A Formal Framework for ASTRAL Intralevel Proof Obligations

Alberto Coen-Porisini, *Member, IEEE*, Richard A. Kemmerer, *Senior Member, IEEE*, and Dino Mandrioli

***Abstract*— ASTRAL is a formal specification language for real-time systems. It is intended to support formal software development, and therefore has been formally defined. This paper focuses on how to formally prove the mathematical correctness of ASTRAL specifications. ASTRAL is provided with structuring mechanisms that allow one to build modularized specifications of complex systems with layering. In this paper, further details of the ASTRAL environment components and the critical requirements components, which were not fully developed in previous papers, are presented. Formal proofs in ASTRAL can be divided into two categories: *interlevel* proofs and *intralevel* proofs. The former deal with proving that the specification of level $i+1$ is consistent with the specification of level i, and the latter deal with proving that the specification of level i is consistent and satisfies the stated critical requirements. This paper concentrates on intralevel proofs.**

***Index Terms*—Formal methods, formal specification and verification, real-time systems, timing requirements, state machines, ASLAN, TRIO.**

I. Introduction

ASTRAL is a formal specification language for real-time systems. It is intended to support formal software development, and therefore has been formally defined. Reference [8] discusses the rationale of ASTRAL's design and demonstrates how the language builds on previous language experiments. Reference [9] discusses how ASTRAL's semantics are specified in the TRIO formal real-time logic. It also outlines how ASTRAL specifications can be formally analyzed by translating them into TRIO and then using the TRIO validation theory.

Recently, a number of approaches have been proposed to build formal proofs for real-time systems [1], [2], [5]–[7], [10], [12]. Many of these exploit the so-called dual language approach [10], [11], where a system is modeled as an abstract machine (e.g., a finite state machine or a Petri net) and its properties are described through some assertion language (e.g., a logic or an algebraic language). However, they are based on low-level formalisms, i.e., abstract machines and/or assertion languages that are not provided with modularization and abstraction mechanisms. As a consequence, the proofs lack structure, which makes them unsuitable for dealing with complex real-life systems.

The work of Gerber and Lee [7] provides a layered approach to the verification of real-time systems. With their approach, the CSR application language is used to specify processes, and these processes are mapped to system resources by using a configuration schema. A CSSR specification is then automatically generated. This approach is similar to the ASTRAL to TRIO translation; however, their approach is much more operational than the ASTRAL/TRIO approach.

ASTRAL provides structuring mechanisms that allow one to build modularized specifications of complex systems with layering [8], [9]. In this paper, further details of the ASTRAL environment components and the critical requirements components, which were not fully developed in previous papers, are presented.

Formal proofs in ASTRAL can be divided into two categories: *interlevel* proofs and *intralevel* proofs. The former deal with proving that the specification of level $i+1$ is consistent with the specification of level i, whereas the latter deal with proving that the specification of level i is consistent and satisfies the stated critical requirements. This paper concentrates on intralevel proofs.

In the next section, a brief overview of ASTRAL is presented along with an example system, which is used throughout the remainder of the paper for illustrating specific features of ASTRAL. Section III discusses how to represent assumptions about the environment as well as the representation of critical requirements for the system. Section IV presents a formal framework for generating proof obligations in ASTRAL, and Section V presents an example proof. Finally, in Section VI, some conclusions from this research are presented, and possible future directions are proposed.

II. Overview of ASTRAL

ASTRAL uses a state machine process model and has types, variables, constants, transitions, and invariants. A real-time system is modeled by a collection of state machine specifications and a single global specification. Each state machine specification represents a process type, of which there may be multiple statically generated instances in the system.[1]

Manuscript received August 1993; revised May, 1994. The research by A. Coen-Porisini was supported by Consiglio Nazionale delle Ricerche—Comitato Nazionale per la Scienza e le Tecnologie dell'Informazione. The research by R. A. Kemmerer and D. Mandrioli was partially funded by the National Science Foundation under Grant CCR-9204249, and by the Loral Western Development Laboratories and the University of California through a MICRO grant. Recommended by I. Sommerville.

A. Coen-Porisini and D. Mandrioli are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano, Italy.

R. A. Kemmerer is with the Reliable Software Group—Department of Computer Science, University of California, Santa Barbara, CA 93106 USA; e-mail: kemm@cs.ucsb.edu.

IEEE Log Number 9403573.

[1] Static rather than dynamic processes are used in ASTRAL to simplify both the syntax and semantics of the formalism. Furthermore, most real-life real-time systems avoid dynamic processes. The reader is referred to [8] for more details on the ASTRAL design goals.

0098–5589/94$04.00 © 1994 IEEE

The process being specified is thought of as being in various *states*, with one state differentiated from another by the values of the state *variables*. The values of these variables evolve only via well-defined *state transitions*, which are specified with Entry and Exit assertions and have an explicit non-null duration. State variables and transitions may be explicitly exported by a process. This makes the variable values readable by other processes, and makes the transitions callable by the external environment; exported transitions cannot be called by another process. Interprocess communication occurs via the exported variables, and is accomplished by inquiring about the value of an exported variable for a particular instance of the process. A process can inquire about the value of any exported variable of a process type or about the start or end time of an exported transition.

The ASTRAL computation model views the values of all variables being modified by a transition as being changed by the transition in a single atomic action that occurs when the transition completes execution. Thus, if a process is inquiring about the value of an exported variable while a transition is being executed by the process being queried, the value obtained is the value that the variable had when the transition commenced. Start(Op_i, t) is a predicate that is true if and only if transition Op_i starts at time t and there is no other time after t and before the current time when Op_i starts (i.e., t is the time of the last occurrence of Op_i). For simplicity, the functional notation Start(Op_i) is adopted as a shorthand for "time t such that Start(Op_i, t)," whenever the quantification of the variable t (whether existential or universal) is clear from the context. Start-k(Op_i) is used to give the start time of the kth previous occurrence of Op_i. Inquiries about the end time of a transition Op_i may be specified similarly by using End(Op_i) and End-k(Op_i).

In ASTRAL, a special variable called Now is used to denote the current time. The value of Now is 0 at system initialization time. ASTRAL specifications can refer to the current time ("Now") or to an absolute value for time that must be less than or equal to the current time. That is, in ASTRAL, one cannot express values of time that are to occur in the future. To specify the value that an exported variable var had at time t, ASTRAL provides a past(var, t) function. The past function can also be used with the Start and End predicates. For example, the expression "past(Start(Op), t) $=$ t" is used to specify that transition Op started at time t.

The type ID is one of the primitive types of ASTRAL. Every instance of a process type has a unique identification of type ID. An instance can refer to its own identification by using "Self." For inquiries where there is more than one instance of that process type, the inquiry is preceded by the unique identification of the desired instance followed by a period. For example, i.Start(Op) gives the last start time that transition Op was executed by the process instance whose unique identification is i. However, when the process instance performing the inquiry is the same as the instance being queried, the preceding identification and period may be dropped.

An ASTRAL global specification contains declarations for all of the process instances that comprise the system and for any constants or nonprimitive types that are shared by more than one process type. Globally declared types and constants must be explicitly imported by a process type specification that requires them.

The computation model for ASTRAL is based on nondeterministic state machines and assumes maximal parallelism, noninterruptable and nonoverlapping transitions in a single process instance, and implicit one-to-many (multicast) message-passing communication, which is instantaneous. Maximal parallelism assumes that each logical task is associated with its own physical processor, and that other physical resources used by logical tasks (e.g., memory and bus bandwidth) are unlimited. In addition, a processor is never idle when some transition is able to execute. That is, a transition is executed as soon as its precondition is satisfied (assuming that no other transition is executing). When two or more transitions of the same process are enabled, one of them is nondeterministically chosen for execution.

A detailed description of ASTRAL and of its underlying motivations is provided in [8], which also contains a complete specification of a phone system example. In this paper, only the concepts of ASTRAL that are needed to present the proof theory are discussed in detail. These concepts are illustrated via a simple example that is a variation of the packet assembler described in [13].

> The system contains an object that assembles data items (in the order in which it receives them) into fixed-size packets, and sends these packets to the environment. It also contains a fixed number of other objects, each of which receives data items from the environment on a particular channel and sends those items to the packet maker. The packet maker sends a packet to the environment as soon as it is full of data items.
>
> Each data receiver attaches a channel identifier to each incoming data item; these channel identifiers are included with the data items in the outgoing packets.
>
> If a data receiver does not receive a new datum within a fixed time since the last item arrived, its channel is considered closed until the next datum arrives. Notifications of channel closings are put into the outgoing packets as well as data items. If all channels are closed, then the packet maker should send an incomplete packet to the environment rather than wait for data to complete it.

In the remainder of this paper, this system is referred to as the CCITT system. The Appendix contains a complete ASTRAL specification of the CCITT system.[2] It consists of a packet maker process specification, an input process specification (of which there are N instances), and the global specification.

The input process specification, which corresponds to the data receiver in Zave's system description, contains two variables Msg of type Message and Channel_Closed of type Boolean. It also contains two transitions New_Info and Notify_Timeout, whose duration are N_I_Dur and N_T_Dur, respec-

[2] An earlier version of this specification that did not take into acount the environment, and with different invariants and schedules, was presented in [9].

tively. Transition New_Info, which is exported, prepares a message to be sent to the packet maker process through a channel. The message contains a data part, which is provided by the external environment when the transition is invoked, and two other parts that allow the system to unequivocally identify which instance of process Input has produced that message and how many messages have been produced so far by that particular process instance.

```
TRANSITION New_Info(x:Info)        N_I_Dur
  EXIT
       Msg[Data_Part] = x
   &   Msg[Count] = Msg'[Count] + 1
   &   Msg[ID_Part] = Self
   &   ~Channel_Closed
```

In ASTRAL Exit assertions, variable names followed by a prime (′) indicate the value that the variable had when the transition fired. Transition Notify_Timeout is executed when no datum is received from the external environment for more than Input_Tout time units. It prepares a message to be sent to the packet maker process containing the information that no datum has been received (i.e., the value of the data part is the constant Closed). Moreover, Notify_Timeout marks the channel through which messages are usually sent as being closed.

```
TRANSITION Notify_Timeout     N_T_Dur
  ENTRY
       EXISTS t1: Time (   Start(New_Info, t1)
                         & Now - t1 ≥ Input_Tout)
   &   ~Channel_Closed
  EXIT
       Msg[Data_Part] = Closed
   &   Msg[Count] = Msg'[Count] + 1
   &   Msg[ID_Part] = Self
   &   Channel_Closed
```

The packet maker specification has three variables: Packet and Output of type Message_List, and Previous(Receiver_ID) of type Time. Also, it has two transitions: Process_Msg and Deliver, which correspond to processing a message from an input channel and delivering a packet, respectively. Transition Process_Msg is enabled whenever the packet is not full and either the present message has been produced since the last message from that channel was processed or the value of the current message is Closed and the value of the previously processed message from that channel was not Closed. The result of transition Process_Msg is that the current message from that channel is appended to the packet and the channel's previous processing time is updated to be the current time.

```
TRANSITION Process_Msg(R_id:Receiver_ID)     P_M_Dur
  ENTRY
      LIST_LEN(Packet) < Maximum
  &   ( Receiver[R_id].End(New_Info) > Previous(R_id)
      | (  Receiver[R_id].Msg[Data_Part]=Closed
        & past(Receiver[R_id].Msg[Data_Part],
                                   Previous(R_id)) ≠ Closed ))
  EXIT
      Packet = Packet' CONCAT
                                LIST(Receiver[R_id].Msg)
  &   Previous(R_id) BECOMES Now
```

Transition Deliver is enabled whenever the packet is full or whenever the packet is not empty and Del_Tout time units elapsed since the last packet was output or since system startup time.

```
TRANSITION Deliver     Del_Dur
  ENTRY
       LIST_LEN(Packet) = Maximum
   |   ( LIST_LEN(Packet) > 0
       & ( EXISTS t:Time (   Start(Deliver, t)
                           & Now - t = Del_Tout)
         | Now = Del_Tout - Del_Dur + N_I_Dur))
  EXIT
       Output = Packet'
   &   Packet = EMPTY
```

III. Environmental Assumptions and Critical Requirements

In addition to specifying system *state* (through process variables and constants) and system *evolution* (through transitions), an ASTRAL specification also defines desired system *properties* and *assumptions* on the behavior of the environment that interacts with the system. Assumptions about the behavior of the environment are expressed in environment clauses and imported variable clauses, and desired system properties are expressed through invariants and schedules. Because these components are critical to the ASTRAL proof theory and were not fully developed in previous papers, they are discussed in more detail in this section.

A. Environment Clauses

An *environment clause* formalizes the assumptions that must always hold on the behavior of the environment to guarantee some desired system properties. They are expressed as first-order formulas involving the calls of the exported transitions, which are denoted $\mathrm{Call}(\mathrm{Op}_i)$ (with the same syntactic conventions as $\mathrm{Start}(\mathrm{Op}_i)$). For each process p there is a local environment clause, Env_p, which expresses the assumptions about calls to the exported transitions of process p. There is also a global environment clause, Env_G, which is a formula that may refer to all exported transitions in the system.

In the CCITT example there is a local environment clause for the input process and a global clause. The local clause states that for each input process, the time between two consecutive calls to transition New_Info is not less than the duration of New_Info , and that there will always be a call to New_Info before the timeout expires:

```
      (EXISTS t:Time (Call-2 (New_Info, t))              [Env_in]
         → (Call (New_Info) - Call-2 (New_Info) ≥ N_I_Dur)
  &   (Now ≥ Input_Tout
         → EXISTS t:Time (Call (New_Info, t))
         & Now - Call(New_Info) < Input_Tout)
```

The global environment clause states that exactly N/L calls to transition New_Info are cyclically produced, with time period N/L*P_M_ Dur + Del_Dur (where P_M_Dur is the duration of transition Process_Message; Del_Dur is the duration of Deliver; and L denotes a constant that is used to specify that N/L processes are producing messages).[3]

```
FORALL t:Time (                                  [EnvG]
     t MOD (N/L*P_M_Dur + Del_Dur) = 0
   → EXISTS S: Set_Of_Receiver_ID
     (   |S| = N/L
      & FORALL i:Receiver_ID (i ISIN S
            ↔ Receiver[i].Call(New_Info) = t)))
&  FORALL t:Time (
     t MOD (N/L*P_M_Dur + Del_Dur) ≠ 0
   → FORALL i:Receiver_ID (
        ~Receiver[i].Call(New_Info, t)))
```

B. Imported Variable Clauses

Each process p may also have an *imported variable* clause, IV_p. This clause formalizes assumptions that process p makes about the context provided by the other processes in the system. For example IV_p contains assumptions about the timing of transitions exported by other processes that p uses to synchronize the timing of its transitions. It also contains assumptions about when variables exported by other processes change value. For instance, p might assume that some imported variable changes no more frequently than every 10 time units.

In the CCITT example only the Packet_Maker process has an imported variable clause. It states that the ends of transition New_Info executed by input processes follow the same periodic behavior as the corresponding calls. The clause is similar to the global environment clause.

C. Invariant Clauses

Invariants state properties that must initially be true and must be guaranteed during system evolution, according to the traditional meaning of the term. Invariants can be either local to some process, I_p, or global, I_G. These properties must be true regardless of the environment or the context in which the process or system is running. Invariants are formulas that express properties about process variables and transition timing according to some natural scope rules, which are given in [3].

In the CCITT example the global invariant consists of two clauses. The second clause states that every input data will be output within $H1$ time units after it is input, but not sooner than $H2$ time units.

```
FORALL i:Receiver_ID, t1:Time, x:Info (              [IG]
      t1≤Now - H1
    & past(Receiver[i].End(New_Info(x)),t1) = t1
→   EXISTS t2:Time, k:Integer (
       t2 ≥ t1 + H2 & t2 ≤ Now & Change(Output,t2)
     & 0 < k & k ≤ LIST_LEN(past(Output,t2))
     & past(Output[k][Data_Part],t2)=x
     & past(Output[k][Count],t2) =
                     past(Receiver[i].Msg[Count],t1)
     & past(Output[k][ID_part],t2) = Receiver[i].Id))
```

The other global clause states that no message is output other than those produced by the input processes. The Input process local invariant states that after Input_Tout time units have elapsed without receiving any new message a timeout occurs, and that the last message received is kept until a Deliver timeout occurs.

The Packet_Maker's local invariant states that changes in the exported variable Output occur at, and only at, the end of a Deliver and that no new messages are generated by the packet assembler. It also states that the order that messages appear in an output packet is the order in which they were processed from a channel, this order is preserved across output packets, and every message in Output was previously in Packet and if Output changes Now, then each of the elements of Packet are unchanged from when they were put into the packet. All of the invariants are given in the appendix.

D. Schedule Clauses

Schedules are additional system properties that are required to hold under more restrictive hypotheses than invariants. Unlike invariants, the validity of a schedule may be proved using the assumptions expressed in the associated environment and/or imported variable clauses.

Like invariants, schedules may be either local, Sc_p, or global, Sc_G, and obey suitable scope rules in the same style as invariants. Unlike invariants, however, they may refer to calls to exported transitions. Typically, a schedule clause states properties about the reaction time of the system to external stimuli and on the number of requests that can be "served" by the system. This motivates the term "schedule."

Because there may be several ways to assure that a schedule is satisfied, such as giving one transition priority over another or making additional assumptions about the environment, and because this kind of decision should often be postponed until a more detailed design phase, in ASTRAL the schedules are not required to be proved. It is important, however, to know that the schedule is feasible. That is, it is important to know that if further restrictions are placed on the specification and/or if further assumptions are made about the environment, then the schedule can be met. For this reason, a further assumptions and restrictions clause may be included as part of a process specification. Unlike other components of the ASTRAL specification this clause is only used as guidance to the implementer; it is not a hard requirement. The details of this clause are given in the next subsection.

In the CCITT example the global schedule states that the time that elapses between the call of a New_Info transition and the delivery of the message it produced is equal to N/L*P_M_Dur + N_I_Dur + Del_Dur.

The local schedule for the Input process states that there is no delay between a call of a New_Info transition and the start of its execution. The Packet_Maker's schedule states that the transition Deliver is executed cyclically and that a packet is always delivered with N/L elements.

```
   (EXISTS t:Time (End-2(Deliver,t))             [Scpm]
     →  End(Deliver) - End-2(Deliver) =
                      N/L*P_M_Dur + Del_Dur)
&  FORALL t: Time (past(End(Deliver), t) = t
     →  LIST_LEN(past(Output, t)) = N/L)
```

[3] For simplicity, the traditional cardinality operator, | |, is adopted, even though it is not an ASTRAL operator.

A proof of the Packet_Maker's schedule is presented in Section V.

E. Further Assumptions and Restrictions Clause

As mentioned before, schedules can be guaranteed by exploiting further assumptions about the environment or restrictions on the system behavior. These assumptions constitute a separate part of the process specification, the *further assumptions and restrictions clause*, FAR_p. It consists of two parts: a further environment assumptions section and a further process assumptions section.

The *further environment assumptions* section, FEnv_p, obeys the same syntactic rules as Env_p. It simply states further hypotheses on the admissible behaviors of the environment interacting with the system. Of course, it cannot contradict previous general assumptions on the environment expressed in Env_p and Env_G.

A *further process assumptions* section, FPA_p, restricts the possible system implementations by specifying suitable selection policies in the case of nondeterministic choice between several enabled transitions, TS_p, or by further restricting constants, CR_p. In general, FPA_p reduces the level of nondeterminism of the system specification.

The *transition selection* part, TS_p, consists of a sequence of clauses of the following type:

$$\{\mathrm{OpSet}_i\}\langle\text{Boolean Condition}_i\rangle\{\mathrm{ROpSet}_i\}$$

where

- $\{\mathrm{OpSet}_i\}$ defines a set of transitions.
- $\{\mathrm{ROpSet}_i\}$ defines a restricted but nonempty set of transitions that must be included in the set defined by $\{\mathrm{OpSet}_i\}$.
- $\langle\mathrm{BooleanCondition}_i\rangle$ is a boolean condition on the state of process p.

The operational semantics of the transition selection part is defined as follows.

1) At any given time the set of enabled transitions, $\{\mathrm{ET}\}$, is evaluated by the process abstract machine.
2) Let $\{\mathrm{OpSet}_i\}$, $\langle\text{Boolean Condition}_i\rangle$ be a pair such that ET is $\{\mathrm{OpSet}_i\}$ and $\langle\text{Boolean Condition}_i\rangle$ holds. Notice that such a pair does not necessarily exist.
3) If there are pairs that satisfy condition 2, then the set of transitions that actually are eligible for firing is the union of all $\{\mathrm{ROpSet}_i\}$ corresponding to the above pairs $\{\mathrm{OpSet}_i\}$, $\langle\text{Boolean Condition}_i\rangle$ that are satisfied.
4) If no such pair exists, the set of transitions eligible for firing is $\{\mathrm{ET}\}$.

The *constant refinement* part, CR_p, is a sequence of clauses that may restrict the values that system constants can assume w.r.t. what is stated in the remaining part of the system specification. For example, one can further restrict a constant $T1$ that is bounded between 0 and 100, by stating that $T1$'s value is actually between 10 and 50, or that it is exactly 5.

Notice that the further assumptions and restrictions section can only restrict the set of possible behaviors. That is, if $\{\mathrm{B}\}$ denotes the set of system behaviors that are compatible with the system specification without the FAR clause and $\{\mathrm{RB}\}$ denotes the set of behaviors that are compatible with the system specification including the FAR clause, then it is easy to verify that $\{\mathrm{RB}\}$ is contained in $\{\mathrm{B}\}$.

For the CCITT system two different further assumptions clauses were used with the Packet_Maker process. The first contains both a constant refinement part and a transition selection part. The CR part states that the timeout of transition Deliver is 0 and that the packet length is equal to N/L.

$$\mathrm{Del\_Tout} = 0 \;\&\; \mathrm{Maximum} = N/L$$

The TS part states that the Process_Message transition has higher priority than Deliver.

$$\{\mathrm{Process\_Message}, \mathrm{Deliver}\}\mathrm{TRUE}\{\mathrm{Process\_Message}\}$$

The second further assumptions clause contains only a constant refinement part, which states that Deliver's timeout is N/L*P_M_Dur + Del_Dur and that Maximum = N.

Either of these further assumptions clauses is sufficient to prove that the schedules are met.

IV. Intralevel Proof Obligations in ASTRAL

In this section, the ASTRAL intralevel proof obligations are presented. However, it is first necessary to present some notation.

Let S denote a top level ASTRAL specification. S is composed of a set of process specifications P_p and a global specification G. Each P_p, in turn, is composed of a set of transitions $\mathrm{Op}_{p1},\cdots,\mathrm{Op}_{pn}$, a local invariant I_p, a local schedule Sc_p a local environment Env_p, imported variable assumptions IV_p, a further local environment FEnv_p, a further process assumption FPA_p and an initial clause $\mathrm{Init\_State}_p$. Moreover, every transition Op_{pj} is described by entry and exit clauses denoted EN_{pj} and EX_{pj}, respectively. The global specification G is made up of a global invariant I_G, a global schedule Sc_G and a global environment Env_G clause.

Proving that S satisfies its critical requirements can be partitioned into the following proof obligations:

1) Every process specification P_p guarantees its local invariant I_p;
2) Every process specification P_p guarantees its local schedule Sc_p;
3) The specification S guarantees the global invariant I_G;
4) The specification S guarantees the global schedule Sc_G.

For soundness the following proof obligations are also needed:

5) The imported variable assumptions IV_p are guaranteed by the specification S
6) All the assumptions about the environment (Env_G, Env_p and FEnv_p) are consistent.

In what follows a formal framework for these proof obligations is presented.

A. Axiomatization of ASTRAL Abstract Machine

An informal description of the ASTRAL computational model is given in [8], [9]. However, a formal description of the ASTRAL abstract machine is needed to carry out the ASTRAL proofs.

The semantics of the ASTRAL abstract machine is defined by three axioms. The first axiom states that the time interval spanning from the starting to the ending of a given transition is equal to the specified duration of the transition.

```
FORALL t:Time, Op: Trans_of_p (                    [A1]
    Now - t ≥ TOp
  → (  past(Start(Op),t) = t
     ↔ past(End(Op),t+TOp) = t +TOp))
```

where T_{Op} represents the duration of transition Op.

The second axiom states that if a processor is idle and some transitions are enabled then one transition will fire. Let S_T denote the set of transitions of process p.

```
FORALL t:Time (                                    [A2]
  EXISTS d: Time, S'T: SET OF Trans_of_p(
    FORALL t1:Time, Op: Trans_of_p (
        t1 ≥ t - d & t1 < t & Op ISIN ST
    &   past(Start(Op),t1) < past(End(Op),t)
    &   S'T ⊆ ST & S'T ≠ EMPTY
    &   FORALL Op':Trans_of_p (
           Op' ISIN S'T → Eval_Entry(Op',t))
    &   FORALL Op':Trans_of_p (
           Op' ~ISIN S'T → ~Eval_Entry(Op',t))
    →   UNIQUE Op':Trans_of_p (
           Op' ISIN S'T & past(Start(Op'),t)=t)))),
```

where Eval_Entry(Op, t) is a function that given a transition Op and a time instant t evaluates the entry condition $\mathrm{EN}_{\mathrm{Op}}$ of transition Op at time t.

Because the ASTRAL model implies that the starting time of a transition equals the time in which its entry condition was evaluated, the Eval_Entry function is introduced to prevent the occurrence of a contradiction. More specifically, when the entry condition of transition Op refers to the last start (2nd last, etc) of itself, the evaluation at time t of Start(Op) in the entry condition should refer to the value of Start immediately before the execution of Op at time t. Since Op has a non-null duration this can be expressed by evaluating Start(Op) at a time t' which is prior to t and such that transition Op has not fired in the interval $[t', t)$.

Finally, the third axiom states that for each processor the transitions are nonoverlapping.

```
FORALL t1, t2:Time, Op: Trans_of_p (               [A3]
      Start(Op)=t1 & End(Op)=t2 & t1 < t2
  →  FORALL t3: Time, Op': Trans_of_p (
          t3 ≥ t1 & t3 < t2 & Start(Op')= t3
      →  Op = Op' & t3 = t1)
  &  FORALL t3: Time,Op': Trans_of_p (
          t3 > t1 & t3 ≤ t2 & End(Op')= t3
      →  Op = Op' & t3 = t2))
```

B. Local Invariant Proof Obligations

The local invariant I_p represents a property that must hold for every reachable state of process p. Furthermore, the invariant describes properties that are independent from the environment. Therefore, the proof of the invariant I_p may not make use of any assumption about the environment, imported variables or the system behavior as described by Env_p, FEnv_p, IV_p and FPA_p.

To prove that the specification of process p guarantees the local invariant one needs to show that:

1) I_p holds in the initial state of process p, and
2) If p is in a state in which I_p holds, then for every possible evolution of p, I_p will hold.

The first proof consists of showing that the following implication is valid:

$$\mathrm{Init\_State}_p \ \& \ \mathrm{Now} = 0 \rightarrow I_p$$

To carry out the second proof one assumes that the invariant I_p holds until a given time t_0 and proves that I_p will hold for every time $t > t_0$. Without loss of generality, one can assume that t is equal to $t_0 + \Delta$, for some fixed Δ greater than zero, and show that the invariant holds until $t_0 + \Delta$.

In order to prove that I_p holds until time $t_0 + \Delta$ it may be necessary to make assumptions on the possible sequences of events that occurred within the interval $[t_0 - H, t_0 + \Delta]$, where H is a constant a *priori* unbounded, and where by event is meant the *starting* or *ending* of some transition Op_{pj} of process p.

Let σ denote one such sequence of events. A formula F_σ describing the sequence of events that belong to σ can be algorithmically generated from σ. For each event occurring at time t one has:

Eval_Entry(Op_{pj}, t) & past(Start(Op_{pj}, t), t) if the event is the start of Op_{pj} or

past(EX_{pj}, t) & past(End(Op_{pj}, t), t) if the event is the end of Op_{pj}.

F_σ is the logical conjunction of all such predicates. Then the prover's job is to show that for any σ:

```
A1 & A2 & A3 ⊢
      Fσ & FORALL t:Time (t ≤ t0 → past(Ip,t))
   → FORALL t1:Time (    t1 > t0 & t1 ≤ t0 + Δ
                         → past(Ip, t1))
```

Notice that as a particular case, the implication is trivially true if F_σ is contradictory, since this would mean that σ is not feasible.

C. Local Schedule Proof Obligations

The local schedule Sc_p of a process p describes some further properties that p must satisfy when the assumptions on the behavior of both the environment and p hold (i.e., Env_p, IV_p, FEnv_p and FPA_p).

To prove that the specification of process p guarantees the local schedule Sc_p it is necessary to show that:

1) Sc_p holds in the initial state of process p, and
2) If p is in a state in which Sc_p holds, then for every possible evolution of p compatible with FPA_p, when the environment behavior is described by Env_p and FEnv_p, and the imported variables behavior is described by IV_p, Sc_p will hold.

Note that one can also assume that the local invariant I_p holds; i.e., I_p can be used as a lemma. The initial state proof obligation is similar to the proof obligation for the local invariant case; however the further hypothesis on the values of some constants expressed by CR_p can be used:

$$\mathrm{Init\_State}_p \ \& \ \mathrm{Now} = 0 \ \& \ \mathrm{CR}_p \rightarrow \mathrm{Sc}_p$$

The second proof obligation is also similar to the local invariant proof. However, in this case events may be external calls of exported transitions Op_{pj} in addition to the starting and ending of all transitions of p. If the event is the call of Op_{pj} from the external environment, then "past(Call(Op_{pj}), t) $= t$" can be used to represent that transition Op_{pj} was called at time t.

The prover's job is to show that for any σ:

$$\begin{aligned}&\mathbf{A1 \ \& \ A2' \ \& \ A3 \ \& \ A4 \ \& \ Env_p \ \& \ FEnv_p \ \& \ IV_p} \vdash\\ &\quad \mathbf{CR_p \ \& \ F_\sigma \ \& \ FORALL \ t{:}Time \ (\quad t \leq t_0}\\ &\qquad\qquad\qquad\qquad \mathbf{\rightarrow past(Sc_p, t))}\\ &\mathbf{\rightarrow FORALL \ t_1{:}Time \ (\quad t_1 > t_0 \ \& \ t_1 \leq t_0 + \Delta}\\ &\qquad\qquad\qquad\qquad \mathbf{\rightarrow past(Sc_p, t_1))}\end{aligned}$$

where A2′ and A4 are defined in what follows.

A2′ is an axiom derived from A2 by taking into account the TS_p section, which restricts the non-determinism of the machine, and the fact that the exported transitions can fire only if they are called by the environment.

The TS_p section can be viewed as the definition of a function TS: $2^{\{\mathrm{Op1},\cdots,\mathrm{Opn}\}} \rightarrow 2^{\{\mathrm{Op1},\cdots,\mathrm{Opn}\}}$, having as domain and range the powerset of the transitions of process p. Its semantics is the following: denoting with ET the set of enabled transitions then TS(ET) returns a restricted set of enabled transitions, ET′, where ET′ $\subseteq$ ET. The processor will nondeterministically select which transition to fire from the transitions in ET′.

Let ST denote the set of transition of process p:

```
FORALL t:Time (                                           [A2']
  EXISTS d: Time, S'T: SET OF Trans_of_p (
    FORALL t1:Time, Op: Trans_of_p (
        t1 ≥ t - d & t1 < t & Op ISIN ST
      & past(Start(Op),t1) < past(End(Op),t)
      & S'T ⊆ ST & S'T ≠ EMPTY
      & FORALL Op':Trans_of_p (
            Op' ISIN S'T → Eval_Entry'(Op',t))
      & FORALL Op':Trans_of_p (
            Op' ~ISIN S'T → ~Eval_Entry'(Op',t))
      → UNIQUE Op':Trans_of_p (
            Op' ISIN TS(S'T) & past(Start(Op'),t)=t))),
```

where Eval_Entry′ (Op′, t). = Eval_Entry(Op′,t) & Issued_call(Op′), iff Op′ is exported and Eval_Entry′(Op′, t)= Eval_Entry(Op′,t), iff Op′ is not exported.

A4 states that Issued_call(Op) is true iff the environment has called transition Op and transition Op has not fired since then:

```
FORALL Op: Trans_of_p (                                   [A4]
    EXISTS t1: Time (
        t1 ≤ Now & Call(Op, t1)
      & FORALL t: Time (
            t ≥ t1 & t ≤ Now & ~Start(Op,t)
          → past(Issued_call(Op),t)))
  & EXISTS t1: Time (
        t1 ≤ Now & Start(Op, t1)
      & FORALL t: Time (
            t > t1 & t ≤ Now & ~Call(Op,t)
          → ~past(Issued_call(Op),t)))).
```

D. Global Invariant Proof Obligations

Given an ASTRAL specification S composed of n processes, the state of S can be defined as the tuple $\langle s_1, \cdots, s_n\rangle$, where s_p represents the state of process p. The global invariant I_G of S describes the properties that must hold in every state of S.

To Prove that I_G is guaranteed by S it is necessary to prove that:

1) I_G holds in the initial state of S, and
2) If S is in a state in which I_G holds, then for every possible evolution of S, I_G will hold.

Since the initial state of S is the tuple $\langle \mathrm{Init\_State}_1, \cdots, \mathrm{Init\_State}_n\rangle$, where each $\mathrm{Init\_State}_p$ is a formula describing the initial state of process p, to prove point 1 one needs to prove the validity of the following logical implication:

$$\bigwedge_{p=1}^{n} (\mathrm{Init\_State}_p) \quad \& \quad \mathrm{Now} = 0 \rightarrow I_G$$

Point 2 can be proved in a manner very similar to the local invariant case. However in this case the sequences of events σ will contain starting and ending events for exported transitions belonging to any process of S. Moreover, the local invariant of each process p composing S can be used to prove that every σ preserves the global invariant.

The prover's job is to show that for any σ:

$$\begin{aligned}&\mathbf{A1 \ \& \ A2 \ \& \ A3} \vdash\\ &\quad \mathbf{F_\sigma \ \& \ FORALL \ t{:}Time \ (t \leq t_0 \rightarrow past(I_G, t))}\\ &\mathbf{\rightarrow \ FORALL \ t_1{:}Time \ (\quad t_1 > t_0 \ \& \ t_1 \leq t_0 + \Delta}\\ &\qquad\qquad\qquad \mathbf{\rightarrow past(I_G, t_1))}\end{aligned}$$

Notice that unlike local proofs, for global proofs it may happen that a sequence σ contains contemporary events. More precisely two sequences σ_1 and σ_2 may differ only in the order of some events that occur at the same time. In this case, anyone of the sequences can be chosen since the associated σ's are obviously logically equivalent.

E. Global Schedule Proof Obligations

The global schedule Sc_G of the specification S describes some further properties that S must satisfy, when all its processes satisfy their own schedules and the assumptions on the behavior of the global environment hold.

Thus, to prove that Sc_G is consistent with S one has to show that:

1) Sc_G holds in the initial state of S, and
2) If S is in a state in which Sc_G holds, then for every possible evolution of S, Sc_G will hold.

In both proofs one can assume that the global invariant I_G and every local invariant I_p and local schedule Sc_p holds as well as the global environment assumptions Env_G. Note that none of the local environment assumption (Env_p and $FEnv_p$) may be used to prove the validity of the global schedule.

The first proof requires the validity of the formula:

$$\bigwedge_{p=1}^{n} (\mathrm{Init\_State}_p) \quad \& \quad \mathrm{Now} = 0 \quad \& \quad \mathrm{Env}_G \rightarrow \mathrm{Sc}_G$$

The second proof requires the construction of the sequences of events σ. Each σ will contain calling, starting and ending of exported transitions belonging to any process p of S. The prover's job is to show that for any σ:

```
A1 & A2" & A3 & A4 & EnvG ⊢
      Fσ & FORALL t:Time (t ≤ t0 → past(ScG, t))
  → FORALL t1:Time (  t1 > t0 & t1 ≤ t0 + Δ
                          → past(ScG, t1))
```

where A2″ is an axiom derived from A2 by taking into account that the exported transitions can fire only if they are called by the environment.

```
FORALL t:Time (                                    [A2"]
  EXISTS d: Time, S'T: SET OF Trans_of_p (
    FORALL t1:Time, Op: Trans_of_p (
        t1 ≥ t - d & t1 < t & Op ISIN ST
    &  past(Start(Op),t1) < past(End(Op),t)
    &  S'T ⊆ ST & S'T ≠ EMPTY
    &  FORALL  Op':Trans_of_p(
               Op' ISIN S'T → Eval_Entry'(Op',t))
    &  FORALL  Op':Trans_of_p (
               Op' ~ISIN S'T → ~Eval_Entry'(Op',t))
    →  UNIQUE  Op':Trans_of_p (
               Op' ISIN S'T & past(Start(Op'),t)=t))))
```

where Eval_Entry′(Op′,t) = Eval_Entry(Op′,t) & Issued_call(Op′), iff Op′ is exported and Eval_Entry′(Op′, t) = Eval_Entry(Op′, t), iff Op′ is not exported.

F. Imported Variable Proof Obligation

When proving the local schedule of a process p one can use the assumptions about the imported variables expressed by IV_p. Therefore, these assumptions must be checked against the behavior of the processes from which they are imported.

The proof obligation guarantees that the local environment, local schedule and local invariant of every process of S (except p), and the global environment, invariant and schedule imply the assumptions on the imported variables of process p:

$$\mathrm{A1} \;\&\; \mathrm{A2} \;\&\; \mathrm{A3} \;\&\; \bigwedge_{i \neq p} \mathrm{Env}_i \& \bigwedge_{i \neq p} I_i \& \bigwedge_{i \neq p} \mathrm{Sc}_i$$
$$\& \;\; \mathrm{Env}_G \;\&\; I_G \;\&\; \mathrm{Sc}_G \rightarrow \mathrm{IV}_p$$

G. Environment Consistency Proof Obligation

Every process p of S may contain two clauses describing assumptions on the behavior of the external environment, Env_p and $FEnv_p$. These clauses are used to prove the local schedule of p. The global specification also contains a clause describing assumptions on the system environment behavior Env_G.

For soundness, it is necessary to verify that none of the environmental assumptions contradict each other, i.e., that a behavior satisfying the global as well as the local assumptions can exist. This requires proving that the following formula is satisfiable:

$$\bigwedge_{i=1}^{n} \mathrm{Env}_i \;\&\; \bigwedge_{i=1}^{n} \mathrm{FEnv}_i \;\&\; \mathrm{Env}_G.$$

V. An Example Correctness Proof in ASTRAL

In this section the proof of the local schedule of process Packet_Maker is considered:

```
Scpm:
    EXISTS t:Time (End-2(Deliver, t ))
        → End(Deliver) - End-2(Deliver)
                      = N/L*P_M_Dur + Del_Dur
&   FORALL t:Time (past(End(Deliver),t) = t
        → LIST_LEN(past(Output,t)) = N/L)
```

To prove Sc_{pm} the imported variables assumptions IV_{pm} and the second further process assumptions, FPA_{pm}, of process Packet Maker are used:

```
IVpm:
    FORALL t:Time (
        (t - N_I_Dur) MOD (N/L*P_M_Dur + Del_Dur) = 0
      → EXISTS S:Set_of_Receiver_ID (|S| = N/L
          & FORALL i:Receiver_ID (
                i ISIN S ↔ Receiver[i].End(New_Info) = t)))
  & FORALL t:Time (
        (t - N_I_Dur) MOD (N/L*P_M_Dur + Del_Dur) ≠ 0
      → FORALL i:Receiver_ID (
            ~Receiver[i].End(New_Info) = t))
  & FORALL i:Receiver_ID (
        Receiver[i].Msg[Data_Part] ≠ Closed)
```

FPA_{pm} :

$$\mathrm{Del\_Tout} = N/L * \mathrm{P\_M\_Dur} + \mathrm{Del\_Dur} \;\&\; \mathrm{Maximum} = N$$

Consider a time instant p_0 such that Sc_{pm} holds until p_0; it is necessary to prove that Sc_{pm} holds until $p_0 + \Delta$, where Δ is big enough to require an End(Deliver) to occur within $(p_0, p_0 + \Delta]$. Without loss of generality, assume that:

1) at time p_0 transition Deliver ends and
2) $\Delta = N/L*\mathrm{P\_M\_Dur} + \mathrm{Del\_Dur}$.

Now, by [A1] one can deduce that at time $p_0 - \mathrm{Del\_Dur}$ a Start(Deliver) occurred. Fig. 1 shows the relevant events for the discussion that follows on a time line.

The Entry assertion of Deliver states that Deliver fires either when the buffer is full or when the timeout expires and at least one message has been processed.

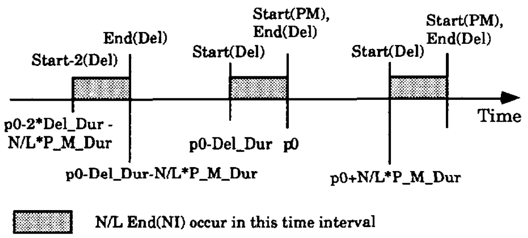


Fig. 1. Example time line.

```
EnDel:
  LIST_LEN(Packet) = Maximum
 | ( LIST_LEN(Packet)>0
   & ( EXISTS t: Time (
                  Start(Deliver,t ) & Now - t ≥ Del_Tout)
     | Now = Del_Tout + N_I_Dur - Del_Dur))
```

Because Sc_{pm} holds until p_0 and from the Exit assertion for Deliver it is known that:

1) For all t less than or equal to p_0 and such that an end of transition Deliver occurred, Output contains N/L messages at time t (Sc_{pm}), and
2) The content of Output at the end of Deliver is equal to the content of Packet at the beginning of Deliver (Exit assertion of Deliver).

From this one can conclude that at time t − Del_Dur the buffer contained N/L messages (i.e., it was not full). As a consequence *transition Deliver has fired because the timeout has expired.*

Furthermore, assume as lemma $L1$ that Process_Message is disabled every time Deliver fires (this lemma will be proved later).

The Entry condition of Process_Message is:

```
   LIST_LEN(Packet) < Maximum
&  ( EXISTS t1:Time (
          Receiver[R_id].End(New_Info) = t1
        & t1 > Previous(R_id))
   | (   Receiver[R_id].Msg[Data_Part] = Closed
      &  past(Receiver[R_id].Msg[Data_Part],
                         Previous(R_id)) ≠Closed))
```

and since

1) the buffer is not full (Sc_{pm}), and
2) no notification of closed channel can arrive (IV_{pm})

one can conclude that *no new message is available when Deliver fires* ($L1$).

IV_{pm} states that N/L messages are received every N/L∗P_M_Dur + Del_Dur time units. As a consequence:

1) the N/L messages output at time p_0 have been received before time p_0 − Del_Dur − N/L∗P_M_Dur, in order to allow Process_Message to process each of them, and
2) they have been received after the second last occurrence of Delivery prior to p_0 (because of $L1$)

Thus, one can conclude that the N/L messages output at time p_0 have been received in the interval:

(Start-2(Deliver), p_0− Del_Dur −N/L∗P_M_Dur],

that is,

(p_0 − 2*Del_Dur − N/L∗P_M_Dur,
p_0 − Del_Dur − N/L∗P_M_Dur]

because of Sc_{pm}. As a consequence of IV_{pm}, N/L new messages will arrive after N/L∗P_M_Dur + Del_Dur time units from the last arrival, i.e., in the interval (p_0−Del_Dur, p_0].

Thus, at time p_0 Process_Message will become enabled and the N/L messages will be processed within time p_0 + N/L∗P_M_Dur, since Deliver is disabled until that time. Moreover, at time p_0 + N/L∗P_M_Dur Process_Message will be disabled, since there are exactly N/L messages to process.

Thus, at time p_0 + N/L∗P_M_Dur the buffer contains N/L messages and Deliver fires because the timeout has expired. Also, at time p_0 + N/L∗P_M_Dur + Del_Dur, Deliver ends and the length of the Output buffer will be equal to N/L (Exit clause of Deliver). Therefore, the schedule will hold until time p_0 + N/L∗P_M_Dur + Del_Dur.

To complete the proof it is necessary to prove lemma $L1$, which states that Process_Message is disabled every time Deliver fires. The proof is carried out by induction in what follows.

Initially, the first time that Deliver fires, Process_Message is disabled. In fact, the first N/L End(New_Info) occur at time N_I_Dur (IV_{pm}). Transition Process_Message will finish processing these messages at time N_I_Dur +N/L∗P_M_Dur, and at that time Deliver will become enabled.

Since no End(New_Info) can occur in (N_I_Dur, N_I_Dur +N/L∗P_M_Dur + Del_Dur) (by IV_{pm}), then at time N_I_Dur +N/L∗P_M_Dur transition Process_Message is disabled and Deliver fires.

Now suppose that when Deliver fires Process_Message is disabled; it is necessary to prove that Process_Message is again disabled the next time Deliver fires.

Let q_0 be the time when Deliver starts; by hypothesis at time q_0 Process_Message is disabled. As a consequence the messages in Packet at time q_0 have been received in the interval (q_0−Del_Dur −N/L∗P_M_Dur, q_0 − N/L∗P_M_Dur] (Sc_{pm}).

Thus, by IV_{pm} the next N/L messages will arrive in the interval (q_0, q_0 + Del_Dur]. Furthermore, the timeout for Deliver will expire at time q_0 + N/L∗P_M_Dur + Del_Dur. Therefore, Deliver cannot fire before that time unless the buffer is full.

At time q_0 + Del_Dur Process_Message will become enabled, and it will fire until either all messages have been processed or the buffer becomes full. At time q_0 + Del_Dur +N/L∗P_M_Dur the N/L messages that arrived in the interval (q_0, q_0 + Del_Dur] will be processed, and since no new message can arrive before q_0 + Del_Dur + N/L∗P_M_Dur at that time Process_Message will be disabled. Similarly, at that time Deliver will be enabled and thus will fire.

This completes the proof of lemma $L1$ and thus the proof of Sc_{pm}.

VI. Conclusion and Future Directions

In this paper, the environment and critical requirements clauses, which were only briefly sketched in previous papers, were presented in detail. The intralevel proof obligations were also presented and an example proof was demonstrated.

All of the proofs for the CCITT specification have been completed. In addition, the proofs of five different schedules that can be guaranteed by using different further assumptions clauses have also been completed. The proofs of these schedules did not require any new or changed invariants. The CCITT proofs demonstrate that formal correctness analysis can be applied to complex real-time systems by suitably structuring both the specifications and the proofs.

Normal correctness proofs are probably the most advanced and critical application of formal methods to software construction. In any proof within an undecidable theory a "creative" part cannot be avoided. For instance, in the proof of traditional sequential programs, this part typically consists of the invention of suitable invariants. The difficult part of the ASTRAL proofs is choosing the appropriate event sequences and showing that all of the possible event sequences are included in the set of sequences chosen. This is essentially due to the fact that most often the desired properties of reactive systems are of the type "as a consequence of event A, event B must occur within Δ time units (or not before Δ time units)". Thus, the sequencing of events becomes a central issue. Our limited experience, however, showed that in all practical examples considered so far, the "shape" of the event sequences to be analyzed were always quite similar to the sequences of the CCITT example presented in this paper. The examples investigated include a phone switching system, a traffic light system, a timed light switching system, along with five different versions of the global schedule of the CCITT example. Thus, this similarity may considerably reduce the amount of ingenuity necessary to carry out ASTRAL proofs, after an initial experience with some sample systems.

The interlevel proofs for the CCITT specifications have also been completed. The details of these proofs as well as the complete two-level CCITT specification can be found in [4]. In that paper, the details of the implementation mappings and the refinement of process specifications are also discussed.

Future work will concentrate on applying ASTRAL to more varied and complex real-time systems. Work will also continue on building a tool suite for formally designing real-time systems using ASTRAL.

Appendix
ASTRAL Formal Specification for the CCITT System

```
GLOBAL Specification CCITT

PROCESSES
   Receiver: array [1..N] of Input,
   Assembler: Packet_Maker

TYPE
   Data,
   Message IS STRUCTURE OF
    (Data_Part: Data,
     Count: Integer,
     ID_Part: ID),
   Message_List IS LIST OF Message,
   Pos_Integer: TYPEDEF i: Integer (i > 0),
   Receiver_ID: TYPEDEF i:Pos_Integer (i ≤ N),
   Set_Of_Receiver_ID IS SET OF Receiver_ID,
   Info: TYPEDEF D:Data (D ≠ Closed)

CONSTANT
   N, L: Pos_Integer,
    /*N denotes the number of processes of type Input,
    L denotes a value such that the number of input
    processes producing messages at the same time is N/L*/
   Closed: Data,
   N_I_Dur, P_M_Dur, Del_Dur: Time
    /*These are the duration for transitions New_Info,
    Process_Message, and Deliver*/
   H1, H2: Time
    /*H1, H2 are lower and upper bounds on the time
    for an input to be output*/

AXIOM
  N MOD L = 0

DEFINE
  Change(L_Msg:Message_List,t:Time):Boolean ==
   EXISTS e: Time (
      e > 0 & e ≤ t
    & FORALL d: Time (
          d ≥ t - e & d < t
       →  past(L_Msg, d) ≠ past(L_Msg,t)))

ENVIRONMENT
     /*The environment cyclically produces exactly
     N/L messages every N/L*P_M_Dur + Del_Dur
     time units*/
   FORALL t:Time (
        t MOD (N/L*P_M_Dur + Del_Dur) = 0
     →  EXISTS S: Set_Of_Receiver (
              |S| = N/L
          &  FORALL i:Receiver_ID (
                (  i ISIN S
                ↔ Receiver[i].Call(New_Info) = t)))
  & FORALL t:Time (
        t MOD (N/L*P_M_Dur + Del_Dur) ≠ 0
     →  FORALL i:Receiver_ID  (
             ~Receiver[i].Call(New_Info, t)))

INVARIANT
    /* Every data output was received sometime in the past */

   FORALL k:Integer (
        k > 0 & k ≤ LIST_LEN(Output)
     & Output[k] [Data_Part] ≠ Closed
   → EXISTS i:Receiver_ID, t:Time, j:Integer (
         t < Now
      & Receiver[i].Start-j(New_Info(Output[k]
                                       [Data_Part]) = t))

&     /* Every input data will be output within H1 time units after
      it is input, but not sooner than H2 time units*/

   FORALL i:Receiver_ID, t1:Time, x:Info (
        t1≤Now-H1
      & past(Receiver[i].End(New_Info(x)),t1) = t1
   → EXISTS t2:Time, k:Integer (
          t2 ≥ t1+H2 & t2 ≤ Now & Change(Output,t2)
       &  0 < k & k ≤ LIST_LEN(past(Output,t2))
       &  past(Output[k][Data_Part],t2)=x
       &  past(Output[k][Count],t2) =
                past(Receiver[i].Msg[Count],t1)
       &  past(Output[k][ID_part],t2) = Receiver[i].Id))
```

```
SCHEDULE

  /*The time that elapses between the call of a New_Info transition and
  the delivery of the message it produced is equal to N/L*P_M_Dur +
  N_I_Dur + Del_Dur*/

 FORALL i:Receiver_ID, t1:Time, x:Info (
      t1 ≤ Now - N/L*P_M_Dur - N_I_Dur - Del_Dur
    & past(Receiver[i].Call(New_Info(x)),t1)=t1
 → EXISTS t2:Time, k:Integer (
       t2 = t1 + N/L*P_M_Dur + N_I_Dur + Del_Dur
     & 0 < k & k ≤ LIST_LEN(past(Output,t2))
     & Change(Output,t2)
     & past(Output[k][Data_Part],t2)=x
     & past(Output[k][ID_Part],t2)
              = Receiver[i].Id))

END CCITT


SPECIFICATION Input
LEVEL Top_Level


IMPORT
   Data, Message, Info, Closed, N_I_Dur


EXPORT
   New_Info, Msg


VARIABLE
   Msg: Message,
   Channel_Closed:Boolean


CONSTANT
   Input_Tout, N_T_Dur: Time


ENVIRONMENT

    (EXISTS t:Time (Call-2 (New_Info, t))
     → Call (New_Info) - Call-2 (New_Info) ≥ N_I_Dur)
  & (Now ≥ Input_Tout
      →  EXISTS t:Time (Call (New_Info, t))
       & Now - Call(New_Info) < Input_Tout)


INITIAL
       ~Channel_Closed
     & Msg[Data_Part] ≠ Closed
     & Msg[Count]=0


INVARIANT

  /* After Input_Tout time units have elapsed without receiving any new
  message a timeout occurs */

  FORALL t1: Time (
       Start(New_Info, t1) & Now - t1 > Input_Tout
    → EXISTS t2: Time (
          Start(Notify_Timeout, t2)
        & t2 = t1 + Input_Tout))

&    /* The last received message is kept until a timeout occurs */

  FORALL t1: Time, x: Info (
        End(New_Info(x), t1)
      & Now - t1 < Input_Tout - N_I_Dur + N_T_Dur
         → Msg[Data_part] = x)
    & (  End(New_Info(x), t1)
      & Now - t1 ≥ Input_Tout - N_I_Dur + N_T_Dur
         → Msg[Data_part] = Closed))



SCHEDULE
  FORALL t: Time, x: Info (
   t ≤ Now → ((Call(New_Info(x)) = t) ↔ Start(New_Info(x)) = t))



TRANSITION New_Info(x:Info)  N_I_Dur
 EXIT
      Msg[Data_Part] = x
    & Msg[Count] = Msg'[Count] + 1
    & Msg[ID_Part] = Self
    & ~Channel_Closed



TRANSITION Notify_Timeout  N_T_Dur
 ENTRY
      EXISTS t1: Time (Start(New_Info,t1) & Now - t1 ≥ Input_Tout)
    & ~Channel_Closed
 EXIT
      Msg[Data_Part] = Closed
    & Msg[Count] = Msg'[Count] + 1
    & Msg[ID_Part] = Self
    & Channel_Closed



END Top_Level
END Input



SPECIFICATION Packet_Maker
LEVEL Top_Level


IMPORT
  Receiver, Data, Message, Message_List, Pos_Integer,
  Receiver_ID, Set_Of_Receiver_ID, Info, Closed, N, L,
  P_M_Dur, Del_Dur, N_I_Dur, Msg


EXPORT
  Output


VARIABLE
  Packet: Message_List,
  Previous(Receiver_ID): Time,
  Output: Message_List
```

```
CONSTANT
  Maximum: Pos_Integer,
  Del_Tout, H3: Time
   /*H3 denotes an upperbound for the time to deliver
    a message after it has been processed*/

IMPORTED VARIABLE CLAUSE
      FORALL t: Time (
       (t - N_I_Dur) MOD (N/L*P_M_Dur + Del_Dur) = 0
          → EXISTS S: Set_Of_Receiver_ID (
                 | S | = N/L
              & FORALL i: Receiver_ID
                   (i ISIN S ↔ Receiver[i].End(New_Info) = t)))
  & FORALL t:Time (
      (t - N_I_Dur) MOD (N/L*P_M_Dur + Del_Dur) ≠ 0
         → FORALL i:Receiver_ID (
              ~Receiver[i].End(New_Info) = t))
  & FORALL i:Receiver_ID (
      Receiver[i].Msg[Data_Part] ≠ Closed)

INITIAL
      Packet = EMPTY
    & FORALL i:Receiver_ID (Previous(i)=0)
    & Output = EMPTY

INVARIANT
    /*Changes in Output occur at and only at
     the end of a Deliver*/
  FORALL t: Time (
   Change(Output, t) ↔ past(End(Deliver), t) = t)
&
   /* No new messages are generated by the packet
  assembler */
  FORALL k:Integer (
      k>0 & k≤LIST_LEN(Output)
   → EXISTS i:Receiver_ID, t:Time (
        t<Now & past(Receiver[i].Msg,t)=Output[k]) )
&
   /*The order that messages appear in an output packet
   is the order in which they were processed from the channels*/
   FORALL k:Integer (
       k>0 & k<LIST_LEN(Output)
     → EXISTS t1,t2:Time (
          t1 < t2 < Now
        & past(End(Process_Message), t1) = t1
        & past(End(Process_Message), t2) = t2
        & Output[k] =
             past(Packet[past(LIST_LEN(Packet),t1)], t1)
        & Output[k+1] =
             past(Packet[past(LIST_LEN(Packet),t2)], t2))
&
   /* The order is also preserved across output packets */
  EXISTS t:Time (
       Start-2(Deliver, t) & End(Deliver) > Start(Deliver))
   → EXISTS t1,t2:Time (
         t1 < t2 < Now
     & past(End(Process_Message, t1) = t1
     & past(End(Process_Message, t2) = t2
     & past(Output[past(LIST_LEN(Output), Start(Deliver)],
           Start(Deliver)) =
             past(Packet[past(LIST_LEN(Packet),t1)], t1)
     & Output[1] =
          past(Packet[past(LIST_LEN(Packet),t2)], t2))
&
   /* Every message in Output was previously in Packet and
   all of the elements of Packet have not changed from when
   they were put into the packet until the packet is output*/
  FORALL k:Integer (
       k>0 & k≤LIST_LEN(Output)
   ↔ EXISTS t:Time (
          t<End(Deliver)
       & past(End(Process_Message,t)=t
       & past(Packet[past(LIST_LEN(Packet),t)], t)
             = Output[k]
       & FORALL t1:Time (
            t1≥t & t1<End(Deliver)
          → past(Packet[past(LIST_LEN(Packet), t)], t) =
               past(Packet[past(LIST_LEN(Packet), t)], t1))))
&
  FORALL t1:Time (
       t1≤Now-H3 & past(End(Process_Msg),t1)=t1
   → EXISTS t2:Time (
         t2>t1 & t2 ≤ Now
      & past(End(Deliver),t2)=t2
      & past(Packet[past(LIST_LEN(Packet), t1)], t1) =
          past(Output[past(LIST_LEN(Packet), t1)], t2)
      & FORALL t:Time (
            t≥t1 & t<t2
          → past(Packet[past(LIST_LEN(Packet), t1)], t1) =
               past(Packet[past(LIST_LEN(Packet), t1)], t))))

SCHEDULE
  /*The transition Deliver is activated cyclically. Furthermore,
  it always delivers a packet with N/L elements*/
    EXISTS t:Time  (End-2(Deliver,t))
      → End(Deliver) - End-2(Deliver) =
                    N/L*P_M_Dur + Del_Dur)
  & FORALL t: Time (past(End(Deliver), t) = t
      → LIST_LEN(past(Output, t)) = N/L)

TRANSITION Process_Msg(R_id:Receiver_ID)   P_M_Dur
 ENTRY
  /*Packet is not full and either (a) the present message has
  been produced after the last message that has been processed
  from that channel, or (b) the value of the current message is
```

```
 Closed and the value of the previously processed message
 for that channel was not Closed*/
   LIST_LEN(Packet) < Maximum
 & (Receiver[R_id].End(New_Info) > Previous(R_id)
    | (  Receiver[R_id].Msg[Data_Part]=Closed
       & past(Receiver[R_id].Msg[Data_Part],
                         Previous(R_id)) ≠ Closed ))
EXIT
   Packet = Packet' CONCAT
                         LIST(Receiver[R_id].Msg)
 & Previous(R_id) BECOMES Now


TRANSITION Deliver  Del_Dur
 ENTRY
 /*Either Packet is full or Packet is not empty and the
 timeout elapsed from the last Deliver or from the initial
 time*/
    LIST_LEN(Packet) = Maximum
  | ( LIST_LEN(Packet) > 0
    & (  EXISTS t:Time (  Start(Deliver, t)
                       & Now - t = Del_Tout)
       | Now = Del_Tout - Del_Dur + N_I_Dur))

EXIT
     Output = Packet'
  & Packet = EMPTY


FURTHER ASSUMPTIONS #1
  CONSTANT REFINMENT
   Del_Tout = 0 & Maximum = N/L
  TRANSITION SELECTION
   {Process_Message, Deliver} TRUE {Process_Message}


FURTHER ASSUMPTIONS #2
  CONSTANT REFINMENT
      Del_Tout = N/L*P_M_Dur + Del_Dur
   & Maximum = N


END Top_Level
END Packet_Maker
```

REFERENCES

[1] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," *5th IEEE LICS 90*, 1990, pp. 414–425.

[2] C. Chang, H. Huang, and C.C. Song, "An approach to verifying concurrency behavior of real-time systems based on time Petri net and temporal logic," *InfoJapan 90*, 1990, pp. 307–314.

[3] A. Coen-Porisini, R. Kemmerer, and D. Mandrioli, "Formal verification of real-time systems in ASTRAL," Tech. Rep. TRCS 92-22, Dept. of Comput. Sci., Univ. of California, Santa Barbara, CA, USA, Sept. 1992.

[4] ——, "A formal framework for ASTRAL interlevel proof obligations," Tech. Rep. TRCS 93-04, Dept. of Comput. Sci., Univ. of California, Santa Barbara, CA, USA, Apr. 1993.

[5] M. Felder, D. Mandrioli, and A. Morzenti, "Proving properties of real-time systems through logical specifications and Petri net models," *IEEE Trans. Software Eng.*, vol. 20, pp. 127–141, Feb. 1994.

[6] A. Gabrielian and M. Franklin, "Multilevel specification of real-time systems," *CACM 34*, vol. 5, pp. 51–60, May 1991.

[7] R. Gerber and I. Lee, "A layered approach to automating the verification of real-time systems," *IEEE Trans. Software Eng.*, vol. 18, pp. 768–784, Sept. 1992.

[8] C. Ghezzi and R. Kemmerer, "ASTRAL: An assertion language for specifying real-time systems," in *Proc. 3rd Eur. Software Eng. Conf.*, 1991, pp. 122–146.

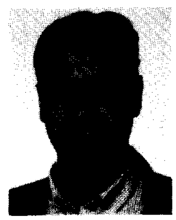
[9] ——, "Executing formal specifications: The ASTRAL to TRIO translation approach," in *Proc. TAV4: Symp. Testing, Anal., Verification*, 1991, pp. 112–119.

[10] J. Ostroff, *Temporal Logic for Real-Time Systems*, vol. 1, Advanced Software Development Series. Taunton, UK: Research Studies Press, 1989.

[11] A. Pnueli, "The temporal logic of programs," in *Proc. 18th Annu. Symp. Foundations of Comput. Sci.*, 1977, pp. 46–57.

[12] I. Suzuki, "Formal analysis of alternating bit protocol by temporal Petri nets," *IEEE Trans. Software Eng.* vol. 16, pp. 1273–1281, Nov. 1990.

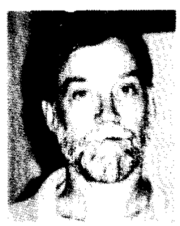
[13] P. Zave, "PAISLey user documentation volume 3: Case studies," Comput. Technol. Res. Lab. Rep., AT&T Bell Laboratories, Murray Hill, NJ, USA, 1987.



A. Coen-Porisini received the laurea degree in electrical engineering and the Ph.D. degree in computer science from the Politecnico di Milano, Milan, Italy, in 1987 and 1992, respectively.

He was a Visiting Scholar at the University of California at Santa Barbara, USA, from June 1992 to August 1993. He is currently an Assistant Professor of Computer Science at the Politecnico di Milano, Milan, Italy. His main research interests are in software engineering, with particular reference to specification languages and object-oriented systems.

Dr. Coen-Porisini is a member of the ACM and the IEEE Computer Society.



R. A. Kemmerer received the B.S. degree in mathematics from the Pennsylvania State University, State College, PA, USA, in 1966, and the M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles, CA, USA, in 1976 and 1979, respectively.

He is a Professor and Chair of the Department of Computer Science at the University of California, Santa Barbara, USA. He also leads the Reliable Software Group there. He has been a Visiting Scientist at the Massachusetts Institute of Technology, and a Visiting Professor at the Wang Institute and the Politecnico di Milano, Italy. From 1966 to 1974, he worked as a Programmer and Systems Consultant for North American Rockwell and the Institute of Transportation and Traffic Engineering at the University of California at Los Angeles, USA. His research interests include formal specification and verification of systems, computer system security and reliability, programming and specification language design, and software engineering.

Dr. Kemmerer is author of the book *Formal Specification and Verification of an Operating System Security Kernel*, and coauthor of the book *Computer's at Risk: Safe Computing in the Information Age*. He has served as a member of the National Academy of Science's Committee on Computer Security in the DOE (1987–88), the System Security Study Committee (1989–91), and the Committee for Review of the Oversight Mechanisms for Space Shuttle Flight Software Processes (1992–93). He has also served as a member of the National Computer Security Center's Formal Verification Working Group, and was a Member of the NIST's Computer and Telecommunications Security Council. He is also the past Chairman of the IEEE Technical Committee on Security and Privacy, and is a past member of the Advisory Board for the ACM's Special Interest Group on Security, Audit, and Control. He also serves on the editorial boards of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and the *ACM Computing Surveys*. He is a member of the Association for Computing Machinery, the IEEE Computer Society, and the International Association for Cryptologic Research.

D. Mandrioli received the degree in electrical engineering from the Politecnico di Milano, Milan, Italy, in 1972, and received an advanced degree in mathematics at the Universitá Statale di Milano, Italy, in 1976.

He was an Assistant Professor and an Associate Professor at the Politecnico di Milano from 1976 to 1980, and was a Professor at the Université di Udine, Italy, from 1981 to 1983. Since then, he has been a Professor of Computer Science at the Politecnico di Milano, Milan, Italy. He was also a Visiting Scholar at the University of California at Los Angeles in 1976, at the University of California at Santa Barbara in 1981 and 1992, and at Hewlett Packard Research Laboratories in Palo Alto, CA, USA, in 1989. His research interests include theoretical computer science and software engineering, with particular reference to specification languages and environments, programming languages, and real-time systems.

Dr. Mandrioli has published more than 70 scientific papers in major journals. He is also a coauthor, with C. Ghezzi, of the book *Theoretical Foundations of Computer Science*, and of the book *Fundamentals of Software Engineering*, with C. Ghezzi and M. Jazayeri. He has also written several other books in Italian. Mandrioli serves as a reviewer for many international conferences and journals and has participated on the program committees of many international conferences. He is a member of the ACM and the New York Academy of Sciences.