Validating Real-Time Constraints in Embedded Systems

R. K. Shyamasundar*
Tata Institute of Fundamental Research
Homi Bhabha Road, Mumbai 400 005
India, email:shyam@tcs.tifr.res.in

J.V. Aghav
Tata Institute of Fundamental Research
Homi Bhabha Road, Mumbai 400 005
India, email:aghav@tcs.tifr.res.in

Abstract

There is a growing demand for software tools that can assist in designing, analyzing and validating embedded real-time system applications. ESTEREL *a synchronous language, is widely used in the development of embedded systems and hardware/software codesign. In this paper, we describe a method that uses timed annotations for* ESTEREL *programs that makes it possible to predict the timing constraints required to be satisfied by the embedded system. Using the specified annotations and the programming environment of* ESTEREL *we describe a method and a tool for validating the concrete realization relative to time-annotated* ESTEREL *specifications. Also, the method derives time constraints to be satisfied by the concrete architectures for realizing the logical specification. We shall illustrate the technique with examples as well as the structure of the tool implemented.*

Keywords Embedded Systems, Esterel, Real-Time Systems, Synchronous Languages, Validation & Verification.

1. Introduction

There has been a dramatic growth of not only embedded system applications but also in the complexity of software required for these applications. A large class of the embedded applications is hard real-time sensitive. In hard real-time systems it is essential to have the right computation at the right time. In other words, it is not only necessary to establish the functional correctness of the system but also the timeliness of the system. This demands a methodology and a programming environment that will aid development of reliable products with reduced developmental costs.

There have been a variety of programming languages for the design of real-time systems. Various models and logics

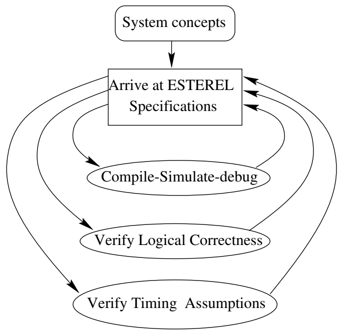


Figure 1. General Methodology of Validation

[2] have been employed to build real time systems and their abstractions to finite-state formalisms for purposes of verification [1]. In classical asynchronous real-time languages, constructs allow us to specify and realize timing constraints of the implemented systems. However, establishing formal correctness has not been easy. On the other hand, family synchronous languages such as Esterel [4] have proven their application for reactive systems. In this paper, we use ESTEREL as the base language for specification and propose a methodology and a tool for the real-time validation of real-time systems. ESTEREL is one of the highly developed languages of the family used extensively in industry that has a powerful programming environment for compiling, simulation, and verification.

A two-step approach under a synchronous framework for the development of real-time systems is briefed 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 synchrony hypothesis and clock-time references.

The general methodology of ESTEREL validation including the proposed time validation is shown in Fig. 1. In this paper, we propose a method and the design of a tool for

*Work was done under project 2202-1 *Formal specification and verification of hybrid and reactive systems* sponsored by the Indo-French Centre for the Promotion of Advanced Research, India. Part of the work was done while visiting Max Planck Institute of Informatics, Saarbrucken.

(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 through an adaptation of *TimeC* developed in [6] (in the context of verifying constraints in compilers meant for ILP processors) and deriving the global constraints for the 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 for ESTEREL programs through an adaptation of *TimeC*.
2. A method of relating timing constraints on the signal environment of ESTEREL for the satisfaction of synchrony assumptions and scheduling constraints.
3. Structure of the tool for the proposed method.

Rest of the paper is organized as follows: After an informal introduction in Section 2, the proposed methodology is informally discussed in Section 3. The general method of validation is described in Section 4. Implementation structure of the tool is described in Section 5 followed by conclusion in Section 6.

2. Background on Esterel

The basic object of ESTEREL without value passing (referred to as Pure ESTEREL) is the signal. Signals are used for communication with the environment as well as for internal communication. The programming unit is the module. A module has an interface that defines its input and output signals and a body that is an executable statement:

```
module M:
   input I1, I2;
   output O1, O2;
   input relations
   statement
end module
```

Input relations can be used to restrict input events and a typical exclusion relation is declared as `relation I1 # I2;` Such a relation states that input events cannot contain `I1` and `I2` together. That is, it is an assertion on the behavior of the asynchronous environment.

At execution time, a module is activated by repeatedly giving it an input event consisting of a possibly empty set of input signals assumed to be present and satisfying the input relations. The module reacts by executing its body and outputs the emitted output signals. We assume that the reaction is *instantaneous* or *perfectly synchronous* in the sense that *the outputs are produced in no time*. Hence, all necessary computations are also done in no time. In Pure ESTEREL these computations are either signal emissions or control transmissions between statements; in full ESTEREL there can be value computations and variable updates as well. The only statements that consume time are the ones explicitly requested to do so. The reaction is also required to be deterministic.

2.1. Statements

ESTEREL has two kinds of statements: the kernel statements, and the derived statements to make the language user-friendly. The list of kernel statements is:

```
nothing
halt
emit S
stat1; stat2
loop stat end
present S then stat1 else stat2 end
do stat watching S
stat1 || stat2
trap T in stat end
exit T
signal S in stat end
```

Kernel statements are imperative in nature, and most of them are classical in appearance. The `trap-exit` constructs form an exception mechanism fully compatible with parallelism. Traps are lexically scoped. The local signal declaration "`signal in` *stat* `end`" declares a lexically scoped signal S that can be used for internal broadcast communication within stat. The `then` and `else` parts are optional in a present statement. If omitted, they are supposed to be `nothing`. For more details on ESTEREL see [3, 4].

Perfect synchrony can be captured as follows:

- Outputs are produced synchronously with inputs.
- Concurrent statements evolve in a tightly coupled way.
- Communication is done by instantaneous broadcasting, the receiver receiving it exactly at the time it is sent.
- There is no explicit clock; for using clocks explicitly, they have to be defined in the environment.

Thus, in Esterel

- The behavior can be treated as an infinite sequence of inputs and outputs called reaction is termed as an *instant*. A reaction takes place on the occurrence of every external event and a special signal *tick*.
- Inputs are indeterminate.

- Reaction to any input is *finite*.

Thus, in synchronous languages, several variables are allowed to change their values in a single step (*i.e.* simultaneity is possible). 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 are considered simultaneous. Obviously there is a need for ensuring that these assumptions are met. Reference to clock-time is required in implementations.

3. An Informal Overview

Some of the distinct advantages of using ESTEREL in the design of embedded systems are:

- Efficient embedded code can be generated.
- Transformed automatically into a circuit or a net list.
- Formal verification of properties can be established.
- Shortened specification and development times.

It is easy to observe that ESTEREL programs can be treated as an executable specification of embedded systems. Our approach is based on using timed annotations in the original ESTEREL program and deriving timing constraints for the satisfaction of the synchrony hypothesis. The method consists of

(i) Annotating ESTEREL fragments with the required timing 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 are labeled with the original signals and the additional logical signals introduced,

(iv) Derive timing constraints to be met for satisfaction of the synchrony hypothesis, and

(v) Verify the validity of the timing constraints.

An overview of the various steps of the method is illustrated through an ESTEREL program shown in Figure 2, used for controlling the path of a boat.

Specifying Timing Constraints: Various time constraints such as maximal, minimal, exact [9] can be specified. The timing constraints are specified through the following steps:

```
        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 **code with location labels**

1. The time constraint to be satisfied by any concrete implementation is specified by first placing *markers* as in *TimeC* [6] at various locations of the code. Note that even though the input and output signals in ESTEREL are assumed to be instantaneous, in practice actuators cause delay in the completion of reaction. These actual timing delays can be specified at the respective control locations through these logical markers in the ideal model.

2. Now the time constraints at the respective locations are specified by specifying the timing relations among the logical markers introduced. For the code fragment shown in Figure 2, *markers* are depicted as labels at various points of code and the constraints to be satisfied are shown in Table 1. Predicate *time(*ℓ*)* indicates the time (global) when the control reaches label ℓ.

Note: It may be noted that the constraints shown in (5) of Table 1 can also be used for specifying explicit clock times relative to the signals used.

Deriving Global Timing Constraints: Having specified the architectural timing constraints, we have to derive all 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

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 to the next iteration

Table 1. Timing Constraints

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 case of ESTEREL which is a *perfectly synchronous language* [3, 4] it is necessary to consider only the interactions of the different execution instants of the body due to the global notion of the instant as highlighted in section 2.1. In other words, the effects are not carried from one instant to another except for the recording of the states.

In view of the above characteristics, in the case of ESTEREL validating the timing constraints corresponds to validating *perfect synchronicity*. This can be established by looking at all the 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. The process of validation is described below:
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. Obtain the global transition system of the program.
3. From the transitions and the synchrony hypothesis, derive the global timing constraints.

4. Constraint Validation in Esterel Programs

We shall discuss the specification and validation of time constraints given that explicit clock constraints can be specified as in [10, 8, 9].

4.1. Specifying Time Constraints

Timed annotation consists of:

1. **Time markers:** The first task for annotations is to locate control points through the introduction of named markers at various control points 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 block of ESTEREL code given below, "block_1_begin" and "block_1_end" denote two time 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.
   (ii) Markers can be nested but does not include any ESTEREL statement that takes time.

2. **Language**: For specifying time constraints among the various markers, we need to relate the various markers relative to clock time in a formal language for expressing timing constraints similar to that given in Table 1. For instance, we could have the following constraints on the markers `block_begin` and `block_end`:
   $\mathtt{time(block\_end)} - \mathtt{time(block\_begin)} \leq 4\mathtt{units}$

   In our tool, we have used a fragment of specification language used in *TimeC* [6]. The actual fragment and its syntax will be given in the full paper.

4.2. Validation of 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 signal emissions relative to time markers. I.e., for signals of the form some_name_begin add the statement `emit some_name_begin` at that point. For signals of the form some_name_end add the statement `emit some_name_end (Amount)` at that point where "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 computable 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 to synchrony hypothesis. Signals are categorized as follows:
    - *Lsignals*: Pure synchronous signals.
    - *Tsignals*: These signals refer to clocks and hence, are time constrained[1].
3. Obtain the reactive automaton for the program.
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, as statements consume time, there is a need to evaluate the time needed for all the micro-steps spread across the modules in the same instant; i.e., one has to arrive at an expression which would clearly define the partial order of steps involved.
5. Finally, *show that the maximum time for any reaction is less than the minimum time between any two instants of the reactive automaton (from synchrony hypothesis).*

For ease of presentation, we first use ESTEREL programs without any local signals.

4.2.1 Validation of Constraints without Local Signals

Steps (1)-(4) should be quite obvious for the class of programs that do not have local signals. The intricacies involved in step (5) are illustrated through the discussion below. Having specified the timing constraints we have to establish the relationship of the timing specifications in the reactive automaton. Under the notion of synchronous observers [5], it easily follows that emitting the signal does not affect the synchrony assumptions.

Consider the following ESTEREL code and its transition diagram shown in Figure 3(A).

```
module P1:
input I;
output O1, O2;
await I;
%# O1_begin
emit O1;
%# O1_end
    %# O2_begin
present I then
    present O1 then emit O2 end
    %# O2_end
  else
    present O2 then emit O1 end
end present
end module
```

Since the two signals appear in one module and emission is in sequence, it can be easily seen that the total time required for the emission of signals O_1 and O_2 is the sum of the time required for individual emission and hence, the expression for the time required for the transition is given by: $\mathtt{time(O1)} + \mathtt{time(O2)}$ In the same way, let module P2 be the same as module P1 except that signals O_1 and O_2 are replaced by O_3 and O_4 respectively. Consider the following program that combines the above modules:

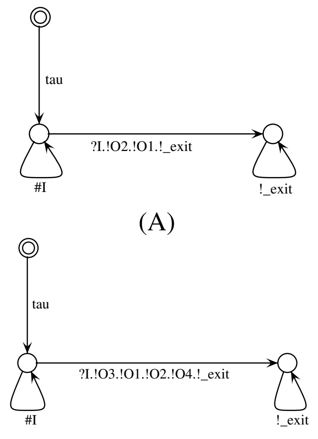
```
module P1:
...
module P2:
...
module P12:
input I;
output O1, O2, O3, O4;
run P1
||
run P2
end module
```

The transition diagram of the combined modules shown in Figure 3(B) displays emission of signals O_1, O_2, O_3 and O_4. Since we know that O_2 follows O_1 and O_4 follows O_3, the total time for the transition wherein signals O1, O2, O3, and O4 are emitted is nothing but the maximum of the time required for the transition of the two modules and the expression is given by:

$max((time(O1) + time(O2)), (time(O3) + time(O4)))$

In other words, the time expression for a transition will be an expression obtained using time-marker signals with operators + (sum or sequence) and $||$ (maximum reflecting concurrent constraint). The actual algorithm of computing the expression is based on the sequentiality or otherwise of the time-marker signals in the transition; this is decided through the control point and transitive information embedded in the time-marker signals. The labeling of control points across modules is done in such a way that it is possible to abstract the sequential order of signals at the micro step level in the global automaton. We adapt the "dot-notation" used for record structures for naming the control points so that a proper order of the actions at the micro-step level can be arrived at. Some of these aspects are illustrated in the following simple examples.

[1] We assume that timing information is carried explicitly in the markers.



(B)

Figure 3. (A) Output signals in the first module (B) Output signals in the combined module

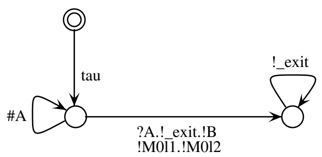


Figure 4. Sequential composition

4.3. Simple statement

```
module one;
    input  A;
    output  B;
    var x: integer in
        await A;
        x:=x+100; %#M0ℓ1
        emit B; %#M0ℓ2
    end var
end module
```

The labels are M0.ℓ_1 and M0.ℓ_2 where M0 is the name of the module (shortened here) and $\ell_1,\ \ell_2$ denote distinct control points in the module. Time for the segments corresponding to the marked parts is given by:
time$(M_0\ \ell_1)$ + time$(M_0\ \ell_2)$

4.4. Signals in Parallel

Labels in the program given below have the form M0$\ell_0(\ell_1)$, M0$\ell_0(\ell_2)$ and M0$\ell_0(\ell_1)$, M0$\ell_0(\ell_1)$ in the two

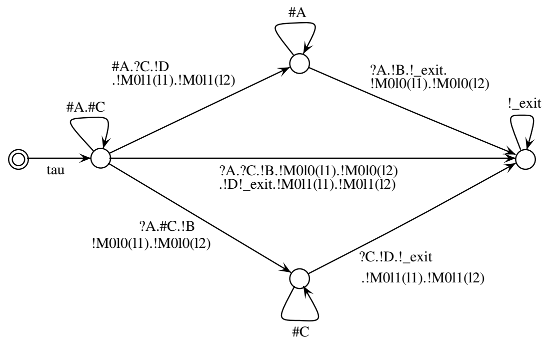


Figure 5. Two signals in parallel

branches of || respectively. Label M0$\ell_0(\ell_1)$ corresponds to M0$\ell_0.\ell_1$ in the dot notation.

```
module one
  input A,C;
  output B,D;
  var x,y:integer in
    [ await A;
      x:=x+30; %#M0ℓ0(ℓ1)
      emit B; %#M0ℓ0(ℓ2)
     ||
      await C;
      y:=y+100;%#M0ℓ1(ℓ1)
      emit D; %#M0ℓ1(ℓ2)
    ]
  end var
end module
```

The time required for the parallel code segments when both of the branches receive and emit signals is given by MAX(x,y), yields the maximum of x,y:

$$\texttt{MAX}\ ((\texttt{time}(M_0\quad \ell_0(\ \ell_1\ ))\ +\ \texttt{time}(M_0\quad \ell_0(\ \ell_2\ ))),$$
$$(\texttt{time}(M_0\quad \ell_1(\ \ell_1\ ))\ +\ \texttt{time}(M_0\quad \ell_1(\ \ell_2\ )))\ )$$

Note that since the emission of signals happens in parallel one has to take the maximum of the two. Note that in the parallel branch, either both of them happen at the same time or the two reactions happen in two distinct instants with the parallel branch terminating in the latter instant. Hence, the maximum of the delays is sufficient.

4.2.2 Validation of Constraints with Local Signals

In the global reactive automaton, the local signals used do not appear in the transitions. For this reason, if there are local signals on which time constraints are to be enforced, it is necessary to add time-markers with the corresponding time quantities so that these constraints are reflected in the automaton for validation. Except for this, the method remains the same as described earlier. The algorithm will also remain the same except for the fact that the labeling of signals has to be done carefully so that there is no clash among names of signals. We have adapted the

SignalConstraints:
time(`beep_end`) - time(`beep_start`) $\leq$ 7 Units;
time(`bad_end`) - time(`bad_start`) $\leq$ 4 Units;
time(`good_end`) - time(`good_start`) $\leq$ 2 Units;
time(`conveyor_end`) - time(`conveyor_start`) $=$ 3 Units;
time(`gotone_end`) - time(`gotone_start`) $\leq$ 5 Units

Figure 6. Constraints for Producer-Consumer

classical "dot-notation" used for record structures for labeling the local-signals on which time constraints are specified.

4.2.3 Validation with Asynchronous Tasks

ESTEREL uses the construct `exec` to interface with asynchronous tasks (cf. [8, 9]). It is assumed that the task does not terminate in the same instant. Informally, a task `T` can be instantiated as follows: (i) a signal `start_T` initiates the task, (ii) waits for reply from task through a signal `reply_T`, and (iii) the module instantiating the task can be killed locally. Thus, the method of arriving at scheduling constraints can be achieved as follows:

1. annotate signals of each task in the source,
2. introduce fresh time-marker signals, and
3. since the ESTEREL program is static, it is possible to arrive at the timing constraints. From such constraints, one can arrive at a spectrum of scheduling strategies.

4.5. Illustrative Example

Consider the producer consumer example shown in Figure 7 and the timing constraints as given in Figure 6.

Consider the transition from vertex S_2 to S_3 shown in Figure 8 that has the following the labels:
$\mathrm{?PRODUCE.!GOOD.!CONVEYOR.!GOTONE.}$
Some of the added signals are shown in bold with integers at the end indicating the time required for response in Figure 7. Let us apply the procedure described earlier relative to the markers associated with the code. Input signals can be recognized by the prefix letter "?" and they carry no other tags. The signals depicting *marked* locations are added concatenated with values denoting the time required for emission for the sake of convenience. Consider the transition from vertex S_2 to S_3 having the following set of labels:

$\mathrm{?PRODUCE.!GOOD2.!GOOD.!}$
$\mathrm{CONVEYOR3.!CONVEYOR.!GOTONE5.!GOTONE}$

Here, $\mathrm{!GOOD2}$ indicates that the output signal GOOD is associated with location `good_start` and the time

```
module PRODUCER:
input PRODUCE, SETUP, START;
output BAD, BEEP, GOOD;
procedure P () ();
var BEEP_COUNT : integer in
  loop
  BEEP_COUNT:=0;
  do
    await START;
    await SETUP;
    every tick do
    %# beep_start
    call P()(); emit BEEP; emit  BEEP7
    BEEP_COUNT:= BEEP_COUNT+1;
    %# beep_end
    end;
  watching PRODUCE
  timeout
    %# bad_start
    if BEEP_COUNT < 4
    then emit BAD ; emit  BAD4;
    %# bad_end
    %# good_start
    else emit GOOD; emit  GOOD2;
    %# good_end
  end
  end
end
end.
  module BUFFER:
    input GOOD;
    output CONVEYOR;
    loop
      await GOOD;
     %# conveyor_start
      emit CONVEYOR; emit  CONVEYOR3;
     %# conveyor_end
  end.
  module CONSUMER:
    input CONVEYOR,FINISHED;
       output GOTONE; emit  GOTOONE5;
    loop
         await CONVEYOR;
      %# gotone_start
         emit  GOTONE;
      %# gotone_end
         await FINISHED
    end.
  module SYSTEM:
    input PRODUCE, SETUP;
    input START,FINISHED;
    output BAD, BEEP, CONVEYOR;
    output GOOD, GOTONE;
    run PRODUCER
         ||
    run BUFFER
         ||
    run CONSUMER
  end module
```

Figure 7. PRODUCER CONSUMER Program

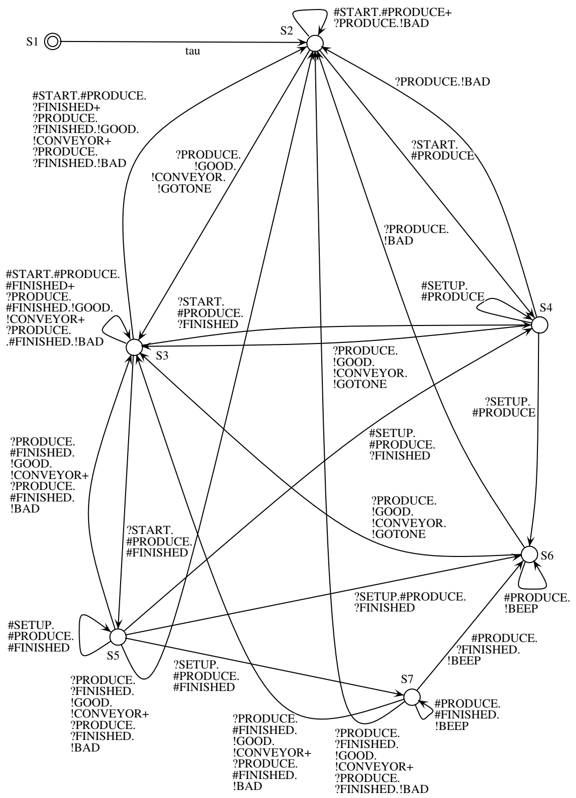


Figure 8. Producer consumer Transitions

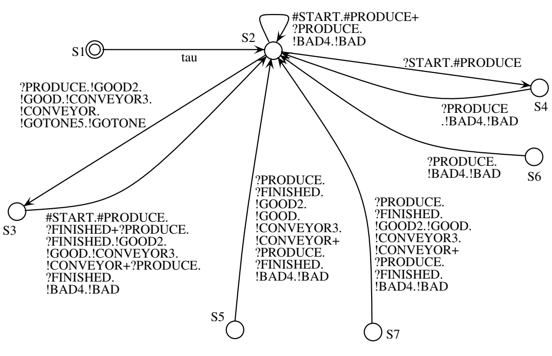


Figure 9. Transitions at S_2

No	**Edge**	Composed Expression	**Time** Units
1	(S_1, S_2)	`tau`	1
2	(S_2, S_2)	`time(BAD)`	4
3	(S_2, S_3)	`max(time(GOOD), time(CONVEYOR), time(GOTONE))`	5
4	(S_3, S_2)	`max(time(GOOD), time(CONVEYOR))`	3
5	(S_2, S_4)	—-	0
6	(S_4, S_2)	`time(BAD)`	4
7	(S_5, S_2)	`max(time(GOOD) time(CONVEYOR))`	3
8	(S_6, S_2)	`time(BAD)`	4
9	(S_7, S_2)	`max(time(GOOD) ,time(CONVEYOR))`	3

Table 2. Computations at vertex S_2

required for this emission is 2 units as specified in the constraints. Similarly signals, $!\mathrm{CONVEYOR3}$ and $!\mathrm{GOTONE5}$, are associated with locations `con-veyor_start` and `gotone_start` respectively and the time required for the emission is 3 units and 5 units respectively as specified in the constraints. The transition diagram of producer consumer example after addition of auxiliary signals concatenated with time in units is shown in Figure 9 for the vertex S_2.

Evaluating the Time required

The time tags attached as above provide the time required for each of the actions; however, from the source code (or locations in modules captured in the tagged signals) we have to derive an expression using these quantities using + (*additive*) and || (*max*) as explained earlier. For instance, the time required for outputting signals GOOD , CONVEYOR and GOTONE in transition from S_2 to S_3 after receiving the input signal PRODUCE is $max(2, 3, 5)$ as the three signals appear after the emission of the three signals from three distinct modules independently. In other words, *the minimum time separation constraint for satisfying the perfect synchrony hypothesis is 5 units.*

For the verification, let us consider vertex S_2. The composed expression for each of transitions from S_2 is shown in table 2. Here, we have assumed that `tau` takes 1 unit of time. Note that the '+' in the label of transition diagram indicates separation of the expressions[2]. The procedure is repeated for all vertices. For checking the satisfaction of the timing constraints, we have to arrive at the minimum separation between any two reactions. The minimum separation

[2]This diagram is produced after reduction in XEVE tool.

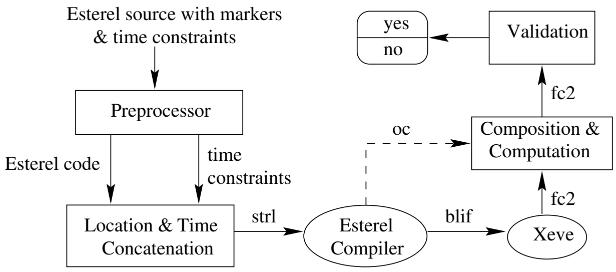


Figure 10. Schematic diagram of the tool

required is found out by finding the maximum time for any of the transitions. *If the* ESTEREL *is fast enough to react at this speed, then we can conclude that the architecture satisfies the timing constraints.*

5. Implementation of the Tool

The structure of the tool for the validation method described is shown in Figure 10. 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 input to the *strl* processor that generates intermediate code by ignoring comments relative to timing specifications.

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.

6 Conclusions

In this paper, we have presented

- a method for validating timing constraints for real-time systems and embedded systems based on the methodology of Esterel.
- the design of a tool for validating the timing constraints

The method assures that the implementation is a realization of the logical specification with the given constraints. It can also be used for arriving at another system from the same logical specification with different timing constraints. That is, the method also can be used to find the minimum speed required by the processor to satisfy the logical specification as well as the timing constraints. The modularity of ESTEREL further allows us validate each module separately and this in turn allows the use of verified modules in arriving at another implementation. As mentioned already, the tool enables the validating of timing constraints and provides a means of interfacing with asynchronous tasks with the underlying scheduling strategies. In fact, systems described in [7] such as digital-copier can be specified and verified in a straight forward way. One of the features of the tool is that it diagnoses the invalidating transitions with pointers to the respective code segments in the ESTEREL source. This would allow refining the specification as required. 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 [9] that unifies the asynchronous and synchronous paradigms; an initial feasibility of the approach is highlighted in [11].

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 & models of real time: A survey. LNCS, 600, pages 74–106. Springer Verlag, 1992.

[3] G. Berry. Constructive semantics of pure ESTEREL, 1996.

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

[5] N. Halbwachs. Synchronous observers and the verification of reactive systems. In *AMAST'93*. Springer Verlag, 1993.

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

[7] M. Rye, J. Park, K. Kim, Y. Seo, and S. Hong. Performance re-engineering of embedded real-time systems. In *Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 80–86. ACM SIGPLAN,34,7, July 1999.

[8] R K Shyamasundar. Specification of hybrid systems in CRP. In *AMAST 93*, pages 227–238. Springer Verlag, 1993.

[9] 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.

[10] R K Shyamasundar. Specifying dynamic Real-Time systems. In 13^{th} *World Computer Congress, IFIP*, pages 75–80. North Holland, 1994.

[11] R K Shyamasundar and J V Aghav. Validating timing constraints in synchrnous language specifications. In *Proc. RTSS 2000, WIP Session*. IEEE Press, 2000.