hence they may not faithfully represent the program. In contrast, we verify the *actual text of* ESTEREL *programs*, guaranteeing that the actual implementation satisfies the intended safety properties (as long as all the compilers involved are correct). This is in the spirit of Berry's WYPIWYE principle –what you prove is what you execute.

In our approach, given a temporal logic safety formula, we first recursively translate it into an ESTEREL program whose traces correspond to the computations that violate the safety formula. The program derived from the formula is then composed in parallel with the given ESTEREL program to be verified. The program resulting from this composition is compiled using commercially available ESTEREL tools [1] ; a trivial analysis of the output of the compiler then indicates whether or not the property is satisfied by the original program. By exhaustively generating all the reachable states of the composed program, the ESTEREL compiler in effect performs model checking [7] .

The foundations for this approach were first introduced in [21] and [20] in the context of temporal logic and finite-state automata. Specifically, given any propositional temporal formula, [21] introduced a procedure to build a finite automaton on infinite words that accepts precisely the sequences that satisfy the formula. In [20] , this result was used to improve a procedure for model checking ([14]) by combining the automata for the program and the formula and checking the language of the resulting automaton for emptiness.

Our work was originally inspired by the work of Halbwachs et al. [6, 9, 10, 18] , which develops a technique for verifying safety properties for programs written in the synchronous language Lustre [8] . Ours is a similar approach, but differs (aside from the fact that it is based on ESTEREL) in that we allow the specification of the properties directly in temporal logic, we make use of system resources such as operating system timers and we provide support for bounded response properties (without blowup in the bound). The current verifier supported in the native ESTEREL environment [1] is based on bisimulation reduction and does not support temporal logic properties.

Our paper is organized as follows. Section 2 describes the class of temporal logic safety properties currently supported by our approach, and a brief introduction to ESTEREL is given in Section 3. Section 4 presents our translation from this class of safety properties into ESTEREL. Our verification technique and its proof of correctness is given in Section 5. Optimizations for response operators are described in Section 6.

We then describe case studies in using our verification technique on three problems. Section 7 gives an overview of some switching software, and Section 8 and Section 9 describe the verification of our ESTEREL versions of two features of the AT&T 5ESS switching system. The verification of an ESTEREL implementation of the generalized railroad crossing problem is described in Section 10. Our conclusions appear in Section 11.

2 Temporal Logic Safety Formulas

ESTEREL does not provide constructs for referencing future states of the program, hence past-tense operators provide a good choice for implementation. Most temporal logics typically contain only future-tense operators, however, past-tense operators are considered to make the formulation of properties more modular as well as natural and convenient [15] . We have found past-tense operators to be as convenient as the future-tense operators in proving properties of actual systems. In our approach, we consider the class of *canonical* safety formulas [16] , further restricted so that the state formulas consist only of signal identifiers; in particular, we do not allow variables to occur in state formulas. Formally,

Definition 1. Let Sig be a fixed finite alphabet of signal names, containing the constants TRUE and FIRST. $\mathcal{PAST}$ is defined to be Sig, closed under the boolean operators $\neg$, $\vee$, $\wedge$, and $\rightarrow$ and the linear temporal logic past operators $\ominus$, $\mathcal{S}$, $\boxminus$, $\diamondminus$, and $\mathcal{B}$ (previous, since, has-always-been, once and back-to). $\mathcal{SAFE}$ is defined to be the set of formulas of the form $\Box\, p$, where $p \in \mathcal{PAST}$ and $\Box$ is the "always" operator of linear temporal logic. We use p, q to range over $\mathcal{PAST}$ and s to range over $\mathcal{SAFE}$.

The definition of satisfaction of a formula in $\mathcal{PAST}$ and $\mathcal{SAFE}$ is standard, thus not repeated here. $\mathcal{PAST}$ is the class of past-tense temporal logic formulas and $\mathcal{SAFE}$ is the class of safety formulas, which can be characterized as follows [16] : a formula s is a safety formula iff any sequence σ violating s (i.e., satisfying $\neg s$) contains a prefix all whose infinite extensions violate s. Informally these formulas stipulate that "something bad never happens."

In addition to these standard safety formulas we have found it useful in practice to consider bounded versions of response formulas as well. A canonical *response* formula is a formula of the form $\Box\ (p \rightarrow \Diamond q)$[4]. This class of formulas states that for all computations, every position where p holds is followed by or coincides with a position where q holds. However, no guaranteed bounds for obtaining q as response to p are expressed. In real-time systems the interest lies fundamentally in the ability to limit the delay of responses to stimuli, thus stronger formulas than these are needed. Adding timing constraints to temporal operators is a standard way to capture real-time requirements. Standard *bounded-response* formulas [3, 2] [5] are of the form $\Box\ (p \rightarrow \Diamond_d q)$ which state that every p-position is followed by a q-position within d reactions. Intuitively, this class of formulas are in fact safety formulas; they operate in an "interval" in which if the formula is violated, all extensions of it will be violated. We define two past-tense versions of the bounded response formulas, the *bounded response* (1) and the *bounded ensures* (2) operators, which can in turn be defined in terms of the standard

[4] The alternative form $\Box\Diamond p$ may be more familiar.

[5] We do not currently consider interval operations such as $p \rightarrow \diamondminus_{[a,b]} q$.

past operators as follows:

$$p \rightarrow \diamondminus_d q \equiv (\ominus^d p \rightarrow \bigvee_{k=0}^{d} \ominus^k q) \quad (1) \qquad \text{and} \qquad p \rightsquigarrow \diamondminus_d q \equiv (\bigwedge_{k=0}^{d-1} \ominus^k p) \rightarrow q \quad (2),$$

where $\ominus^0 f \equiv f$ and $\ominus^n f \equiv \ominus(\ominus^{n-1} f)$ for a past temporal logic formula f.

3 The ESTEREL programming language

ESTEREL [5] is a language, with a precisely defined formal semantics, for programming the class of deterministic reactive systems that wait for a set of possibly simultaneous inputs, react to the inputs by computing and producing outputs, and then quiesce, waiting for new inputs. Since ESTEREL is based on the "synchrony hypothesis," every reaction to a set of inputs is considered to be instantaneous. The programming model in ESTEREL is the specification of components, or modules, that run in parallel. Modules communicate with each other and the outside world through *signals*, which are broadcast and may carry values of arbitrary types. Consistent with the synchrony hypothesis, the emission and reception of signals is considered to be instantaneous.

ESTEREL allows only deterministic behaviors to be specified: the inputs to every reaction (and the current values of variables) fully determine the outputs emitted in that reaction as well as the input-output behavior of the rest of the program. Along with the synchrony hypothesis, both communication and pre-emption preserve determinism. Furthermore, all internal communication is compiled away, and a single deterministic finite state machine is generated by the compiler. Thus, the parallelism in ESTEREL is a structuring tool for programming convenience, and does not incur any run-time overhead — the compiler automatically performs the complex interleaving between parallel modules. Since this implementation is a finite state machine, an added benefit is that the maximum amount of time taken by any reaction can be accurately bounded if the execution times of the transitions are known.

4 Structural Translation

Our approach is based on recursively translating safety formulas in $\mathcal{SAFE}$ into ESTEREL programs. The resulting ESTEREL program can be viewed as a *finite state acceptor* of the computations satisfying the formula. Namely, it emits a special ESTEREL signal SAT_s in exactly those reactions satisfying the formula s.

The program derived from s is then composed in parallel with the original ESTEREL program to be verified. The ESTEREL program resulting from this composition is then translated into a single finite-state automaton by the ESTEREL compiler [1]. The ESTEREL compiler in effect performs model checking *as it compiles*, by exhaustively generating all the reachable states. In particular, the process of compilation computes the value of all subformulas of the formula along with the program itself. The verification problem is simply reduced to the problem of checking that s is satisfied in all states of this automaton; namely, that the signal SAT_s is emitted in every state of this automaton.

For practical reasons, the ESTEREL code we produce emits a signal VIOLATED_s in every state that the safety formula is *not* satisfied. The checking algorithm searchs for a state where VIOLATED_s is emitted, returning successfully if no such state exists, or returning a computation of the program that violates s otherwise. Hence the checking algorithm is very simple and very efficient in practice.

For pure ESTEREL programs, namely those which do not use variables or valued signals and whose control structure is thus fully determined at compile-time, this verification technique is both sound and complete. However, for programs whose control structure does depend on run-time values, our technique is sound but not complete because the ESTEREL compiler considers all paths, including those that are impossible due to data values. Thus if a program violates a property, a computation leading to the state violating it will always be found by our tools, but our procedure may incorrectly indicate that some otherwise correct programs can violate a property.

An advantage of ESTEREL that carries into the verification technique is that it is not necessary to model the environment under which the program is verified. The verification is done over all possible environments (the "relation" feature in ESTEREL that restricts the set of possible inputs at every time instant is enough for most practical purposes).

4.1 Translation of Past Formulas

We define a recursive transformation $\mathcal{E}_p$ which given a past formula p, produces an ESTEREL *program fragment* that emits a signal SAT_p in exactly the reactions that p holds true. It is clear from the definition below that the generation of $\mathcal{E}_p$ is linear (in time and space) with respect to the size of p. In the following definitions, let p and q denote past formulas. The base case below is the translation of plain signals; for any signal $\mathcal{S} \in$ Sig, there is an associated signal SAT_$\mathcal{S}$ which is present/absent exactly at those time instants $\mathcal{S}$ is.

Constants

The translation of TRUE and FIRST are very simple: $\mathcal{E}_{\mathrm{TRUE}} = \texttt{sustain SAT\_TRUE}$ and $\mathcal{E}_{\mathrm{FIRST}} = \texttt{emit SAT\_FIRST}$.

Boolean Operators

The transformation for boolean operators is straightforward to define. We present here the translation of a formula involving the $\vee$ operator. The translation for each of the $\wedge$, $\rightarrow$ and $\neg$ operators is similarly defined.

$\mathcal{E}_{p \vee q} = \mathcal{E}_p \;||\; \mathcal{E}_q \;||$
```
[
  every immediate tick do
    present [SAT_p or SAT_q]
      then emit SAT_(p ∨ q)
    end present
  end every
]
```

Past Operators

Below we illustrate the translation for the $\ominus$ and $\mathcal{S}$ operators. The translation for $\boxminus$, $\diamondminus$ and $\mathcal{B}$ can either be defined in terms of these operators ($\mathcal{E}_{\diamondminus p} = \mathcal{E}_{\mathrm{TRUE}\ \mathcal{S}\ p}$, $\mathcal{E}_{\boxminus p} = \mathcal{E}_{\neg \diamondminus \neg p}$ and $\mathcal{E}_{p\ \mathcal{B}\ q} = \mathcal{E}_{(p\ \mathcal{S}\ q) \vee \boxminus p}$), or with more direct translations to ESTEREL, omitted here due to space limitations. The direct translation is somewhat more efficient, and hence has been incorporated into our tools.

$$\mathcal{E}_{\ominus p} = \mathcal{E}_p \,||\, [$$

```
loop
  present SAT_p then
    await tick;
    emit SAT_⊖p
  else await tick
  end present
end loop
]
```

$$\mathcal{E}_{p \,\mathcal{S}\, q} = \mathcal{E}_p \,||\, \mathcal{E}_q \,||\, [$$

```
every immediate SAT_q do
  do
    sustain SAT_(p S q)
  watching [not SAT_p]
end every
]
```

Bounded-Response and Bounded-Ensures Operators
Since the time constrained operators can be defined in terms of the above operators, they can be translated by expanding their definition and then applying the transformations above. While this translation via standard past formulas is simple, it generates an ESTEREL program fragment whose state space is at best quadratic in the bound d. This renders such translation futile for many practical purposes; we briefly describe in Section 6 some optimizations that make translations of these operators linear in d, hence making them more useful in practice.

4.2 Translation of Safety Formulas

We now define the translation for the safety properties. This translation introduces the signal denoting the violation of the property.

Definition 2. Let $s = \Box\, p$ be a canonical formula of $\mathcal{SAFE}$. Let I_p be the set of signal names occurring in p minus {FIRST, TRUE}, and let L_p be the set of signals in $\mathcal{E}_p$ minus I_p — these are local signals. Then the translation $\mathcal{E}_s$, which emits the signal VIOLATED_s as soon as the safety formula s is violated, is defined as follows:

$$\mathcal{E}_s =$$

```
module Safety:
  input I_p;
  output VIOLATED_s;
  signal L_p in
    E_p || [
           await immediate [not SAT_p];
           emit VIOLATED_s
         ]
  end signal
end module
```

5 Properties of the Translation

We now prove the correctness of our translation. We first need to do some minor syntactic adjustments to conform to the modular structure of ESTEREL.

Definition 3. Let $\mathcal{E}_p$ be the ESTEREL program fragment associated with $p \in \mathcal{PAST}$, let I_p be the set of signal names that occur in p, and let O_p be the set of signal names that occur in $\mathcal{E}_p$ but not in p. Then $\mathcal{PROG}$ of p is defined in Figure 1(a).

$$\mathcal{PROG}(p) = \begin{array}{l}\texttt{module PROG\_p:}\\ \quad\texttt{input } \texttt{I\_}p\texttt{;}\\ \quad\texttt{output } \texttt{O\_}p\texttt{;}\\ \quad\;\mathcal{E}_p\\ \texttt{end module}\end{array}$$

(a)

$$\mathcal{VERIF}(E,s) = \begin{array}{l}\texttt{module Verify\_Es:}\\ \quad\texttt{input } I_E\texttt{;}\\ \quad\texttt{inputoutput } O_E\texttt{;}\\ \quad\texttt{output VIOLATED\_}s\texttt{;}\\ \qquad\texttt{run } E \texttt{ || run } \mathcal{E}_s\\ \texttt{end module}\end{array}$$

(b)

Fig. 1. Definitions for the top level translation of formulas in $\mathcal{SAFE}$.

Lemma 4. *Let I_p be the set of signal names that occur in $p \in \mathcal{PAST}$, minus* $\{$FIRST, TRUE$\}$*, and let O_p be the set of signal names that occur in $\mathcal{E}_p$ but not in p. Let $\sigma = I_1 \dots I_k$ be a sequence over the powerset of I_p, and let $\gamma = O_1 \dots O_k$ be the unique sequence over the powerset of O_p produced by $\mathcal{PROG}(p)$ in reaction to σ. Then for any j with $1 \leq j \leq k$, O_j contains the signal* SAT_p *iff $(\sigma, j) \models p$.*

Definition 5. Let $\mathcal{E}_s$ be the ESTEREL program fragment associated with $s \in \mathcal{SAFE}$ and let I_s be the set of signal names that appear in s, minus $\{$FIRST, TRUE$\}$. Furthermore let E be an ESTEREL program without causality cycles and let I_E and O_E respectively be the sets of input and output signals of E with $I_E \cup O_E \supseteq I_s$[6]. We define $\mathcal{VERIF}$ as in Figure 1(b).

Lemma 6. *Let $s \in \mathcal{SAFE}$ and let E be an* ESTEREL *program without causality cycles. Then the program $\mathcal{VERIF}(E, s)$ does not have causality cycles in it.*

The proof for Lemma 4 follows from an induction on the structure of safety formulas. Lemma 6 follows easily from the definition of causality cycles [5]. Theorem 7 now gives a sound and complete verification technique for pure ESTEREL programs, namely those without `if` statements. The control structure of such programs is fully determined at compile-time.

Theorem 7. *Let $s \in \mathcal{SAFE}$ and let E be an* ESTEREL *program without causality cycles and no* `if` *statements. Then* VIOLATED_s *will be emitted in some state of the program $\mathcal{VERIF}(E, s)$ iff E does not satisfy s.*

The theorem follows easily from Lemma 4, the above definitions and the semantics of ESTEREL. We now show that our technique is sound but not complete for ESTEREL programs containing `if` statements. We have found that this is not a significant difficulty in practice, since the interesting values of variables can be typically indicated finitely through signal emissions when needed.

Theorem 8. *Let $s \in \mathcal{SAFE}$ and let E be an* ESTEREL *program without causality cycles, but possibly containing* `if` *statements. If E does not satisfy s, then there is some state of the program $\mathcal{VERIF}(E, s)$ that emits the output signal* VIOLATED_s.

[6] We disallow s from containing references to signals not included in the input or output sets of E for stylistic reasons. If s reads signals that E ignores, without loss of generality we require that they appear in the input set of E.

The proof is analogous to that of Theorem 7. We emphasize, however, that the converse of this theorem does not hold. Let E contain the following code, where B is the only piece that causes the property s to be violated.

```
var x:int in
    x:=1;
    if x = 1 then A
             else B
    end if
end var
```

In this scenario, fragment B can never be executed; however, since the ESTEREL compiler does not evaluate expressions, the state machine generated after compiling the translation appears to potentially emit VIOLATED_s.

6 Optimized Translation

This section discusses alternative translations for the bounded-response and the bounded-ensures formulas. While the translation via standard past formulas presented in Section 2 is simple, it generates an ESTEREL program fragment whose state space is quadratic in the bound d. The new translations we propose, on the other hand, are more complex, but yield ESTEREL program fragments whose state space is linear in the bound d: hence these have been incorporated into our tools. Due to space limitations, we omit the actual translations for these operators and only describe two interesting properties of the optimized translations.

Lemma 9. *There exists an optimized translation such that the state space of $\mathcal{PROG}(p)$ for any bounded-ensures or bounded-response formula p with bound d is $O(d)$.*

We note that Lemma 4 holds for the new definition of the bounded-ensures operator (we can prove this by using the verification technique itself!). However, our optimized translation for the bounded-response operator satisfies the following weaker property:

Lemma 10. *Let $\mathcal{S}$ denote the signal* VIOLATED_s *in $\mathcal{E}_s$, for a bounded-response formula s with the standard translation for s. Let $\mathcal{S}'$ denote the* VIOLATED_s *signal in $\mathcal{E}'_s$ with the alternate translation for s. Then the formula $\Box\,(\diamondminus\mathcal{S} \leftrightarrow \diamondminus\mathcal{S}')$ is satisfied by the parallel composition of both translations.*

Lemma 10 states that "once $\mathcal{S}$ *iff* once $\mathcal{S}'$ " or in other words, once s is violated, both translations capture it. However, the more powerful Lemma 4 does not hold for the optimized translation. As a consequence, under the optimized translation, Theorems 7 and 8 hold only for those formulas in which no bounded-response formula appears as a proper subformula. For other formulas, the standard translation must be used (through an option in the tool set).

This completes the presentation of our technique. The remainder of the paper describes the use of the technique to two problems from industry and a benchmark problem.

7 An Overview of Our Case Studies for Switching Software

We have performed a case study [13] to assess the suitability of ESTEREL to switching software by writing ESTEREL versions of two features in the AT&T 5ESS telephone switching system [17] . We then used our technique and tools to verify that our ESTEREL versions satisfy some safety properties required for the existing switch software. The descriptions of our case studies appear in the following two sections.

The 5ESS switch is a reactive, real-time system which provides telecommunications services. Long-distance telephone calls are typically connected through a network of hardware, referred to as *carrier groups*. Telephones are connected to switches via *lines*. Inputs to the switch include requests for placing and disconnecting telephone calls, requests for call forwarding and other calling features, as well status changes — such as malfunctions or recoveries from malfunctions — on carriers and lines [11] . In response to these inputs, the switch connects or disconnects calls, activates calling features, or in the case of malfunctions/recoveries, removes/restores the associated carriers and lines and routes new calls over functioning carriers and lines. The switch software is quite large and complex, consisting of several million lines of code; many subsystems embedded in the switch can themselves be regarded as reactive systems that communicate with one another and with the outside world.

8 Case Study: Carrier Group Alarm Software

In switches, a wide variety of carrier group types are used to transmit data corresponding to end-to-end telephone connections. These carrier groups are attached to various hardware units on a distributed set of processors, which are responsible for routing telephone calls. Malfunctions on these carrier groups, such as lost framing, lost signals, or physical accidents, can result in disturbance or abrupt termination of existing phone calls. The *Carrier Group Alarms (CGA)* software in the 5ESS switch is responsible for reporting status changes — malfunctions or recoveries from malfunctions — on carrier groups, so that other 5ESS software can respectively remove or restore the associated carrier groups from service, and route new telephone calls accordingly [11] .

One of the main sources of inputs to the CGA software are "summary-requests" from higher-level entities. When a *summary request* is received, either from a human operator or from some other part of the switch, the CGA software must collect data about the status of all the carriers on all the relevant processors, and print this information on various consoles and printers via the *Human-Machine Interface (HMI)*. The corresponding portion of the CGA software can be considered a reactive system whose inputs are summary requests and status change data about carriers, and whose outputs are requests to the relevant processors for data, and outputs of relevant status information via the HMI. For the purposes of our case study, we refer to this software as the "CGA Collection Software." In our ESTEREL version, there are other pieces of software, meant to reside on the processors to which carrier groups are attached; these pieces of software are responsible for sending the status data about the attached carrier groups to the CGA Collection Software. We do not discuss this software here due to space limitations.

As part of a case study to assess the suitability of ESTEREL to switching software, we have written an ESTEREL version of the summary request functionality of the Carrier

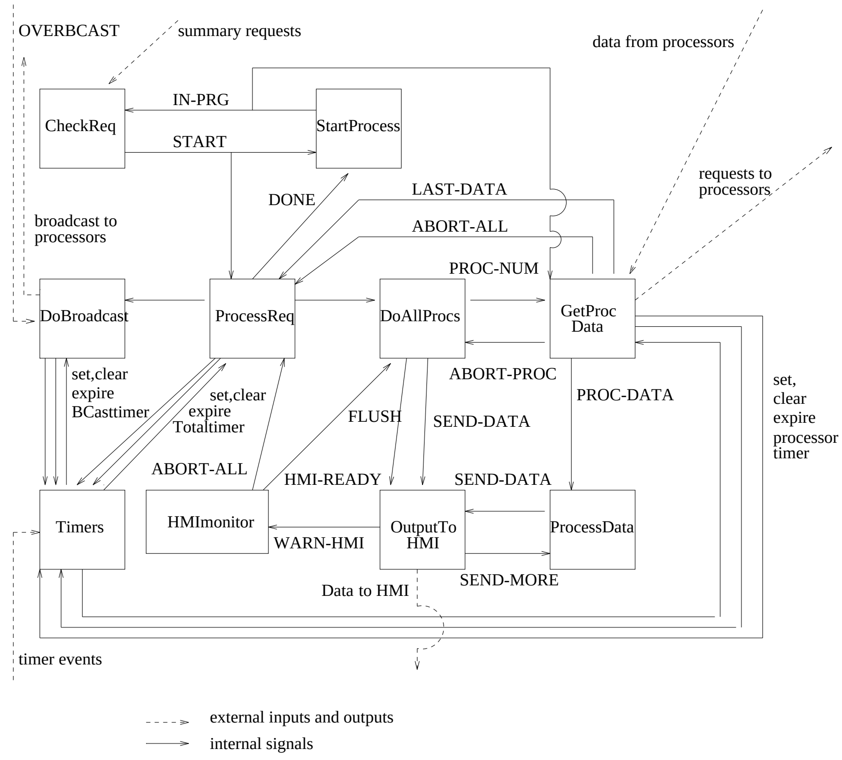

Fig. 2. Simplified Design of ESTEREL Version of the Carrier Group Alarms Software

Group Alarms software. We emphasize that the descriptions of our ESTEREL version given in this paper do not reflect the actual 5ESS switch software. Figure 2 illustrates our design of the CGA Collection Software. All modules (denoted by boxes in the figure) are composed by the ESTEREL parallel operator. Arrows emanating from modules indicate signals that are emitted by those modules; arrows pointing to modules indicate signals that are received by those modules. We note however that all these signals are broadcast and hence can actually be observed by all the modules; for clarity, our communication graph contains arrows only to those modules that actually respond to the corresponding signal. Our ESTEREL program is simply an implementation of each of these modules, and is described in more detail in [13]. Figure 2 is actually a simplified picture of our ESTEREL version. For instance, in our version three different kinds of summary requests (automatic periodic, manual and all-alarm) may be received; also, the rate and the queue in the HMI where information is sent differs depending on the type and kind of message, thus there are actually two different `HMI_READY` signals, one for each queue.

Our version uses several operating system timers; these correspond to the maximum time allotted to the processing of any given request, the maximum time spent waiting for a given processor to respond, and the period and maximum time spent on waiting for the HMI to respond. These timers are set and cleared by our ESTEREL implementation;

upon expiration of these timers, corresponding input signals are sent to the ESTEREL program by the operating system.

Our ESTEREL version consists of approximately 1000 lines of ESTEREL code. The generated state machine has approximately 200 states if *HMImonitor* is reduced to continually emit both kinds of HMI_READY signals, approximately 450 states if the *HMImonitor* only models one queue, and approximately 1500 states with the fully refined *HMImonitor* module. In most state machine based approaches, each state encodes a single, relatively small decision, there are many internal states, and realistic applications typically involve very large numbers of states. In contrast, in an ESTEREL-generated state machine, each state may encode a significantly large decision diagram representing all the internal computation performed in a reaction; consequently, the total number of states is substantially smaller. The corresponding state machine in typical approaches would have size $O(nm)$, where n is the number of states in the ESTEREL-generated state machine, and m is the maximum number of "internal states" in a decision diagram comprising an individual state. We estimate that on average, each ESTEREL-generated state consists of approximately 67 "internal states"; hence, in a conventional approach, our most complex implementation would consist of approximately 100,000 states.

8.1 Safety Properties for the Carrier Group Alarms Software

In order to verify our ESTEREL version of the CGA software, we obtained some of the safety properties that our ESTEREL version must satisfy in order to meet the requirements. The actual timing constants have been omitted here due to proprietary considerations, and have been denoted by symbols c_i. The symbol T_{c_i} represents the number of ESTEREL time instants corresponding to the time interval c_i.

- T0 A summary request must be completed in less than time c_1.
- T1 If a queried processor does not reply within time c_2, the request should be aborted immediately and the next processor should be queried.
- T2 If the HMI blocks on a message, the collection of new CGA data must suspend.
- T3 If the HMI blocks on a message, the message should be resent with a period of time c_3, until the HMI unblocks. If time c_4 elapses and the HMI has not yet unblocked, the summary request should be aborted.
- T4 If HMI unblocks after CGA data collection has been suspended, CGA data collection must be reactivated immediately.
- T5 No summary request should be honored when another summary request is currently running.

8.2 Verification of the Carrier Group Alarms Software in ESTEREL

Using our verification technique and associated tools, we have formally verified that our ESTEREL implementation satisfies the temporal logic version of the above safety properties. We note that since ESTEREL version used operating system timers to enforce timing constraints, our program can only be expected to satisfy the above properties under certain obvious assumptions about these operating system timers. In particular, we need to assume that when a timer is set with the value c_i, it either expires or is cleared within time c_i after it is set.

We give a sketch of such a proof of property **T0** below; the other properties are proved similarly. Ideally, to prove property **T0**, we would like to use our tools to automatically verify that

$$\Box\ \text{TT} \rightarrow (\text{GOT\_COMMAND} \rightarrow \diamondminus_{T_{c_1}+2} \text{WAITING\_FOR\_COMMAND}),$$

where $\text{TT} = \text{TOTAL\_TIMER} \rightarrow \diamondminus_{T_{c_1}} (\text{TOTAL\_TIMER\_EXPIRED} \vee \text{TOTAL\_TIMER\_CLEAR})$, $\text{GOT\_COMMAND} = \text{MANUALSTART} \vee \text{PERIODICSTART} \vee \text{ALLALMSTART}$, and

$$\text{WAITING\_FOR\_COMMAND} = \left\{ \begin{array}{l} (\text{MANUAL} \rightarrow \text{MANUALSTART}) \wedge \\ (\text{PERIODIC} \rightarrow \text{PERIODICSTART}) \wedge \\ (\text{ALLALM} \rightarrow \text{ALLALMSTART}) \end{array} \right\}$$

However, we found that because T_{c_1} is quite a large number, automatic verification of this property was not possible in practice. We thus took the alternate approach of proving that $\text{GOT\_COMMAND} \rightarrow \diamondminus_{T_{c_1}+2}\ \text{WAITING\_FOR\_COMMAND}$, under the assumption that $\Box\ \text{TT}$.

The proof was done in three steps: the first two of these steps used our verification tools, while the last step, done by hand, followed easily from our assumption on timers and simple formal reasoning.

Step 1 $\Box\ \text{GOT\_COMMAND} \rightarrow \diamondminus_1 (\text{TOTAL\_TIMER} \vee \text{TOTAL\_TIMER\_CLEAR})$: The TOTAL timer must be either set or cleared at most 1 time instant after an external command is accepted. We proved this property of our ESTEREL version of the CGA software using our verification technique and tool.

Step 2 $\Box\ (\text{TOTAL\_TIMER\_EXPIRED} \vee \text{TOTAL\_TIMER\_CLEAR}) \rightarrow \diamondminus_1\ \text{WAITING\_FOR\_COMMAND}$: The system returns to a state where external commands are accepted at most 1 time instants after the TOTAL timer has expired or cleared. We proved this property of our ESTEREL version of the CGA software using our tools.

Step 3 $\Box\ \text{GOT\_COMMAND} \rightarrow \diamondminus_{T_{c_1}+2} \text{WAITING\_FOR\_COMMAND}$: Done by hand; follows easily from the assumption on timers and transitivity of properties 1 and 2 above.

The proofs of the other safety properties are similar, and rely upon analogous assumptions on the bounds T_{c_i} and the expiration of timers. We now briefly describe the second telecommunication application and our solution to the benchmark problem.

9 Case Study: Automatic Protection Switching Software

Telecommunications systems need to use redundant communication channels so that spares are available in the event of failure. To ensure redundancy, a protocol called “Automatic Protection Switching” is often used. Under this protocol, whenever a line degrades or fails, a backup line, called a “protection line,” is used instead. The protocol ensures that the better quality line is always selected for communication, while responding to line quality changes and technician requests. A set of safety properties was drawn from a 5ESS application for automatic protection switching. We implemented a protocol in ESTEREL, and verified that our implementation satisfies these safety properties, using the technique and tools presented here. A more detailed presentation of the problem and our solution appears in [4] .

10 Case Study: Generalized Railroad Crossing

The generalized railroad crossing problem is a benchmark problem that has been recently proposed [12] to compare formal methods that exist for specifying, designing and analyzing real-time systems and to better understand their utility in the development of practical systems. It consists of a gate controlling a railroad intersection I within a region of interest R, where trains travel on multiple tracks in both directions. A sensor system determines when each train enters and exits R. Our solution involves choosing a model of gate and train behavior. Each train is modeled as follows. The presence of signal APPROACH_0 denotes train 0 entering R. We pick a fixed number of time units (IN_R is sustained), until it enters I. We then again pick a number of time units for the train to leave the region (IN_I sustained in that interval). We choose to model the gate as being in one of possible four states: UP, DOWN, GOING_UP, GOING_DOWN, and we pick fixed values for the time it takes to go from GOING_UP to UP and from GOING_DOWN to DOWN, commanded by the RAISE and LOWER signals, emitted by the controller. We have to assume that for the system to behave as specified, (our model of) the gate must take less time to lower than any train in the region can take to reach the intersection; the properties below do not hold otherwise. Our top-level solution is:

```
module GRC:
    input APPROACH_0, ... APPROACH_N;
    output UP, DOWN;
    signal RAISE, LOWER, GOING_UP, GOING_DOWN, IN_R, IN_I, EXIT_R in
        run Gate || run Controller
      ||
        run Train_0 || ... || run Train_N
      ||
        run Properties
    end signal
end module
```

The problem then consists of developing a system to control the crossing gate that ensures the satisfaction of the following properties, which we have formally verified with the tools and techniques shown in this paper (see [19] for details):

Safety Property: $\Box$ IN_I $\rightarrow$ DOWN. The gate is down if there is any train in the crossing.
Utility Property: $\Box\ \neg($IN_R $\vee$ IN_I$) \leadsto \ominus_5$UP. The gate is up when no train is in the crossing for some specified time (5 time units in this case).

11 Conclusions

We have presented a technique for automatically verifying safety properties of ESTEREL programs. This verification is performed using the ESTEREL compiler itself. An advantage of our technique is that verification is performed on the *actual text* of ESTEREL programs as opposed to models of programs. We argue that our technique is very useful in practice and demonstrate this through applications to a suite of problems, including ESTEREL case study versions of two features of the AT&T 5ESS telephone switching system.

Acknowledgments

We thank Patrice Godefroid and Kedar Namjoshi for comments on background and previous research on some of the ideas in this paper. We are grateful to Mark Ardis, Peter

Mataga, Mark Staskauskas, David Weiss and Mary Zajac for many useful discussions.

References

1. AGEL workshop manual version 3.0, 1989. Produced by ILOG.
2. R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *Proc. ACM Symp. on Principles of Distributed Computing*, 1991.
3. R. Alur and T. Henzinger. Time for logic. *ACM SIGACT News*, 22(3), 1991.
4. M. Ardis, John A. Chaves, L. Jagadeesan, P. Mataga, C. Puchol, M. Staskauskas, and J. Von Olnhausen. A framework for evaluating specification methods for reactive systems. In *Proc. 17th Intl. Conf. on Software Engineering*, April 1995.
5. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
6. A. Bouajjani, J.C. Fernandez, and N. Halbwachs. On the verification of safety properties, 1994. Draft.
7. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.
8. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79:1305–1320, 1991.
9. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *Transactions on Software Engineering*, 18(9):785–793, 1992.
10. N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.C. Glory. Specifying, programming and verifying reactive systems, using a synchronous declarative language. In *Workshop on Automatic Verification Methods for Finite State Systems, LNCS Vol. 407*, 1989.
11. G. Haugk, F.M. Lax, R.D. Royer, and J.R. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Tech. Journal*, 64(6 part 2):1385–1416, Jul-Aug 1985.
12. C. Heitmeyer, R.D. Jeffords, and B. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proc. 10th International Workshop on Real-Time Operating Systems and Software*, May 1993.
13. L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen. A formal approach to reactive systems software: A telecommunications application in ESTEREL. In *Proc. Workshop on Industrial-strength Formal Spec. Techniques*, April 1995.
14. O Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specifications. In *ACM Symposium on Priciples of Programming Languages*, pages 97–107, 1985.
15. O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Conference on Logics of Programs*, 1985.
16. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer-Verlag, 1992.
17. K.E. Martersteck and A.E. Spencer. Introduction to the 5ESS(TM) switching system. *AT&T Tech. Journal*, 64(6 part 2):1305–1314, Jul-Aug 1985.
18. D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multi-form time. In *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Techniques, LNCS Vol. 331*, 1988.
19. C. Puchol. A solution to the generalized railroad crossing problem in ESTEREL. Technical Report UTCS-TR95-05, Dept. of Com. Sci., Univ. of Texas at Austin, Feb 1995.
20. M.Y. Vardi and P. Wolper. An automata–theoretic approach to automatic program verification. In *Proc. LICS*, pages 332–339, 1986.

21. P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.