Communication Protocols Verification with Esterel

Jorge Graña Gil Manuel Vilares Ferro Raphäel Bernhard

Abstract

This work summarizes design, implementation and verification processes of a digital telephone switchboard in the ESTEREL real-time programming environment. Our aim is to show the modularity in the description and of flexibility the verification process.

We also show the control synchronization mechanisms to coordinate concurrent processes. The goal is to prevent in compile-time deadlock and lockout phenomena, a feature that is not available in most programming languages.

Key Words and Phrases: Automata Verification, Communication Protocols, Reactive System, Real-Time System, Synchronous Programming.

1 Introduction

A telephonic connection is a simple and well-known communication protocol, but not easy to implement. Synchronism problems among phones can appear in the connection phase. If we do not dispose of an efficient method for treating the concerned signals, deadlocks could show up on our systems. Furthermore, we need to detect these phenomena *a priori* in order to avoid a total redesign of the system.

Synchronous programming languages, like ESTEREL, have become very convenient for dealing with these kinds of problems in the context of reactive systems. In effect, their semantics makes the verification process,

Jorge Graña Gil and Manuel Vilares Ferro are with the Computer Science Department, University of Corunna, Campus de Elviña s/n, 15071 La Coruña, Spain. E-mail: grana@dc.fi.udc.es, vilares@dc.fi.udc.es.

Raphäel Bernhard is with France Telecom in the Centre National d'Etudes des Télécommunications, 905 rue Albert Einstein, 06921 Sophia Antipolis Cedex, France. E-mail: bernhard@sophia.cnet.fr.

The present work was partially supported by projects XUGA10501A93, XUGA10501B93 and XUGA10502B94 from the *Xunta de Galicia*, the Autonomous Government of Galicia.

and the detection and elimination of typical synchronism problems in communication protocols easy.

Section 2 of this work is an overview of fundamental notions on reactive systems, and ESTEREL features. In Section 3, we describe the specification of the communication protocol for a digital switchboard. Section 4 is a first approach to the description of a phone in ESTEREL, and Section 5 outlines the definitive implementation. A final conclusion can be found in Section 6.

2 Reactive systems

Reactive real-time systems are computer-based systems which must react instantaneously to events within their environment. They can be seen as a "black box" which receives input signals and emits answers in the form of output signals. Going more deeply into the architecture, they can consist of several interior devices which run in parallel and communicate with one another by means of local signals, as is shown in figure 1.

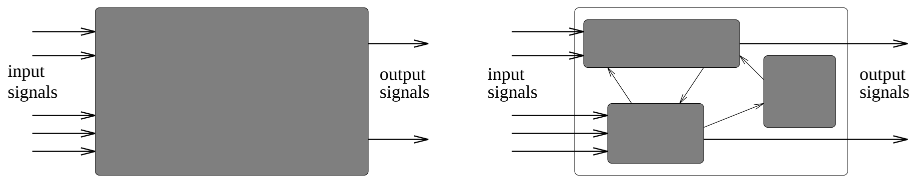

Figure 1: Macroscopic vision and interior vision of a reactive system

The connect operation depends on the exact ordering of the input events, which implies the existence of some kind of synchronization control mechanism. In this context, conventional computer-programming techniques are not suitable. A solution is to adopt the *perfect synchrony* hypothesis [2], which states that time is defined externally to programs by the flow of inputs, and that program internal bookkeeping is done in zero-delay with respect to all external time units [3]. This hypothesis shows the problem of inter-process communication. To solve it, all synchronous languages consider broadcasting as a basis to communicate concurrent statements. In this manner, an emitted signal can be received by any other process in scope.

However, as a direct consequence of the perfect synchrony assumption, two important questions arise: causality cycles and instantaneous loops[1]. A way of thinking when dealing with these kinds of problems is to accept them and let systems detect a deadlock in run-time. A more high premium is represented by synchronous languages that detects them at compile-time, and therefore no ill-running code is produced.

[1] the first one appears when there are circular dependencies between the status of the processes in the system; the second one makes reference to a loop where no statement in its body takes time

Another important problem to be taken into account is the verification of applications. The programming environment should include mechanisms to reduce the total system described as a quotient of the one under study. The parameter of this reduction should be a user-defined abstraction criterion, which embodies a particular point of view on the system in order to verify particular properties.

Esterel is a synchronous programming language for reactive systems that presents the features described above [4]. The compiler translates a source program into a finite state machine adapted for dealing with parallel activities and for synchronizing concurrent tasks. The language has an associated environment, that integrates all activities: edition, compilation, graphical simulation, symbolic debugging and formal verification.

3 Specification of the example

Our aim is to simulate a practical environment where a certain number of users talk by using several phones. Formally, we shall assume a definition for the phones that can fit most usual construction techniques. So, we shall represent a phone by the following elements:

- An *earpiece*, which can be picked up or hung up by the user. We will use "up" and "down" respectively, since they are intuitive enough.
- Some *buttons*, in order to call the other phones.
- A *bell*, which informs the phone user when there is a call.
- A *tone emitter*, which informs the user about what phase the communication is in. We have chosen five tones: "go", "calling", "talking", "busy" and "none".

A phone is a common example of a reactive system. In effect, as we show in figure 2, a phone interacts with the user receiving and emitting signals.

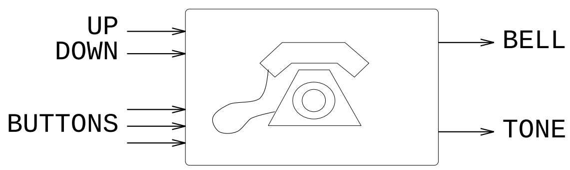

Figure 2: A phone as a reactive system

To be more exact, the input signals are as follows:

- `UP`, received when the user picks the phone up.
- `DOWN`, received when the user hangs the phone up.
- `BUTTON_i`, received when the user pushes the button number i in order to call the phone number i.

In relation to output signals, they are:

- `BELL`, informing the user when there is a call.
- `TONE`, informing the user about the state of the communication.

In order to handle the reactive behavior of several phones, we shall need to implement a digital switchboard consisting of automatic relays and dynamic connectors, with the phones connected to it.

The protocol we use to treat the information during the communication process is very simple: when two phones talk to one another, the protocol must ensure that they start talking at the same time and end talking at the same time. This is the main requirement in our application.

4 The basic model

In order to approach for the first time the process of writing the behavior of a phone in ESTEREL, we shall build an application that implements two phones directly connected. This application will be called TWO_PHONES. To shorten the explanation, we shall only comment on the most important modules. We shall also omit their interface declarations and the non-relevant signal substitutions in the run instructions for the sub-modules[2].

4.1 Expressing the behavior of a phone in ESTEREL

Basically, the behavior of a phone is an infinite loop with two sequential blocks of code:

1. The phone is free and can be called or picked up by the user.
2. The phone is not free and cannot be called, since it will be talking or trying to establish a communication.

We can express this behavior by the following module, called PHONE:

```
module PHONE:
   loop
      <free>;
      <busy>
   end loop
end module
```

A phone is free until the instant it receives a call or the user picks it up. So, we write a new module called FREE that makes the signal I_AM_FREE present in every reaction, while the signals YOU_CALL_ME or MY_USR_UP do not arrive. The meanings of the signals are as follows:

- I_AM_FREE is emitted when the phone can receive a call.
- YOU_CALL_ME is received when the remote phone calls.
- MY_USR_UP is received when the user picks the phone up.

The corresponding module is as follows:

```
module FREE:
   do
      sustain I_AM_FREE
   upto [MY_USR_UP or YOU_CALL_ME]
end module
```

[2]the run instruction is equivalent to the pre-processor #include instruction, and a signal substitution in a run instruction is equivalent to a #define instruction in a C program.

We can now replace <free> by the module FREE, and <busy> by a piece of code that studies what event has happened:

- If YOU_CALL_ME arrives, the phone turns into a called phone and we run the module CALLED_PHONE.
- If MY_USR_UP arrives, we run the module BUTTONS, since the phone will be a caller phone only when the user pushes the button.
- We also consider a particular case: when the signals YOU_CALL_ME and MY_USR_UP arrive at the same time, both phones will be talking, and we run the module TALKING.

We complete the module PHONE by running a bell manager and a tone manager in parallel with the rest of the code. Moreover, from now on we include the signals which relate to the bell and the tone. These new signals will help us to understand the global behavior:

```
module PHONE:
   [
      run BELL_MANAGER
   ||
      run TONE_MANAGER
   ||
      loop
         emit TONE_NO_TONE;
         run FREE;
         await
            case immediate [YOU_CALL_ME and MY_USR_UP] do
               run TALKING
            case immediate YOU_CALL_ME do
               run CALLED_PHONE
            case immediate MY_USR_UP do
               run BUTTONS
         end await
      end loop
   ]
end module
```

where emit TONE_NO_TONE emits the signal constituting its argument, await waits for one of the three events, and immediate tests for the immediate presence of these events.

When two phones start talking, the module TALKING receives the control in both phones. That is, there will be two instances of the module TALKING running in parallel, since the global application consists of two instances of the module PHONE running in parallel, as we shall see later. Both phones reach the module TALKING from different ways, but they are executing the same statements in this instant. The connection is established and the conversation lasts until the moment when one of the users hangs the phone up. At that moment, the other phone emits the busy tone[3] and the only thing that the user can do is to hang the phone up too, in order to let it be free again.

```
module TALKING:
   emit TONE_TALKING;
   await [MY_USR_DOWN or YOUR_USR_DOWN];
   do
      emit TONE_BUSY
   upto immediate MY_USR_DOWN
end module
```

[3]the behavior in terms of the tones emitted is taken from the French phones.

As we have seen, when the phone receives a call, the module CALLED_PHONE is executed. The phone bell starts ringing and we consider the following events:

- If the caller user hangs the phone up before somebody attends the call, then the called phone receives the signal YOUR_USR_DOWN, the bell stops ringing, and the phone will be free again.
- If the called user picks the phone up, then the called phone receives the signal MY_USR_UP, the bell stops ringing, and both users will be talking.
- If both signals arrive together, the bell stops ringing, the tone emitter emits the busy tone, and the only thing the user can do is to hang the phone up, in order for it to be free again.

Hence, the module consists of the following code:

```
module CALLED_PHONE:
   emit MY_BELL_ON;
   await
      case [YOUR_USR_DOWN and MY_USR_UP] do
         emit MY_BELL_OFF;
         emit TONE_BUSY;
         await MY_USR_DOWN
      case YOUR_USR_DOWN do
         emit MY_BELL_OFF
      case MY_USR_UP do
         emit MY_BELL_OFF;
         run TALKING
   end await
end module
```

When the user picks the phone up in order to make a call, the phone is not a caller phone yet. We must check whether the user pushes the button or decides to give up his call. So, we run the module BUTTONS, which waits for the user to push the button, and watches the signal MY_USR_DOWN in every reaction. The module CALLER_PHONE will receive the control only if the signal MY_USR_BUTTON has been received before. To implement this, we use a **trap-handle** statement defining an escape named BUTTON, and enclosing a watchdog control structure.

```
module BUTTONS:
   trap BUTTON in
      do
         emit TONE_GO;
         await MY_USR_BUTTON do
            exit BUTTON
         end await
      watching MY_USR_DOWN
   handle BUTTON do
      run CALLER_PHONE
   end trap
end module
```

In the module CALLER_PHONE we know that the user has pushed the button. But, before calling the remote phone, we must check first whether it is free or not, by testing the presence of the signal YOU_ARE_FREE:

- If it is free, the caller phone calls it by sending the signal I_CALL_YOU, and waits for the remote user to pick the phone up, emitting the tone calling during this waiting time.

- If it is not free, the caller phone emits the tone busy and the only thing the user can do is to hang up.

As in the module BUTTONS, we must watch the signal MY_USR_DOWN in every reaction, because the user in the caller phone can decide to hang the phone up if nobody attends the call. Therefore, the control reaches the module TALKING only if the signal YOUR_USR_UP arrives before.

```
module CALLER_PHONE:
   trap CONNECTION_ESTABLISHED in
      do
         present YOU_ARE_FREE then
            emit I_CALL_YOU;
            do
               emit TONE_CALLING
            upto immediate YOUR_USR_UP;
            exit CONNECTION_ESTABLISHED
         else
            emit TONE_BUSY
         end present
      upto MY_USR_DOWN
   handle CONNECTION_ESTABLISHED do
      run TALKING
   end trap
end module
```

The module above ends the description of a phone in ESTEREL. Now, we want to connect two of those phones. We can do it by writing a new module called TWO_PHONES. The interface of this module consists of the real inputs and outputs in our application world. The `relation` instruction declares the input signals UP and DOWN as incompatible, i.e. a phone cannot produce both signals in the same event. The body of the module consists of two modules of PHONE running in parallel. Here, we make the substitutions explicit in the `run` instructions, since they show how to connect modules in ESTEREL. So, by using this mechanism of name change, we shall have:

- UP_1 will be MY_USR_UP in the first phone, and YOUR_USR_UP in the second one.
- DONW_1 will be MY_USR_DOWN in the first phone, and YOUR_USR_DOWN in the second one, and so forth.

Finally, to establish a perfect synchrony between both phones, we also need four local signals:

- FREE_1 will be I_AM_FREE in the first phone, and YOU_ARE_FREE in the second one.
- T1_CALLS_T2 will be I_CALL_YOU in the first phone, and YOU_CALL_ME in the second one, and so forth.

The corresponding implementation is given by the code that follows:

```
module TWO_PHONES:
   input
      UP_1, DOWN_1, BUTTON_1,
      UP_2, DOWN_2, BUTTON_2;
   output
      BELL_1, TONE_1 (string),
      BELL_2, TONE_2 (string);
   relation
      UP_1 # DOWN_1,
      UP_2 # DOWN_2;
```

```
   signal
      FREE_1, T1_CALLS_T2,
      FREE_2, T2_CALLS_T1
   in
      [
         run PHONE   [signal  UP_1/MY_USR_UP,
                            DOWN_1/MY_USR_DOWN,
                          BUTTON_1/MY_USR_BUTTON,
                            BELL_1/MY_BELL,
                            TONE_1/MY_TONE,
                            FREE_1/I_AM_FREE,
                            FREE_2/YOU_ARE_FREE,
                              UP_2/YOUR_USR_UP,
                            DOWN_2/YOUR_USR_DOWN,
                       T1_CALLS_T2/I_CALL_YOU,
                       T2_CALLS_T1/YOU_CALL_ME]
      ||
         run PHONE   [signal  UP_2/MY_USR_UP,
                            DOWN_2/MY_USR_DOWN,
                          BUTTON_2/MY_USR_BUTTON,
                            BELL_2/MY_BELL,
                            TONE_2/MY_TONE,
                            FREE_2/I_AM_FREE,
                            FREE_1/YOU_ARE_FREE,
                              UP_1/YOUR_USR_UP,
                            DOWN_1/YOUR_USR_DOWN,
                       T2_CALLS_T1/I_CALL_YOU,
                       T1_CALLS_T2/YOU_CALL_ME]
      ]
   end signal
end module
```

At this moment, our application is ready to be compiled.

4.2 Compiling the application

Let us think about this situation: the first phone is free, and the user in the second one picks it up and pushes the button. The first phone will be executing the following piece of code:

```
module FREE:
   do
      sustain FREE_1
   upto [UP_1 or T2_CALLS_T1]
   ...
```

and the second phone will be executing:

```
module CALLER_PHONE:
   ...
   present FREE_1 then
      emit T2_CALLS_T1;
   ...
```

Before the second phone emits `T2_CALLS_T1`, the signal `FREE_1` is present. When `T2_CALLS_T1` is emitted, the body of the `do-upto` instruction is aborted, making `FREE_1` not present. Since the `emit` instruction takes no time and the `do-upto` instruction aborts its body instantaneously, the signal `FREE_1` has two possible states at that same instant. Therefore, these two pieces of code running together produce a causality cycle, and our application is rejected by the compiler. In order to avoid this cycle between the signals I_AM_FREE and YOU_CALL_ME, we must rewrite the module FREE and replace the do-upto instruction by a trap structure:

```
module FREE:
   trap BUSY in
      [
         sustain I_AM_FREE
      ||
         await [MY_USR_UP or YOU_CALL_ME]; exit BUSY
      ]
   end trap
end module
```

As before, when the signal YOU_CALL_ME arrives, the body of the trap is also aborted and the control will also leave that structure, but the semantics of the trap instruction lets the signal I_AM_FREE be present in the current reaction. This difference allows us to eliminate the causality cycle. Now, there is no logical or physical problem with the signals. So, the only thing that remains undone is to analyze the behavior.

4.3 Verifying the application

In order to check whether our model implements the problem specifications, we rewrite the module TALKING by adding two new signals:

- START_TALKING, emitted when the phone starts talking.
- END_TALKING, emitted when the conversation is finished.

Since they are output signals, the global behavior is not modified:

```
module TALKING:
   emit START_TALKING;
   emit TONE_TALKING;
   await [MY_USR_DOWN or YOUR_USR_DOWN];
   emit END_TALKING;
   do
      emit TONE_BUSY
   upto immediate MY_USR_DOWN
end module
```

Now, the mechanism we use to study the behavior is the following:

1. We rewrite the module TWO_PHONES by adding four global output signals, TALKING_1, TALKING_2, NOT_TALKING_1 and NOT_TALKING_2, where the signal START_TALKING (resp. END_TALKING) is replaced by TALKING_1 (resp. NOT_TALKING_1) in the first phone, and by TALKING_2 (resp. NOT_TALKING_2) in the second one.
2. We create a verification criterion by selecting these four signals, i.e. a filter that removes the rest of the signals in the application.
3. We apply this criterion to the full automaton, and obtain a reduced automaton in which only states and transitions in relation with the signals in the criterion appear.

The resulting automaton in figure 3 shows two non-expected behaviors. When reaching state 27, the phone 1 is talking alone because the signal TALKING_2 has not been emitted synchronously with the signal TALKING_1. When reaching state 28, the phone 2 is talking alone too.

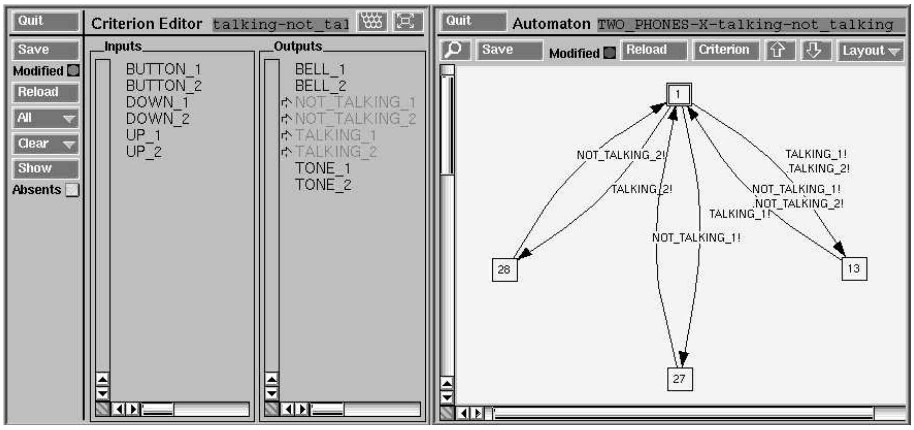

Figure 3: Anomalous behavior in the TWO_PHONES application

At this point, it is important to remark that if the programming language used to implement the application were not ESTEREL, these incorrect behaviors would probably not have been detected by the programmer.

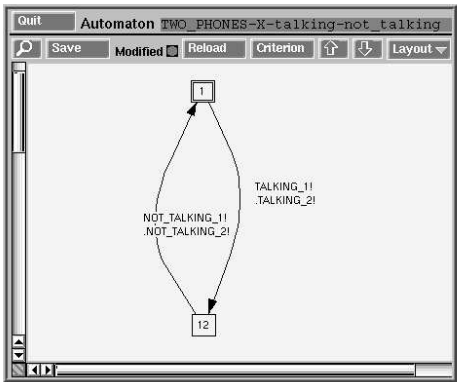

Figure 4: Correct behavior in the TWO_PHONES application

By using the debugger, we have found the problem in the module FREE: when the phone is hung, it cannot receive the signal MY_USR_UP immediately, since the signals UP and DOWN have been declared as incompatible; however, the phone can be called at the same time the user hangs it up. Therefore, we must use `immediate`, but only in front of the signal YOU_CALL_ME. This module is then modified as follows:

```
module FREE:
   trap BUSY in
      [
         sustain I_AM_FREE
      ||
         await immediate YOU_CALL_ME; exit BUSY
      ||
         await MY_USR_UP; exit BUSY
      ]
   end trap
end module
```

Once more, we compile the application and apply the criterion. Now, the new reduced automaton in figure 4 shows that the non-expected behaviors have disappeared, i.e. our application is right, and we can prove the safe behavior of both phones.

5 The digital switchboard

In our first attempt, the phones were connected directly. But, in real life, we have considered that no intelligence to establish a communication is situated on them. So, the automatic relays in the digital switchboard will perform this functionality. This is the essential feature that has been incorporated to the basic model as two separate modules called `RELAY` and `SWITCHBOARD`.

The module `RELAY` implements basically the same technique we use in the previous application: the **`present`** and **`trap`** structures avoiding the causality cycles between the signals `I_AM_FREE` and `YOU_CALL_ME` coming from any relay. The module `SWITCHBOARD`, consists of several relays running in parallel, which represents a good modular conception [1].

Finally, a module called `THREE_PHONES`, implements the real connection of the three phones, by running three instances of the module `PHONE` and one of the module `SWITCHBOARD`. The correct behavior of this new application validates the proposed construction.

6 Conclusion

We have shown how the perfect synchrony and well-defined mathematical semantics of ESTEREL is an excellent framework to model communication protocols, making them readable and manageable. Our aim was to develop a modular and incremental program for a telephonic switchboard, using the synchronous parallelism provided by ESTEREL.

Another important point is the verification of a reactive system, often described by several automata acting together. ESTEREL provides the user with formal proof mechanisms dedicated to computing small-scale models in order to check properties on the generated automaton, which allows us to ensure that the program is an implementation of our specification.

References

[1] R. Bernhard. Esterel v4: une extension modulaire d'Esterel. Thèse d'informatique, Université de Nice, 1992.

[2] G. Berry. The semantics of pure Esterel. In M. Broy, editor, *Program Design Calculi*, volume 118 of *Series F: Computer and System Sciences*, pages 361–409. NATO ASI Series, 1993.

[3] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1270–1282, 1991.

[4] D. Vergamini. Vérification de réseaux d'automates finis par équivalence observationnelle: le système Auto. Thèse d'informatique, Université de Nice, 1987.