A Solution to the Generalized Railroad Crossing Problem in ESTEREL

Carlos Puchol
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188
cpg@cs.utexas.edu
http://www.cs.utexas.edu/users/cpg

Abstract

We present a solution to the Generalized Railroad Crossing benchmark problem based on the ESTEREL programming language. The solution is shown to satisfy the formal statements of the properties that the system requirements specify by using a verification method for safety properties of ESTEREL programs recently developed. The solution and verification presented have been developed within the synchronous system model, i.e. discrete time, global broadcast of events and instantaneous reactions.

Keywords: ESTEREL, reactive systems, synchronous systems, system verification, systems specification, formal methods.

1 The Generalized Railroad Crossing Problem

The Generalized Railroad Crossing (GRC) problem is a benchmark problem that has been recently proposed [4] 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. Informally, it consists of a gate controlling a railroad intersection I within a region of interest R, where trains travel through, on multiple tracks, in both directions. A sensor system determines when each train enters and exits R. The problem statement includes two properties that the system must satisfy at all times. The formal statement of the GRC problem is as follows:

> The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e., $I \subseteq R$. A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We define a set $\{\lambda_i\}$ of *occupancy intervals,* where each occupancy interval is a time interval during which one or more trains are in I. The ith occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the ith entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have any trains that entered the crossing after τ_i).
>
> Given two constants ξ_i and ξ_2, $\xi_1 > 0, \xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:
>
> **Safety Property:** $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$ (The gate is down during all occupancy intervals.)
>
> **Utility Property:** $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$ (The gate is up when no train is in the crossing.)

We present a solution to the GRC problem using the ESTEREL programming language, which is based on a synchronous execution model (see Appendix A for a brief outline of the synchronous model and the ESTEREL language itself). We have mechanically verified the two safety properties using the verification technique and associated tools presented in [5]. The solution comprises modeling "reasonable" behaviors for the gate and the trains; they are run in parallel to the actual implementation of the controller module.

This report is organized as follows. Section 2 presents the solution for the controller itself as well as describing each of the models for the rest of the system. Section 3 outlines the process of verification of safety properties of ESTEREL programs and Section 4 describes the formal verification process for the solution presented and a discussion of the issues faced to achieve it as well as the performance of the verification itself. We conclude with a discussion in Section 5.

2 The Solution in ESTEREL

The main ESTEREL module of the solution is shown below. It uses the parallel (synchronous) composition operator to run all the elements of the problem in parallel, namely, the gate, the trains and the controller:

```
module GRC:
        input APPROACH_0, ..., APPROACH_N;
        output UP, DOWN, IN_R, IN_I;
        signal RAISE, LOWER, GOING_UP, GOING_DOWN,
                ENTER_R ENTER_I, EXIT in
                run Gate
        ||
                run Track [signal APPROACH_0/APPROACH]
        ||
                ...
        ||
                run Track [signal APPROACH_N/APPROACH]
        ||
                run Controller
        end signal
end module
```

Our solution involves choosing a model of gate and train behaviors. This choice is essentially due to the fact that it is not possible to specify non-deterministic behaviors in ESTEREL programs other than by inclusion of external signals. Implementing more sophisticate models of the gate or trains is possible (e.g. letting the trains enter and leave the intersection in shorter times instead of fixed times) but we feel it is unnecessarily cumbersome, since they would not embody worse case scenarios.

Similarly, the number of trains that the solution comprises needs to be chosen at compile time — it is not possible to provide a solution with the number of trains not being fixed, due to the nature of both the ESTEREL programming language and the proof technique. In order to achieve proofs this general, techniques at other levels of abstraction would be necessary. More details of the different parts of the solution are shown in the subsections below.

2.1 Modeling the Trains

The presence of signal APPROACH_i denotes a train entering R in track i. This is an external signal which can arrive at any instant, thus modeling the non-deterministic nature of a train approaching on a track. The system is only responsive to this signal when there is no train in the corresponding track; the signal can be viewed as really modeling a track (thus the name). We pick a fixed number of time units (signal IN_R is sustained for 6 time units), until the train enters I. We then again pick a number of time units for the train to leave the region (signal IN_I sustained in an interval of 4 time units), then we emit the EXIT signal and go back to waiting for the approach signal to be present.

The IN_ and ENTER_ signals as well as the EXIT signal are local signals that are instantaneously broadcast by each of the trains (in ESTEREL one signal is present in a time instant as long as it is emitted at least once). The implementation of the train module is shown below:

```
module Track:
        input APPROACH;
        output ENTER_R, ENTER_I, EXIT, IN_R, IN_I;
        loop
                await immediate APPROACH;
                emit ENTER_R;
                emit IN_R; await tick;
                emit IN_R; await tick;
                emit IN_R; await tick;
                emit IN_R; await tick;
                emit IN_R; await tick;
                emit IN_R; await tick;
                emit ENTER_I;
                emit IN_I; await tick;
                emit IN_I; await tick;
                emit IN_I; await tick;
                emit IN_I; await tick;
                emit EXIT
        end loop
end module
```

2.2 Modeling the Gate

We choose to model the gate as being in one of possible four states: UP, DOWN, GOING_UP, and GOING_DOWN. We pick fixed values for the time it takes to go from GOING_UP to UP (4 time units) and from GOING_UP to UP (4 time units). The gate is commanded by the RAISE and LOWER signals, emitted by the controller, which are assumed to never be emitted simultaneously (thus the `relation` statement). The implementation is shown below:

```
module Gate:
    input RAISE, LOWER;
    output UP, DOWN, GOING_UP, GOING_DOWN;
    relation RAISE # LOWER;
    signal TIMER_UP, TIMER_DOWN in
    loop
        do
            sustain DOWN
        watching immediate RAISE;
        trap UPPING in
            [
                [
                    do
                        sustain GOING_UP;
                    watching immediate [LOWER or TIMER_UP];
                    exit UPPING
                ]
            ||
                [
                    await tick;
                    await tick;
                    await tick;
                    await tick;
                    emit TIMER_UP;
                    exit UPPING
                ]
            ]
        end trap;
        do
            sustain UP
        watching immediate LOWER;
        trap DOWNING in
            [
                [
                    do
                        sustain GOING_DOWN;
                    watching immediate [RAISE or TIMER_DOWN];
                    exit DOWNING;
                ]
            ||
                [
```

```
                    await tick;
                    await tick;
                    await tick;
                    await tick;
                    emit TIMER_DOWN;
                    exit DOWNING
                ]
            ]
        end trap
    end loop
    end signal
end module
```

One possible improvement to the model of the gate would involve letting the gate show a more realistic behavior in terms of modeling the fact that it would take less for it to get all the way up if it is not completely down. Another improvement would be to allow it to arrive up (or down) earlier than 4 time instants; this would involve the addition of external signals to model that. Note that the implementation above presents a more restricted (or worse) case, thus the improvements would still not violate the properties, but allow more complicate scenarios to happen, with an obvious enlargement of the overall state space and compilation times.

2.3 The Gate Controller

The implementation of the gate controller is pretty simple: at every time instant, if there is any train in the region, emit the `LOWER` signal, otherwise, emit the `RAISE` signal:

```
module Controller:
        input ENTER_R, ENTER_I, IN_R, IN_I, EXIT;
        output RAISE, LOWER;
        every immediate tick do
                present [IN_R or IN_I] then
                        emit LOWER
                else
                        emit RAISE
                end present
        end every
end module
```

Note that the signals `LOWER` and `RAISE` are sustained when they need to be active. This takes advantage of the features of ESTEREL but if these signals are tied to some physical device such that repeatedly emitting them can cause damage or malfunction to the gate physical devices, it is not wise to use this implementation. The following alternate implementation would suit the same purpose (without violating the properties):

```
module Controller:
        input ENTER_R, ENTER_I, IN_R, IN_I, EXIT;
        output RAISE, LOWER;
        loop
                present [not (IN_R or IN_I)] then
                        % 1st pass through the loop
                        emit RAISE;
                end;
                await immediate [IN_R or IN_I];
                emit LOWER;
                await tick;
                await immediate [not (IN_R or IN_I)];
        end loop
end module
```

3 Verifying Safety Properties of ESTEREL Programs

The process of verifying safety properties of ESTEREL programs involves three steps. This process is fully explained in [5] and is as follows:

1. Translation of the properties from temporal logic to ESTEREL.
2. Compilation of the given program in parallel with the ESTEREL model of the properties.
3. Check for satisfaction/violation of the properties over the resulting finite state machine.

The translation from temporal logic formulas to ESTEREL programs roughly consists of recursively translating a given safety formula in into an ESTEREL program, which can be viewed as a *finite state acceptor* of the computations satisfying the formula. This automata introduces a special ESTEREL signal `SAT_`p which is emitted in exactly those reactions satisfying the formula p. This program is said to “model” the property. This technique is based on work presented in [8]. For practical reasons, the synthetic code emits a signal `VIOLATED_`p in every state that the formula is *not* satisfied. After compiling the formula with the given program, the checking algorithm consists of searching for a state where `VIOLATED_`p is emitted, returning successfully if no such state exists. Otherwise, it terminates returning a computation path starting in the initial state of the program an ending in a state that violates p.

The compilation step of the ESTEREL program is performed by the ESTEREL compiler, available from the ESTEREL development team[1]. The finite state machines generated by the ESTEREL compiler are such that, by construction, it is guaranteed that for all states s in them there exists at least a computation such that s is reachable from the initial state, i.e. there is no state that cannot be potentially reached from the initial state. This feature is what allows the compiler to be used towards constructing a finite state machine of the program and the property; by exhaustively generating all the reachable states of the program and the model of the properties, the ESTEREL compiler in effect performs model checking [3].

The procedure for checking is thus reduced to a linear search of the generated state machine. The compiler is exponential time and space in the worst case since it exhaustively generates all the state space, but it is reasonable to assume that programs and formulas useful in practice do not generally blow up the state space.

3.1 Class of properties supported for verification

The technique just outlined supports the verification of the class of linear temporal logic *safety formulas* over the alphabet of signal identifiers in the given program. Informally, these formulas stipulate that "something bad never happens." More formally, a formula is said to be a safety formula if any sequence violating it contains a prefix all whose infinite extensions violate it as well.

In temporal logic [7], a safety formula is an expression of the form $\Box\, p$, where $\Box$ is the "always" operator, which quantifies over all computation states and p is a "past" formula. The class of past formulas is defined to be the alphabet of signal names closed under the boolean operators and the linear temporal logic past operators (*previous, back-to, since, henceforth,* and *has-always-been*). The technique supports (efficiently) some so-called past *response* operators as well, namely, it supports the two following useful operators (which can be defined in terms of past operators):

- Bounded response formulas of the form $p \rightarrow \diamondminus_d q$, whose value is true at the current time instant if p was true d times instants in the past and q has been true at some point from then up to the current instant.
- Bounded ensures formulas of the form $p \rightsquigarrow \diamondminus_d q$, whose value is true at the current time instant if p was true in the past d instants and q is true in the present instant.

We refer the reader to [7] for further details on temporal logic safety properties and to [5] for the technique and expressions supported by it.

[1] Contact by electronic mail to `esterel-request@cma.cma.fr`.

4 Formal Verification of the GRC Solution

The problem consists of showing that the system satisfies the following properties, expressed in temporal logic [7]:

Safety Property: $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$. The gate is down if there is any train in the crossing.

It is easy to see that this property corresponds to the original formulation of the property: the signal `IN_I` is active when there is some train in some track in the intersection (i.e the union of all the occupancy intervals) and `DOWN` corresponds to the gate being down ($g(t) = 0$), thus the expression for this property is: $\Box$ `IN_I` $\rightarrow$ `DOWN`.

Utility Property: $t \notin \cup_i[\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$. The gate is up when no trains are in the crossing.

It is pretty clear that in a physical system like the one described by the problem, not all given values for ξ_1 and ξ_2 will allow a solution to the problem to be developed. The range of valid values for the constants are influenced by values such as the speed of the trains, the speed of the gate or the distance from the approach sensors to I. Given that we have chosen models for the physical parts of the system, the property has to capture the valid values in it:

- The value for ξ_1 cannot be smaller than the time from then system starting lowering the gate until the train enters I. In our implementation the gate starts being lowered at the same time instant that any train enters R thus in our case, regardless of whatever value ξ_1 may have, we can take the signal `IN_R` to be the "indicator" of `IN_I` becoming active later than ξ_1 time units afterwards. We have to rely in this approach because the verification technique does not support expressions involving "future" events. This approach leads to a stronger property as long as ξ_1 is longer than what it takes for a train to get to I once it enters R and it does not take into account the constant itself. If our implementation would take time between `IN_R` becoming active and the gate starting to lower, then the expression of the utility property would have to be modified to include it.
- The value of ξ_2 must be larger than the time it takes for the gate go all the way up after all trains have left I (in out model of the gate, this is 4 time instants). In out model of the trains the time it takes a train to leave R would have to be added (zero in our case). We take ξ_2 to be 5 for the utility property.

We thus take the most restrictive value of the constant to prove that the system does not violate the utility property. The choice of ignoring ξ_1 in favor of using the sensor system for the region allows the the expression of the utility formula to simplify:

$$\Box \neg(\texttt{IN\_R} \vee \texttt{IN\_I}) \rightsquigarrow \diamondsuit_5 \texttt{UP}.$$

This property specifies that "if there is no train in I nor R during any time interval of size 5, then the gate must be up." In the general case, since we cannot formulate future-tense formulas, we would have to use the assumption mentioned for ξ_1 and the general expression of the property would be: $\Box \neg(\texttt{IN\_R} \vee \texttt{IN\_I}) \rightsquigarrow \diamondsuit_{\xi_2} \texttt{UP}$.

This property uses the $\rightsquigarrow\diamondsuit$ (ensures) operator, which is a "response" operator. Its semantics is as follows: a formula $p \rightsquigarrow \diamondsuit_d q$ specifies that "for any computation interval of size d where p holds continuously, q must hold at the end of the interval." More formally, this operator can be defined in terms of past-tense temporal logic operators[2]:

$$p \rightsquigarrow \diamondsuit_d q \equiv (\bigwedge_{k=0}^{d-1} \ominus^k p) \rightarrow q.$$

These two properties are technically safety properties (since they refer to finite intervals), thus they fall into the class of properties that the technique can formally verify, although the nature of the utility property is slightly different than the "safety" property –it captures requirements that a null implementation (e.g. the gate being down continuously) would not satisfy, even if it satisfies the "safety" property.

We have verified both of the properties using the automated tools associated with the technique outlined in Section 3. For the system to behave as specified, the trains must take longer to reach I once they enter R than the gate to lower all the way down –the properties cannot be proved true at all times by the system otherwise.

The main module used for the verification is shown below. The GRC module is run in parallel with the models of the properties:

```
module Main:
        input APPROACH_0, ..., APPROACH_N;
        output UP, DOWN;
        signal IN_R, IN_I in
                run GRC
        ||
                run Safety
        ||
                run Utility
end signal
```

[2]Note that $\ominus$ is the "previous" operator and $\ominus^k p$ are k applications of $\ominus$ on expression p. The expression $\ominus^0 p$ equivales to p

The compiled state space of the program is 1468 states. We note that this size is not affected by any nor both models of the properties being compiled in parallel with the implementation. This is a common case when properties are not violated by the implementation; the intuitive reason is that the actual value of the properties is always true at all states, thus there is no need to track different values of the model of the properties at each point.

Another quality that the ESTEREL-generated state machines exhibit is that each state actually represents a decision tree which encodes all the internal computations performed within one reaction. In the synchronous model, the reaction to a set of inputs is instantaneous, and the next state of the reaction can be one of several, depending on the state, the inputs and the decisions taken during the reaction based on those. In most state machine based approaches, each state encodes one of these decisions, thus that is usually considered to be the actual size of the system. In our case, the total number of states for the 1468-state machine amounts to 2465.

4.1 Performance of the Verification Process

The performance of the different stages in the verification process has been measured in a Sparc10 workstation from Sun Microsystems with 32Mb of core memory and 62Mb of swap space. The implementation constants picked are as described above, i.e. each train takes 6 time units to enter the intersection once it enters the region and takes 4 time units to leave the intersection once it has entered it, the gate takes at most 4 time units to go all the way up or all the way down. With this configuration, the controller itself has 4 states when compiled, the gate has 11 states and each of the 3 trains has 12 states.

The verifier tool, called `tl2strl`, is a program whose input is a (file with a) set of temporal logic formulas and whose output is (a file with) an ESTEREL program. For the GRC properties, the execution time of the translation is approximately 0.45 seconds. This is a fairly short time because the properties are really small. The translation tool has been developed using the Standard ML programming language.

The next step is to compile the implementation of the solution with the model of the properties composed in parallel with it using the ESTEREL V3 compiler. The compilation of the solution with the model of the safety property takes approximately 10 seconds. The compilation with the model of the utility property takes approximately 11 seconds. The compilation with the models of both properties simultaneously takes approximately 14 seconds.

The last step in the process is to search the resulting finite state machine for any state emitting any of the two signals indicating the violation of the properties. The search takes approximately 11.2 seconds for each property. At this time the search engine over the model is implemented by a rather crude and inefficient scripting language (Perl). This search time can be optimized several orders of magnitude by using an efficient language such as C and performing the search in core memory, or using readily available ESTEREL tools such as AGEL [1].

The various execution times of the stages described can be more or less improved, however, the theoretical bottleneck lies in the ESTEREL compilation step, since the resulting state space of the system can be, in the worst case, exponential on the size of the program. We have also performed verification in several problems (see [6] for a more detailed description of a real-world application), and it has been found to be highly effective in practice.

5 Discussion

We have presented a solution to the Generalized Railroad Crossing within the environment of ESTEREL. The solution includes the formal verification of the two properties listed in the problem statement using a verification technique for safety properties of ESTEREL programs.

Because ESTEREL only allows the specification of purely deterministic behaviors, models for the trains and the gate have had to be chosen so that the verification can be performed against a concrete system. The verification is limited to safety properties expressed in linear-time temporal logic, thus no expressions in future tense are allowed. This forced us to prove a stronger version of the utility property.

At one point during the process of verification of the properties, the verification tools pointed out a (rather subtle) scenario allowed by the implementation which violated one of the properties, allowing us to locate an error in the implementation which we believe would otherwise be very difficult to find in other circumstances.

The advantage of the approach used for verification of programs based on the synchronous model of ESTEREL is that the *actual text* of the program is verified, eliminating any room for errors introduced by any possible manual or semi-manual translation from a verified model to an executable implementation as it is common in other verification methods.

References

[1] AGEL workshop manual version 3.0, 1989. Produced by ILOG.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[3] 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.

[4] C. Heitmeyer, R.D. Jeffords, and B. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proceedings 10th International Workshop on Real-Time Operating Systems and Software*, May 1993.

[5] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen. Verification of safety properties of ESTEREL programs and a telecommunications application.

[6] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen. A formal approach to reactive systems software: A telecommunications application in ESTEREL. In *Workshop on Industrial-strength Formal Specification Techniques*, April 1995.

[7] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer-Verlag, 1992.

[8] 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.

A The ESTEREL programming language

Overview

ESTEREL [2] is a language, with a precisely defined mathematical 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. 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. Furthermore, since this implementation is a finite state machine, the maximum amount of time taken by any reaction can be accurately bounded if the execution times of the transitions are known.

Language Constructs

Our implementation of the solution and the translation of temporal logic safety formulas uses only a subset of ESTEREL constructs, which we motivate here. The `emit S` statement, where `S` is a signal, indicates that `S` is present in the current reaction. The `sustain S` construct is equivalent to emitting `S` forever. The `await S` construct blocks until the next reaction in which `S` is present. The `await immediate S` triggers if `S` is present in the current reaction; otherwise, it blocks until the next reaction in which `S` is present.

The `present S then BODY1 else BODY2 end` construct checks whether the signal `S` is present in the current reaction; if so, `BODY1` is executed, otherwise `BODY2` is executed. The `every S do BODY end` construct restarts `BODY` in every future reaction in which `S` is present. The `every immediate S do BODY end` construct restarts `BODY` in every reaction, including the current one, in which `S` is present. The `do BODY watching S end` construct executes `BODY` until the next reaction in which `S` is present. The `do BODY watching immediate S end` construct executes `BODY` until the first reaction, including the current one, in which `S` is present. Finally, the `loop BODY end` is an infinite loop, in which `BODY` is executed continually.

More generally, boolean combinations of signals can be used instead of a single signal `S` in all the constructs above (except for `emit`), with the obvious interpretation. The reserved signal `tick` is by definition present in every reaction. Thus, `await tick` blocks until the next reaction, and `every immediate tick do BODY end` restarts `BODY` in every reaction.