Realizing Real-Time Systems from Synchronous Language Specifications

R. K. Shyamasundar*
School of Technology & Computer Science
Tata Institute of Fundamental Research
Homi Bhabha Road, Mumbai 400 005, India
shyam@tcs.tifr.res.in

J.V. Aghav
School of Technology & Computer Science
Tata Institute of Fundamental Research
Homi Bhabha Road, Mumbai 400 005, India
aghav@tcs.tifr.res.in

ABSTRACT

Synchronous languages specify reactions of the reactive/real-time systems to its environment within the language. This makes it easier to validate and verify the systems relative to the assumptions and notions of simultaneity. However, concrete embedding of the code makes it necessary to predict time for reactions of the underlying run-time systems and the sensors so that the strict notion of simultaneity can be relaxed keeping the logical correctness intact. Further, for hard real-time systems, there is a need for referring to clock times. In this paper, we describe a method that uses timed annotations for ESTEREL programs that makes it possible to predict the timing constraints on run-time system, the sensors and clock times and validate the concrete realization relative to time-annotated ESTEREL specifications. Using such time annotations, it is possible to establish that the concrete implementation is indeed a correct realization of the abstract system specified in annotated ESTEREL. Further, the method establishes time constraints to be satisfied by the concrete architectures for realizing the logical specification of the system. We shall illustrate the design and implementation of the tool using some of the existing code generation tools of the ESTEREL environment. Furthermore, the method can also be used for arriving at scheduling constraints on the underlying asynchronous tasks.

1. INTRODUCTION

There have been a variety of programming languages for the design of real-time systems. Various models and logics [?] employed to build real time systems and their abstraction to FSA for purposes of verification are given in [?]. In real-time languages, constructs allow us to specify and realize timing constraints of the implemented systems. Synchronous languages [?, ?] are another important family of languages that have been widely used for programming reactive systems. In these languages, the following logical assumptions (often referred to as *perfect synchrony*) are made (i) The program produces its outputs in no observable time, (ii) Concurrent statements evolve in a tightly coupled way, and (iii) Communication is done by instantaneous broadcasting, the receiver receiving it exactly at the time it is sent. Thus, in the context of a synchronous language, several variables are allowed to change their values in a single step (*i.e.* simultaneity is possible). However, such things are not possible in an asynchronous setting on a given architecture. While assumptions such as infinitely fast machines are useful to verify anomalies/paradoxes in the specifications at the logical level, there is a need to relax the requirement of simultaneity for the object code generated running on real machines. The standard acceptable relaxation is that we divide the time line into many small intervals and all the events that occur in the same interval (or instant in the context of synchronous languages) are considered simultaneous. Obviously there is a need for ensuring that these assumptions are met. Furthermore, in an actual implementation there will be need to refer to clock-time. Since there is no clock in ESTEREL all these signals have to come from the environment. Designing real-time systems with synchronous technologies, follow the two-step methodology given below:

1. Establish the logical correctness (following the classical Compile-simulate-verify cycle in ESTEREL).

2. Establish the satisfaction of time constraints are met taking into account the synchrony/simultaneity and the clock-time references; signals/events happening in the same instant are treated to be *simultaneous*.

The general methodology of ESTEREL validation is depicted in Figure ??.

In this paper, we shall propose a method and the design of a tool for (i) specifying timing constraints on the signals in the environment and for execution of code snippets, in ESTEREL specifications, synchrony/simultaneity assumptions, and (ii) a method and a tool for establishing that the implementation is a realization of the specification on the given architecture. Our method is based on annotating ESTEREL programs using the notation called *TimeC* explored in [?] in the context of verifying constraints in compilers meant for ILP processors and deriving the global constraints for the

*Work done under grant from the Indo-French Centre for the Promotion of Advanced Research, New Delhi, under the project 2202-1 *Formal specification and verification of hybrid and reactive systems.*

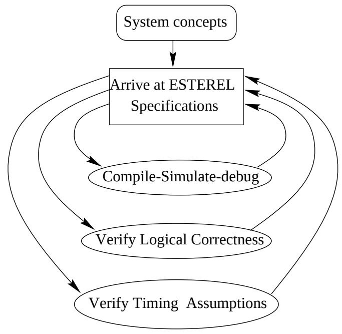

Figure 1: Methodology of Esterel code Validation

satisfaction of the synchrony hypothesis. One of the novel features of, *TimeC*, is that the language is independent of the base language being used to develop embedded applications. The main contributions of the paper are:

1. Specifying time constraints using *TimeC* in synchronous languages such as ESTEREL with reference to the signal environment that may include clocks.
2. A method of relating timing constraints on the signal environment of ESTEREL for the satisfaction of synchrony assumptions. Also, from the method it is possible to derive scheduling constraints of the external asynchronous tasks used for meeting the timing constraints of the signal environment.
3. The structure of a tool for the proposed method of validating the timing constraints relative to the synchronous constraints.

In the following, we shall give an overview of the method. For lack of space, we shall only be sketching the method and the interested reader is referred to the full paper.

2. AN INFORMAL OVERVIEW

Our approach is based on using timing annotations in the original ESTEREL program and deriving timing constraints for the satisfaction of the synchrony hypothesis.. The method essentially consists of

(i) Annotating ESTEREL fragments with the constraints,

(ii) Embed the timing constraints in the program through *logical* (ghost) signals without changing the timing properties of the original program,

(iii) Obtain the automata whose transitions consists of labels consisting of the original signals of the program and the additional logical signals introduced,

(iv) Derive the timing constraints for each of the transitions for the satisfaction of the synchrony hypothesis, and

(v) Verify the validity of the timing constraints.

An overview of the various steps of the method is informally described below using the fragment of the ESTEREL program used for controlling the path of a boat against a given direction as shown in Figure ??.

```
      module closeness:
      function close_enough():integer;
      function reached_p():integer;
      output suppress;
      output terminate;
      output o_continue;
         trap term in
M0:        loop
M1:          if (close_enough() = 1)
                then emit suppress
M2:
             else nothing
M3:             end; %if
M4:         if (reached_p() = 1)
                then emit terminate;exit term;
M5:
             else nothing
M6:          end; % if
M7:             await tick;
M8:        end; % loop
         end; % trap
         .
```

Figure 2: ESTEREL Fragment with Location Labels

No.	Constraint	Comments
1.	time(M2) - time (M1) $\leq$ 10 units	time for testing & then-branch execution
2.	time(M3) - time (M1) $\leq$ 6 units	time for testing & else-branch execution
3.	time(M5) - time (M4) $\leq$ 8 units	time for testing & then-branch execution
4.	time(M6) - time (M4) $\leq$ 4 units	time for testing & else-branch execution
5.	time(M8) - time (M7) ? indeterminate	awaiting for an external event
6.	time(M0) - time (M1) = 3 units	time to get into next iteration

Table 1: Timing Constraints

Specifying Timing Constraints: Various time constraints such as maximal, minimal, exact [?] can be specified. The timing constraints are specified as follows:

1. Even though the input and output signals are assumed to be instantaneous in ESTEREL actuators cause delay in the completion of reaction. The time constraint to be satisfied by any concrete implementation can be specified by placing *markers* as in *TimeC* [?] at various locations of the code.
2. Now the time constraints at the respective locations are to be defined. For the fragment of the code shown in Fig. ??, *markers* have been shown as labels at various points of code and the constraints to be satisfied are shown in Table ??. Here, the predicate *time(ℓ)* indicates the global time when the control reaches label ℓ.

Note: It may be noted that constraints illustrated in (5) of Table ?? can also be used for specifying explicit clock times relative to the signals used. Some of these details are not further discussed here for want of space.

Deriving Global Timing Constraints: Having specified the architectural timing constraints, we have to arrive at the timing constraints to be satisfied for keeping the logical properties invariant.

If the code given is loop-free, then it is conceivable that validating the constraints is trivial. However, in the presence of a loop as is the case in the example, it is not clear as to how the constraints can be validated over all the possible computations. In fact, if the language is treated as just another *imperative language*, the answer is not easy as one has to account for the various interleaved interactions of the different instances of the body of the loop. However, in the context of ESTEREL which is a *perfectly synchronous language* [?] it is necessary to consider the interactions of the different execution instants of the body due to the global notion of the instant. In the case of ESTEREL (or any perfectly synchronous language):

1. Outputs are *produced synchronously* with the inputs; i.e., reactions to inputs happen in *zero time* (an infinitely fast machine). This is in principle the *perfect synchrony hypothesis*. At a programming language level, this is interpreted as: *control and Communications* are *compiled away*.

2. The behaviour can be treated as an infinite sequence of reactions (consisting of inputs and outputs), each reaction is referred to as an *instant*. A reaction takes place on the occurrence of every external event and a special signal *tick* denotes the occurrence of an instant.

3. Inputs are indeterminate; however reaction to any input is finite. Hence, the effects are not carried from one instant to another except for recording states.

Thus, in the context of ESTEREL validating the timing constraints corresponds to validating *perfect synchronicity*. This can be done by looking at all possible reactions from one instant to another. In other words, if the program consists of just one module then the illustration shown can be generalized naturally. However, in the context of several modules it becomes nontrivial.

With the given annotated program, the timing constraints can be obtained broadly through the following steps:

1. Introduce additional signals that carry the timing information specified along with the signals of the program without affecting the temporal properties of the program; emission of additional signals keeps the original timing properties invariant follows from the synchrony hypothesis.

2. Derive the global transition system of the program.

3. From the transitions and the synchrony hypothesis, derive the global timing constraints.

Some of the above aspects are further detailed below.

3. VALIDATING TIMING CONSTRAINTS

3.1 Specifying Time Constraints

Timed annotation needs the use of:

1. **Time markers:** The first aspect of annotations is to locate control points through the introduction of named markers at various points (control) in the source program. We use "%# `some_name`"' to denote a *marker* at some point of the source program. For convenience of reference and specifying constraints, we use "begin" and "end" suffixes for the markers. For the compiler these annotations are regarded as just comments and hence, will have no bearing on either execution or compilation. In the following block of ESTEREL code, "block_1_begin" and "block_1_end" are two markers. These two markers can be used to specify the timing constraints for the execution of the code segment between these two markers.

```
%# block_1_begin
Y := 100;
emit S1(Y);
Y := Y + 100;
X := 7;
emit S2(Y)
%# block_1_end
```

Note: (i) The names of the time markers are unique, and (ii) Markers can be nested but does not include any statement of ESTEREL that does not terminate in the same instant.

2. **Language for specifying time constraints among the various markers:** For instance, we could have the following constraints on the markers `block_begin` and `block_end`:

$$\texttt{time}(\texttt{block\_end}) - \texttt{time}(\texttt{block\_begin}) \leq 4 \texttt{ units}$$

In other words, we need to relate the various markers relative to clock time similar to that given in Table **??**. We use the fragment of specification language used in *TimeC* [?] that are needed for synchronous languages. In fact, since ESTEREL interfaces with C-language, we could use the complete language.

3.2 Validating Timing Constraints

The main steps of validating timing constraints in ESTEREL programs without local signals are described below:

1. Annotate the ESTEREL program with markers as discussed above. Compute the static time relative to the markers by traversing the code segment between a "begin" marker and its corresponding "end" marker. This is done recursively. Since, no instant-definable statement can be there within the marked code segments (note that in the loop-construct of ESTEREL there should be at least instant oriented statement to avoid causality), the procedure essentially amounts to traversing a straight line code in each module.

2. Add additional signals with correspondence to Time markers. That is, for signals of the form "some_name_begin" add the statement
`emit some_name_begin_zero`
at that point where suffix "zero" indicates that the time required is zero; for signals of the form "some_name_end" add the statement
`emit some_name_end_Amount`
at that point where suffix "Amount" is the maximum time that can elapse from the point of emission of the signal "some_name_begin". Note that the value of "Amount" needs to be a statically evaluatable value. The computation of these values is quite easy as no time-delay operators are allowed as mentioned already. Note that adding the additional emitting statements for the logical/ghost signals does not alter either the functional or the timing properties of the program due

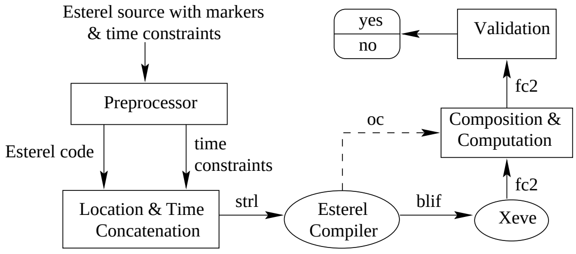

Figure 3: Tool for Validating Time Constraints

to synchrony hypothesis. For convenience (will be used in the implementation tool), we categorize signals based upon whether they are part of time constraint specification or not as follows:
1. *Lsignals*: Signals that are neither clock nor time references; in other words, these are signals that are pure synchronous signals.
2. *Tsignals*: Those signals that refer to clocks are time constrained; these include the signals introduced for each of the markers as in (2) above.
In this presentation, we have assumed that timing information is carried explicitly in the markers.

3. Obtain the reactive automaton for the program. Note that the transitions of the reactive automaton carry the marker-signals introduced.

4. For each transition compute the maximum time required. In the pure synchronous approach, all the signals are expected to occur at the same time. However, in the context of time-constraints, there is a need to evaluate the time needed for all the micro-steps spread across the modules in the same instant; in other words, one has to arrive at an expression which would clearly define the partial order of steps involved (note that time markers carry the control information).

5. Finally, *show that the maximum time for any reaction is less than the minimum time between any two instants (or the ticks) of the reactive automaton – thus, satisfying synchrony hypothesis.*

4. IMPLEMENTATION OVERVIEW

The tool is being built using the classical code generation tools of the ESTEREL environment having a structure as shown in Figure ??. The preprocessor computes the time requirements relative to the markers specified in the source language. In the next block the dummy statements of emission reflecting time requirements for the time-marker signals are added. The ESTEREL source with these additional statements carrying timing information becomes the input to the *strl* processor that generates intermediate code by ignoring the comments relative to the timing specification.

From the *fc2* code and *oc* code generated, the time requirement for each transition is computed. The resultant is the time *marked* reactive automaton in *fc2* format. In the validation process, each transition is checked for the satisfaction of the synchrony hypothesis. The successful exhaustion of the transitions validates the system relative to the specifications. Otherwise, it denotes invalidation of the timing constraints and the underlying tools of ESTEREL can be used in tracing the corresponding source fragments (note that marker signals keep track of the control information).

5. DISCUSSIONS

In this paper, we have sketched a method and the design of a tool for validating the timing constraints relative to the speed of the underlying architecture, assuring that the implementation is a realization of the logical synchronous language specification. The tool is under implementation and the generalized tool will validate timing assumptions or constraints; it will also provide the scheduling constraints on the various asynchronous constructs that could be instantiated – for example, through the `exec` in ESTEREL . The tool diagnoses the invalidating transitions with pointers to the respective code segments in the ESTEREL source. We also intend to use the tool to validate the timing constraints and other notions such as *guarantee* in real-time systems as envisaged through Timed CRP - a generalization of Communicating reactive processes [?, ?] that unifies the asynchronous and synchronous paradigms.

One of the distinct advantages of synchronous framework is that it establishes the logical correctness of the system. Since, the intermediate representation (a kind of automata) is close to implementation models, the implication is that one has verified the implementation itself. However, since the synchronous formalism is based on synchrony hypothesis, it does not consider the architectural constraints with reference to time, and asynchronous interfaces. The methodology proposed in the paper provides to bridge such a gap. We have been experimenting on the method on various embedded systems using such a methodology.

For further details the reader is referred to the full paper.

6. REFERENCES

[1] Rajeev Alur and David L Dill. A theory of timed automata. *TCS*, 126:183–235, 1994.

[2] Rajeev Alur and T. Henzinger. Logics and models of real time: A survey. In *REX Workshop on Real-Time: Theory in Practice*, LNCS, 600, pages 74–106. Springer Verlag, 1992.

[3] G. Berry and A. Benveniste. Synchronous approach to reactive and real-time systems: Another look at real-time programming. *Proc. IEEE*, 79:1270–1282, 1991.

[4] G Berry, S Ramesh, and R K Shyamasundar. Communicating Reactive Processes. In 20^{th} *POPLS*, pages 85–99, 1993.

[5] Gérard Berry and G Gonthier. The ESTEREL sychronous programming language: Design, semantics, implementation. *SCP*, 19(2):87–152, 1992.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th POPLS*, January 1987.

[7] Allen Leung, Krishna V Palem, and Amir Pnueli. TimeC: A time constraint language for ILP processor compilation. In 5^{th} *Australian Conf. on Parallel & Real Time Systems*, pages 57–71. Springer Verlag, 1998.

[8] R K Shyamasundar. Programming dynamic real-time systems in crp. In *Proc. of the CSA Jubilee Workshop on Computing and Intelligent Systems*, pages 76–89. Tata-McGraw Hill Publishing Co., 1994.