Specification of Realtime Systems Using ASTRAL

Alberto Coen-Porisini, *Member*, *IEEE*, Carlo Ghezzi, *Member*, *IEEE*, and Richard A. Kemmerer, *Fellow*, *IEEE*

Abstract—ASTRAL is a formal specification language for realtime systems. It is intended to support formal software development and, therefore, has been formally defined. The structuring mechanisms in ASTRAL allow one to build modularized specifications of complex systems with layering. A realtime system is modeled by a collection of state machine specifications and a single global specification. This paper discusses the rationale of ASTRAL's design. ASTRAL's specification style is illustrated by discussing a telephony example. Composability of one or more ASTRAL system specifications is also discussed by the introduction of a composition section, which provides the needed information to combine two or more ASTRAL system specifications.

Index Terms—Formal methods, formal specification and verification, assertions, temporal logic, realtime systems, timing requirements, state machines, composability, ASLAN, TRIO.

1 Introduction

A REALTIME computer system is a system that must perform its functions within specified time bounds. Realtime computer systems are increasingly being used in critical applications such as aircraft avionics, nuclear power plant control and patient monitoring. These systems are generally characterized by complex interactions with the environments in which they operate and strict time constraints whose violation may have catastrophic consequences. The need for these software systems to be highly reliable is evident.

The best way to improve software quality is to develop it formally. Existing informal software development methods and tools [56], [38], [32] are often unable to provide acceptable levels of assurance for many realtime applications, because of the combination of complexity and critical requirements.

Although research in the area of realtime systems has been quite active, and a number of experimental environments supporting formal specifications have been developed (see Section 4), the search for adequate notations and tools is still ongoing. The ASTRAL formal specification language for realtime systems and its associated support environment, which is currently being developed, are intended to provide a solution to the problem.

Because ASTRAL is intended to be used for specifying large and complex realtime systems, it was designed to support specifications that are layered and compositional. Layering and composition are two complementary approaches to hierarchical system development. A layered specification method allows one to refine the specification of a process to show more detail, without changing the interface of the specified system. This is important because it allows designers to prove, test, or otherwise examine properties of a process whose behavior is specified abstractly, and then iteratively refine the behavioral specification to be as close to an implementation as appropriate for a given assurance level.

A compositional specification method allows one to reason about the behavior of a system in terms of the specifications of its components [59]. That is, the behavior of a system comprising several component processes is completely determined by the component specifications. This is important because it modularizes a system's proof and allows for bottom-up development.

ASTRAL provides mechanisms for specifying critical system requirements, and a formal proof system for proving the satisfaction of the stated requirements has been defined. These proofs are divided into two categories: *intralevel* proofs and *interlevel* proofs. The former deal with proving that the specification of level i is consistent and satisfies the stated critical requirements, while the latter deal with proving that the specification of level $i + 1$ is consistent with the specification of level i. Details of the formal proof system can be found in [15], [16].

The ASTRAL specification language is intended to be executable. An executable specification language allows developers to treat specifications as prototypes. This is important because testing in the design stage, even before attempting proofs, can be a cost-effective means of finding design flaws [42], [20]. A translation scheme for translating ASTRAL to TRIO [31], [50] was defined for an earlier version of the ASTRAL language [28]. More recently, the semantics of ASTRAL have been formally defined using both a model theoretic approach and an axiomatic semantic approach [17]. This provides a firm theoretical basis for the development of a software development environment with

- *A. Coen-Porisini and C. Ghezzi are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano 20133, Italy. E-mail: {coen, ghezzi} @ elet.polimi.it.*
- *R.A. Kemmerer is with the Reliable Software Group, Computer* Science *Department, University of California, Santa Barbara, CA 93106. E-mail: kemm@cs.ucsb.edu.*

Manuscript received 24 Oct. 1994, revised 24 July 1996.
Recommended for acceptance by J. Gannon.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105192.

a specification execution tool as one of its components. The axiomatic semantics of the ASTRAL abstract machine can be found in [15].

The main intent of this paper is to give the reader an introduction to the language. In the next section, the specification language is presented using a telephone system example to clarify the language components and their use. In Section 3 the ASTRAL approach to composing system specifications is described and the telephony example is extended. Section 4 discusses related work. Finally, conclusions drawn from this work and further areas of research are presented.

2 An Overview of ASTRAL

In ASTRAL a realtime system is described as a collection of state machine specifications each of them representing a process type of which there may be multiple statically generated instances plus a global specification, which contains declarations for types and constants that are shared among more than one process type, as well as assumptions about the global environment and critical requirements for the whole system.

Fig. 1 presents the syntactic structure for an ASTRAL specification.

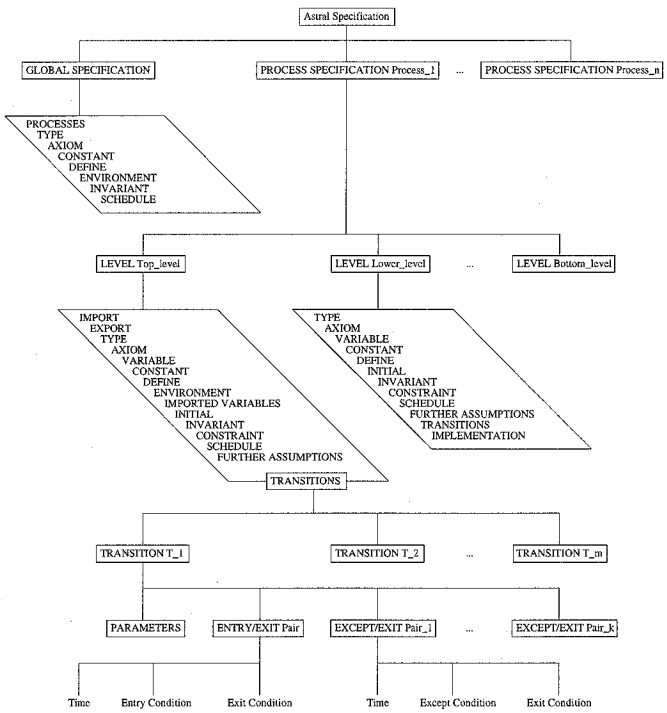


Fig. 1. ASTRAL syntactic structure.

An ASTRAL *process specification* consists of a sequence of *levels*. Each level is an abstract data type view of the system being specified. The first ("top level") view is a very abstract model of what constitutes the process (types, constants, variables), what the process does (state transitions), and the critical requirements the process must meet. Lower levels are increasingly more detailed with the lowest level corresponding closely to high level code.[1]

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* which can be changed only by means of *state transitions*. Transitions are described in term of entry and exit assertions by using an extension of first order predicate calculus. Transition *entry assertions* describe the constraints that state variables must satisfy in order for the transition to fire, while *exit assertions* describe the constraints that are fulfilled by state variables after the transition has fired. An explicit nonnull duration is associated with each transition. Transitions are executed as soon as they are enabled (i.e., when their entry assertion is satisfied).

Every process can export both state variables and transitions; as a consequence the former are readable by other processes while the latter are executable from the external environment. Interprocess communication is accomplished by broadcasting the value of exported variables, as well as the start and end times of exported transitions.

In addition to specifying system state (through process variables and constants) and system evolution (through transitions), an ASTRAL specification also defines system critical requirements and assumptions on the behavior of the environment that interacts with the system. The behavior of the environment is expressed by means of *environment clauses*, which describe assumptions about the pattern of invocation of external transitions. Critical requirements are expressed by means of invariants and schedules. *Invariants* represent requirements that must hold in every state that may be reached from the initial state, no matter what the behavior of the external environment is, while *schedules* represent additional properties that must be satisfied provided that the external environment behaves as assumed.

In what follows the computational model of ASTRAL is first presented and then the main features of the language are discussed.

2.1 The Computational Model

The computational 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.

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. The maximal parallelism approach was chosen on the basis of separating independent concerns; that is, first demonstrate that a design is satisfactory, then, and only then, consider the scheduling problem imposed by a particular implementation's limited resources. This approach, advocated in [22] for realtime systems and in [13] for parallel systems, allows for much cleaner designs than an approach that starts with scheduling concerns. A design based on the structure of the system rather than on its scheduling problems will usually be easier to maintain and/or modify. In

1. For space reasons, layering of specifications is not addressed in this paper. The reader may refer to [16].

addition, architectures that meet the maximal parallelism assumptions are becoming more prevalent.

Process cooperation, which involves both communication and synchronization, may be achieved in essentially two ways: either by data sharing or by message passing [11]. The interface specification of RT-ASLAN [6], which is an earlier realtime specification language developed by the Reliable Software Group at the University of California at Santa Barbara, is an example of modeling communication with shared data in a realtime specification language. At the implementation level, data sharing has obvious performance advantages and is, therefore, often used in current realtime systems. However, there is no apparent advantage in using data sharing at the specification level for describing process interactions at an abstract level. There are instead obvious disadvantages. For example, contention for shared data must be addressed in the specification, which implies that mutual exclusion also must be addressed. Furthermore, future realtime systems are likely to be less tightly-coupled than existing systems. For these reasons, in ASTRAL cooperation is modeled with implicit message passing rather than with data sharing. Implicit rather than explicit message passing was chosen to further simplify the design and to concentrate on the structure of the realtime system. The message passing is also assumed to be instantaneous. The specifics of the implicit multicast message communication model are presented in Section 2.2.6.

When designing a realtime specification language one strives to construct models that are both abstract enough to be understandable and yet realistic enough to be useful. The ASTRAL computation model implicitly makes the assumption that there is a global clock and that multicast communication is instantaneous. The assumption of instantaneous broadcasts of state changes is valid if the target implementation uses shared memory, for then there is truly a zero time communication delay. If, however, the implementation is distributed, then this assumption will hold only if the actual delay times are less than the granularity of the timing units. For example, if the communication delay time is in the order of microseconds and the timing requirements are measured in milliseconds, then the delay is a "virtual zero" communication delay. Obviously, as the processes being modeled are more widely distributed the granularity of their critical timing requirements will need to be greater. Thus, to be able to use the ASTRAL approach on a truly distributed system the communication medium for the system must be able to guarantee a "virtual zero" time delay relative to the critical timing requirements. The assumption about a global clock is acceptable for many systems, but may not be valid for some distributed systems. This problem is one that is currently being addressed by a number of researchers [47], [48], [21].

2.2 The ASTRAL Language

In this section the main features of ASTRAL are discussed using an example specification of a phone system composed of multiple control centers. Each control center is responsible for the phones belonging to its area, and it is provided with all the functionality needed to set up a local call. Control centers are also intended to deal with long distance calls (i.e., calls to other areas). Calls to outside areas are modeled by exported variables (i.e., the data is sent to the external environment), while calls from an outside area are modeled as exported transitions (i.e., they are the information provided in the parameters of a call to an exported transition from the external environment). The example is a simplification of a real phone system, every local phone number is seven digits long, area codes are three digits long, a customer can be connected to at most one other phone (either local or in another area), and ongoing calls cannot be interrupted. The motivation for this example came from the telephony example in a paper by Dasarathy [18]. Using Dasarathy's example as a starting point and the local phone system at the University of California, Santa Barbara for further clarification, the example specification was developed.

Most of the examples of the specification that appear in this section are from the global specification and the Central_Control specification. Complete specifications can be found in Appendix A.

2.2.1 Processes

The phone system consists of two process type specifications: Phone, which models the phone instrument found in most homes, and Central_Control.

The process declarations

```
Phones: array[1 .. Num_Phone] of Phone,
Centrals: array[1 .. Num_Area] of Central_Control,
```

which occur in the global specification, declare that there are Num_Phone static instances of the phone process type and Num_Area static instances of the central control process type. Each of these is accessed as Phones[i] (Centrals[i]), where i is in the range 1..Num_Phone (1..Num_Area).

2.2.2 Types

ASTRAL is a strongly typed language. Integer, Real, Boolean, ID, and Time are the only primitive types. All other simple and constructed types used in a process specification must be either declared in the type section of the specification or must be declared in the global specification and explicitly imported by the process type specification.

The following type declarations appear in the global phone system specification.

```
Digit IS TYPEDEF d: Integer (d ≥ 0 & d ≤ 9)
```

indicates a subtype declaration, and

```
Digit_List IS LIST OF Digit
```

declares Digit_List as a list of Digits, while

```
Connection IS STRUCTURE OF (From_Area, From_Number,
To_Area, To_Number: Digit_List)
```

declares Connection as a structure composed of four fields of type Digit_List. Connection represents the data structure used to store the information related to a long distance call.

The type ID is one of the primitive types of ASTRAL. Every instance of a process type has a unique id. An instance can refer to its own id by using "Self." There is also an ASTRAL specification function *idtype*(*i*), which returns the type of the process that is associated with the id i. Thus the global declaration

```
Phone_ID IS TYPEDEF pid: ID (IDTYPE(pid) = Phone)
```

declares Phone_IDs to be exactly those ids that are for process instances of type Phone.

The central control process specification also defines a new subtype Area_Phone representing the set of phones actually connected to a single phone process instance:

```
Area_Phone IS TYPEDEF p: Phone_ID (In_Area(p) = Self)[2]
```

Finally, the global type declaration

```
Enabled_State IS (Idle, Ready_To_Dial, Dialing, Ringing,
                  Waiting, Talk, Calling, Disconnecting, Busy,
                  Alarm)
```

is an enumerated type and indicates the various modes that a customer's phone can be in. These different modes are used to determine what transitions the phone can execute and how the central control should respond to certain actions of a customer's phone.

2.2.3 Variables, Constants, and Definitions

In ASTRAL one state is differentiated from another by the values of the state variables, and it is the state variables that are referenced and/or modified by the state transitions. All of the state variables must be declared in the variable section of the formal specification.

A special variable called Now is used to denote the current time. The value of Now is zero 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.

In the central control specification there are eight state variables all of which are parameterized by Area_Phone. The first two

```
Phone_State(Area_Phone): Enabled_State,
Long_Distance(Area_Phone): Boolean
```

indicate the central control's view of the mode of each of its customer's phones, and whether or not a phone is involved in a long distance call, respectively.

The reader should note that the central control's view of a particular phone's mode may differ from the actual mode of the phone. For instance, when a customer P first picks up his/her handset the central control may view that particular phone as being idle (i.e., Phone_State(P) = Idle), but it is actually active. The central control, however, will not treat P as active until P's offhook response (i.e., P.Offhook) is processed by the central control's Give_Dial_Tone transition for P. Obviously, this is an action that should occur in a timely fashion. In fact, the global scheduling requirement that is used for the phone system example addresses this issue.

The variables

```
Enabled_Ring_Pulse(Area_Phone): Boolean,
Enabled_Ringback_Pulse(Area_Phone): Boolean
```

are necessary because the central control actually pulses the ring of the callee's phone and the ringback tone of the caller's phone, and they are pulsed independent of each other. When Enabled_Ring_Pulse(P) is true and the mode of phone P is "Ringing" this indicates that phone P should be ringing its bell (i.e., Ring = true for phone P). Note that the central control does not ring the bell, but rather indicates by means of an exported variable (an electronic signal in the actual system) that the bell should be ringing. The transition that actually rings the bell is the Start_Ring transition of process Phone. Enabled_Ringback_Pulse is used in an analogous fashion to pulse the ringback tone in the caller's phone.

The next two variables

```
Connected_To(Area_Phone): Area_Phone,
Number(Area_Phone): Digit_List
```

indicate to what other phone each phone is connected and the number (or partial number) that is being dialed. Connected_To(P) is meaningful only in case of a local call and if P is in waiting, ringing, or talk mode. Likewise, Number(P) is only meaningful when P is in ready to dial or dialing mode.

For long distance calls, the variables

```
LDOut_Line(Area_Phone): Connection,
LDOut_Status(Area_Phone): Connection_Status
```

indicate to whom the phone is connected and the status of the connection. LDOut_Line is not meaningful if Long_Distance is false.

Constants in ASTRAL are values that cannot change over the lifetime of the system. In the global specification, the constant

```
In_Area(Phone_ID): Central_Control_ID
```

is an example of a parameterized constant. It is used to describe that each phone is associated with a unique central control, and that such a binding cannot change.

The first four constants in the central control specification

```
Uptime_Ring, Downtime_Ring, Uptime_Ringback,
Downtime_Ringback: Time
```

are of type Time and are used to indicate the pulse rate for ringing a customer's phone and for giving the ringback tone.

In ASTRAL, definitions are used to make the specification more readable. They may contain zero or more parameters.

There are three definitions used in the central control specification. The first one

```
DEFINE
Plug(L1, L2: Connection): Boolean ==
      L1[From_Area] = L2[To_Area]
   & L1[From_Number] = L2[To_Number]
   & L1[To_Area] = L2[From_Area]
   & L1[To_Number] = L2[From_Number]
```

introduces a predicate which is used in the specification of long distance calls to specify that L2 and L1 hold the same values after having switched the To_Area and To_Number fields of one with the corresponding From_Area and From_Number fields of the other. The second one

```
Count(P:Area_Phone): Integer == LIST_LEN(Number(P))
```

is used to track how many digits have been processed for each customer. LIST_LEN is an ASTRAL specification function that indicates the number of items in the list.

The third one defines the predicate CallingOut (P: Area_Phone, L: Connection) which is used to represent that phone P is making a long distance call using line L.

2. In_Area is a parameterized constant whose meaning will be discussed in Section 2.2.3.

2.2.4 Interface Section

The interface section of an ASTRAL process specification indicates what types, constants, and definitions declared in the global specification are used by the process, what variables exported by other processes are referenced, and what variables and transitions are exported by the process. These are specified by the Import and Export clauses.

State variables and transitions may be explicitly exported by a process, which makes the variable values and information about the transitions visible to other processes. Exported variables and transitions must be explicitly imported to be referenced in another process specification. Exported transitions are visible to the external environment and are executed in response to calls issued by the external environment. The export clause for the phone process

EXPORT
Offhook, Next_Digit,
Pickup, Enter_Digit, Hangup

indicates that the values of Offhook and Next_Digit can be imported by other processes and that the transitions Pickup, Enter_Digit, and Hangup are made available to the external environment.

The import clause indicates which globally declared types and constants are used by a process and which variables and transitions exported by other processes are referenced by this process. The central control specification's import clause

IMPORT
Digit, Digit_List, Connection, Phone_ID, Central_Control_ID,
Enabled_State, Connection_Status, In_Area, Max_Cust,
Phones.Offhook, Phones.Next_Digit, Phones.Pickup,
Phones.Enter_Digit

indicates the types and constants that are declared in the Phone_System global specification that are imported, as well as the exported variables Offhook and Next_Digit and the exported transitions Pickup and Enter_Digit for each declared instance of process phone. Fig. 2 shows the architecture of a local central control and its associated phones.

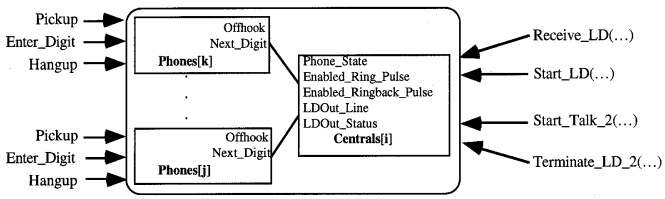


Fig. 2. The phone system.

2.2.5 Initial Clause

The initial clause of a process specification expresses the restrictions on the initial state of the process type. That is, for each state variable it is necessary to express the restrictions that are to be placed on its initial value.

For the central control the initial condition specifies that, in the view of the central control, initially all phones are idle, no digits have been processed, no phones are ringing nor are any receiving a ringback tone, and no long distance call is ongoing.

INITIAL
FORALL P: Area_Phone
(Phone_State(P) = Idle
& Number(P) = NIL
& ~Enabled_Ring_Pulse(P)
& ~Enabled_Ringback_Pulse(P)
& ~Long_Distance(P)
& LDOut_Status(P) = Available)

It is not necessary to specify an initial value for the Connected_To variable because it is only meaningful when a phone is in either waiting, ringing, or talk mode. By reviewing the transitions for the central control process one can see that before a phone can be placed in one of these modes its connected value will be updated appropriately (See the Process_Local_Call transition.). Likewise, it is not necessary to specify a value for LDOut_Line if Long_Distance is false.

2.2.6 State Transitions

ASTRAL transitions are used to specify the ways in which an instance of a process type can change from one state to another. A transition is composed of a header, an entry assertion, and an exit assertion. The header gives type information for the transition's parameters and specifies the exact amount of time required for the transition to execute. The *entry* assertion expresses the enabling conditions that must hold for the transition to occur, and the *exit* assertion specifies the resultant state after the transition occurs. That is, it specifies the values of the state variables in the new state relative to the values they had in the previous state.

In an ASTRAL specification exceptions are dealt with explicitly. That is, a transition can have except/exit pairs in addition to the standard entry/exit pair. An *except* assertion expresses an exception that may occur when a transition is invoked. The corresponding exit assertion specifies the resultant state after the transition occurs.

A transition is executed as soon as its precondition is satisfied (assuming no other transition is executing). If two or more transitions are enabled simultaneously, a nondeterministic choice[3] will occur and only one of them will be executed. Whenever a process instance starts executing an exported transition it broadcasts the start time and the values of the actual parameters to all interested processes (i.e., any process that has imported the transition). When the transition is completed the end time as well as the new value of any exported variables that were modified by the transition are broadcast. Of course, any exported variables that are modified by a nonexported transition are also broadcast by the process when the transition completes execution. Thus, if a process is referencing the value of an exported variable while a transition is being executed by the process exporting the variable, the value obtained is the value the variable had when the transition commenced. That is, 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.

3. To reduce the amount of nondeterminism it is possible to assign priority among transitions by means of the Further Assumption clause (see Section 2.2.10).

Start(T, t) is a predicate that is true if and only if transition T starts at time t and there is no other time after t and before the current time (Now) when T starts (i.e., t is the time of the last firing of T). For simplicity, the functional notation Start(T) is adopted as a shorthand for "time t such that Start(T, t)" whenever the quantification of the variable t (whether existential or universal) is clear from the context. $Start_k(T)$ is used to give the start time of the kth previous occurrence of T. References to the end time of a transition T may be specified similarly using *End(T)* and $End_k(T)$.

The Process_Local_Call transition represents the central control attempting to establish a connection for a caller who has entered seven digits (i.e., the caller is making a local phone call). It has both a normal entry/exit pair and an except/exit pair. The entry/exit pair corresponds to the case where the called party is in idle mode and the except/exit pair corresponds to the case where the callee is busy. The value Tim3 is the execution time for this transition.[4] Notice that in exit assertions, variable names followed by a prime (') indicate the value that the variable had when the transition fired.

```
TRANSITION Process_Local_Call(P:Area_Phone) Tim3
ENTRY
      P.Offhook
   &  ~Long_Distance(P)
   &  Count(P) = 7
   &  Phone_State(P) = Dialing
   &  ~Get_ID(Number(P)).Offhook
   &  Phone_State(Get_ID(Number(P))) = Idle
EXIT
      Phone_State(P) = Waiting
   &  Phone_State(Get_ID(Number'(P))) = Ringing
   &  ~Long_Distance(Get_ID(Number'(P)))
   &  Connected_To(P) = Get_ID(Number'(P))
   &  Connected_To(Get_ID(Number'(P))) = P
   &  FORALL P1:Area_Phone
      (  P1 ≠ P & P1 ≠ Get_ID(Number'(P))
      → NOCHANGE(Phone_State(P1))
      &  NOCHANGE (Connected_To(P1)))
EXCEPT
      P.Offhook
   &  ~Long_Distance(P)
   &  Count(P) = 7
   &  Phone_State(P) = Dialing
   &  (  Get_ID(Number(P)).Offhook
      |  Phone_State(Get_ID(Number(P))) ≠ Idle)
EXIT
      Phone_State(P) BECOMES Busy
```

The entry assertion for the success case specifies that the caller's phone must be offhook, the call is not long distance, a seven digit number must have been processed, the phone is in dialing mode, and the phone to be connected to must not be offhook and it must be idle. The exit assertion specifies that the caller's phone is now in waiting mode, the callee's phone is in ringing mode, and the mode for all other phones is unchanged. Furthermore, the caller is indicated as being connected to the callee and vice versa, and the value of Connected_To for all other phones is unchanged. The last conjunct is necessary, because without it the value of the parameterized variables Connected_To and Phone_State would be undefined for all phones other than the caller and the callee. It should be noted that any variable not mentioned in an unprimed form in the exit clause is assumed to not change. Thus, for the success case the variables LDOut_Line, LDOut_Status, Number, Enabled_Ring_Pulse, and Enabled_Ringback_Pulse do not change. The ASTRAL specification processor automatically generates nochange expressions for these variables when constructing the proof obligations.

The entry assertion for the busy case is identical to the success case except that the phone being called is either not in idle mode or is offhook. The exit assertion for this case indicates that the mode of the caller's phone is now busy and the mode for all other phones is unchanged. The BECOMES operator that is used in this expression is a shorthand provided by ASTRAL for asserting that the value of a parameterized variable changes for some particular argument, but remains unchanged for all of the other arguments. Thus

```
Phone_State(P) BECOMES Busy
```

is equivalent to

```
FORALL P1:Area_Phone
   (  Phone_State(P1) =
         IF P1 = P
         THEN Busy
         ELSE Phone_State'(P1)
         FI )
```

The Enable_Ring and Disable_Ring transitions are used for modulating the Enabled_Ring_Pulse exported state variable to control the ringing of a customer's phone. The Enable_Ring transition sets this variable to true and the Disable_Ring sets it to false. The Phone process type uses the value of the Enabled_Ring_Pulse variable to determine when to ring its bell. The length of time for the ring pulse to be enabled is determined by the constant Uptime_Ring and the time for it to be disabled is determined by the constant Downtime_Ring.

```
TRANSITION Enable_Ring(P: Area_Phone) Tim4
ENTRY
      ~P.Offhook
   &  Phone_State(P) = Ringing
   &  ~Enabled_Ring_Pulse(P)
   &  FORALL t: Time
      (  End(Disable_Ring(P), t)
      &  FORALL t1 (t ≤ t1 ≤ Now
                  → past(Phone_State(P), t1) = Ringing)
      →  Now - t ≥ Downtime_Ring)
EXIT
      Enabled_Ring_Pulse(P) BECOMES True
```

The entry assertion for the Enable_Ring transition requires that the phone not be offhook, that its mode be ringing, and that the ring pulse is currently disabled. The last conjunct of the entry assertion controls the time between ring pulses. This expression provides an example of the use of the ASTRAL variable Now, which represents the current value of time, and of the last end time of the Dis-

4. For simplicity, in this paper it is assumed that the same amount of time is required for executing both the success case of a transition and any exception case. In the actual language design different times are provided for each entry/exit or except/exit pair.

able_Ring transition with parameter value P. It also demonstrates the use of the ASTRAL specification function past, which is used to specify the value that a variable had at some time in the past, that is, past(Phone_State(P), t1) represents the value that variable Phone_State(P) had at time t1. The entry assertion specifies that if there has been a Disable_Ring transition since the phone state became Ringing then it has been at least Downtime_Ring seconds since the last occurrence of the Disable_Ring transition completed execution with parameter P. The reader should note that because Disable_Ring is a transition of the process that is performing the End inquiry, it is not necessary to precede the inquiry with a unique id. The exit assertion for the Enable_Ring transition specifies that the ring pulse for phone P is now enabled.

The Disable_Ring transition works in an analogous manner except that its entry assertion allows for the ring pulse for phone P to be disabled early if phone P is taken offhook.

The invocation of the exported transition Receive_LD by the external environment models the occurrence of an incoming long distance call to a phone connected to the central control. Notice that the entry assertion of the transition requires that the phone not be offhook and that it is idle.

```
TRANSITION Receive_LD(LDIn_Line:Connection) Tim9
ENTRY
      LDIn_Line[To_Area] = Get_Area(Self)
   &  Phone_State(Get_ID(LDIn_Line[To_Number])) = Idle
   &  ~Get_ID(LDIn_Line[To_Number]).Offhook
EXIT
      Phone_State(Get_ID(LDIn_Line[To_Number]))
                  BECOMES Ringing
   &  LDOut_Status(Get_ID(LDIn_Line[To_Number]))
                  BECOMES Connect
   &  Long_Distance(Get_ID(LDIn_Line[To_Number]))
   &  Plug(LDOut_Line(Get_ID(LDIn_Line[To_Number])),
           LDIn_Line)
   &  FORALL P: Area_Phone
      (    P ≠ Get_ID(LDIn_Line[To_Number])
      → NOCHANGE (LDOut_Line(P)))
```

LDIn_Line is information provided by the external environment that identifies the local phone process that is about to receive a long distance call. Thus, the conjunct:

```
Plug(LDOut_Line(Get_ID(LDIn_Line[To_Number])),
LDIn_Line),
```

which uses the Plug definition (presented in Section 2.2.3), states that the local central control unit uses the variable LDOut_Line to maintain the data provided by the parameter LDIn_Line.

The last transition to be discussed is the Generate_Alarm transition.

```
TRANSITION Generate_Alarm(P:Area_Phone) Tim14
ENTRY
      P.Offhook
   &  (  Phone_State(P) = Ready_To_Dial
      |  (Phone_State(P) = Dialing
         &   P.Call(Enter_Digit) < Start(Process_Digit(P))))
   &  (  Count(P) = 0 & (Now-End(Give_Dial_Tone(P))) > 30
      |  (  Count(P) > 0
            & ( ~Long_Distance(P) & Count(P) < 7
               | Long_Distance(P) & Count(P) < 11)
            & Now - End(Process_Digit(P)) > 20)
      |  Now - End(Give_Dial_Tone(P))) > 100 )
EXIT
      Phone_State(P) BECOMES Alarm
```

This transition represents three of the dialing timing requirements that Dasarathy presented in his paper [18]. The restrictions are that

- "After receiving a dial tone, the caller shall dial the first digit within 30 sec."
- "After the first digit has been dialed, the second digit shall be dialed no more than 20 sec later." and
- "A dialer should dial seven digits in 30 sec or less ..."[5]

The first conjunct of the entry assertion for the Generate_Alarm transition requires the phone to be offhook while the second one specifies that the caller is about to begin dialing or in the process of dialing and has not entered a digit since the last digit was processed. The last conjunct captures the three requirements from Dasarathy's paper. The first disjunct specifies that no digits have been processed since the dial tone was enabled (Count(P) = 0) and that more than 30 sec have elapsed since phone P was put in ready to dial mode. The second disjunct specifies that one or more, but less than seven (11 for long distance), digits have been processed for phone P, and that either it has been more than 20 sec since the last digit was processed or more than 100 sec have elapsed since phone P was put in Ready_To_Dial mode. The exit assertion specifies that phone P is put in the alarm mode.

Fig. 3 gives a partial view of how variables Phone_State (denoted as P) and LDOut_Status (denoted as L) are affected by transitions of Central_Control when processing incoming or outgoing long distance calls.

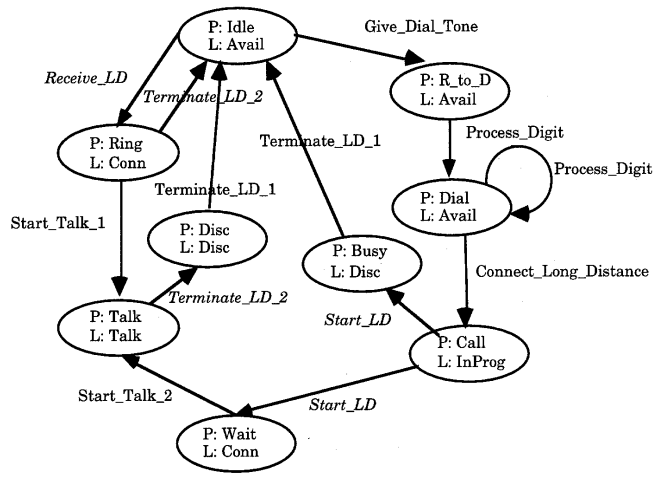


Fig. 3. The Central_Control.

2.2.7 Environment Clause

Because ASTRAL is intended to be used for designing reactive systems, it is necessary to be able to express assump-

5. For the example specification, the second requirement was extended to cover the time between *any* two digits, and the total dial time was changed to 100 sec, which seemed more reasonable.

tions about the external environment in which the system works. This is accomplished by using *environment clauses,* which formalize the assumptions that must hold on the behavior of the environment to guarantee some desired system properties. These assumptions are expressed as first-order formulas involving calls to the exported transitions. If T is an exported transition, *Call(T)* may be used in the environment clause to denote the time of the last occurrence of the call to T (with the same syntactic conventions as Start(T) and End(T)), and $Call_k(T)$ denotes the time of the kth previous occurrence of the call.[6]

There are both local and global environment clauses. Local environment clauses refer to a single instance of the process type they are associated with, while the global one refers to the system as a whole. The local environment clause for the process type Phone is:

```
   FORALL t: Time (Call(Pickup, t) → ~past(Offhook, t))
&  FORALL t: Time (Call(Hangup, t) → past(Offhook, t))
&  FORALL t: Time
   (  Call(Enter_Digit, t)
   →  (  past(Dialtone, t)
      |  EXISTS t1: Time, n: Integer, D: Digit
         (  2 ≤ n & Call_n(Enter_Digit(D), t1)
         & past(Dialtone, t1)
         & (n ≤ 7 & D ≠ 1 | n ≤ 11 & D = 1)
         &  FORALL t2: Time (t1 ≤ t2 ≤ t
                             → past(Offhook, t2))))
&  FORALL t: Time (    Call_2(Pickup, t)
                  → Call(Pickup) → Call_2(Pickup) ≥ 1)
```

This states that the phone must not be offhook in order to pick it up, it must be offhook in order to hang it up, there must be a dialtone in order to dial the first digit or the phone has remained offhook ever since the first digit was dialed, and a customer cannot pick up his/her phone more than twice in a second.

The global environment clause for the phone system specifies that no more than Max_Cust customers connected to the same central control will pick up their phone to initiate a call in any 2 sec time period. It is:

```
FORALL C: Central_Control_ID
(  SET_SIZE({ SETDEF P: Phone_ID(In_Area(P) = C
&  Now-2 ≤ P.Call(Pickup) ≤ Now)}) ≤ Max_Cust)
```

where SET_SIZE and SETDEF are two ASTRAL operators. SET_SIZE(S) returns the cardinality of the set S, while SETDEF is used to define a set of elements that satisfy a given predicate.

2.2.8 Imported Variable Clause

ASTRAL also allows assumptions about the system context provided by other processes in the system to be expressed in the *imported variable clause,* which describes patterns of changes to the values of imported variables, including timing information about any transitions exported by other processes that may be used by the process being specified (e.g., Start(T) and End(T)). The imported variable clause is optional and is not an essential part of an ASTRAL system specification. It is used to aid in proving the correctness of a system in a modular fashion. [15] provides the details of how this clause is used to partition the correctness proof.

The imported variable clause for the central control process is:

```
SET_SIZE(
{ SETDEF P: Area_Phone
      (Now – 2 ≤ P.Start(Pickup) ≤ Now)
}) ≤ Max_Cust
```

It states that there are no more than Max_Cust firings of the imported Pickup transitions in any 2 sec time period.

2.2.9 Critical Requirements

For a realtime system there are two types of critical requirements: behavioral and temporal. In ASTRAL both types are expressed in the invariants, constraints, and schedules.

The *invariants* express the critical requirements that are to hold in every reachable state. That is, they state properties that must initially be true and must be guaranteed to hold during system evolution. The *constraints* express the critical requirements that must hold between any two states that correspond to the start and end of a transition.[7] Invariants can be global or local; the global invariants represent properties that need to hold for the realtime system as a whole, while local invariants and constraints defined at the process type level represent properties that must hold for each process instance. Invariant and constraint properties must be true regardless of the environment or the context in which the process or system is running.

For the central control process type the invariant clause specifies two restrictions on Count and several on phone modes. The invariant is expressed as follows:

```
FORALL P: Area_Phone
(  (Long_distance(P) → Count(P) ≥  0 & Count(P) ≤ 11)
&  (~Long_distance(P)
    → (Count(P) ≥ 0 & Count(P) ≤ 7)
    & (    Phone_State(P) = Waiting
       →  Phone_State(Connected_To(P)) = Ringing)
    & (    Phone_State(P) = Ringing
       → Phone_State(Connected_To(P)) = Waiting)
    & (    Phone_State(P) = Talk
       → Phone_State(Connected_To(P)) = Talk))))
```

The first conjunct of the invariant indicates that for long distance calls the number being dialed by a user can never be more than 11 digits long and that it is always greater than or equal to zero. This invariant demonstrates the use of the definition Count. The second part of the invariant expresses a similar requirement on local calls (the number of digits has to be less than or equal to seven). It also expresses the requirement that when one customer is waiting for another to answer, the other customer's phone is in a ringing state. Similarly, if one's phone is ringing then the phone of the caller should be waiting for an answer. Finally, if a phone is in talk mode the phone it is connected to should also be in talk mode.

The only constraint in the central control specification is

```
FORALL P: Area_Phone
(  (    Phone_State'(P) = Busy
   |    Phone_State'(P) = Alarm
```

6. Note that there may be a delay from the time a transition T is called until it is actually started.

7. The requirements contained in a constraint could be expressed in an invariant. Thus, the constraint is just a notational convenience.

```
    |  Phone_State'(P) = Disconnect)
  & Phone_State(P) ≠ Phone_State'(P)
  → Phone_State(P) = Idle )
```

This constraint specifies that if a user reaches a busy number, doesn't dial quickly enough and gets into alarm mode, or is disconnected because the other party hangs up the phone, then the only choice for the customer is to hangup (i.e., enter idle mode). Recall that in an ASTRAL expression a primed variable (e.g., x′) indicates the value that variable had in the previous state; i.e., when the transition was invoked.

Schedules are additional system properties that are required to hold under more restrictive hypotheses than invariants. Like invariants, schedules may be either local or global and obey suitable scope rules in the same style as invariants. Unlike invariants, however, they express requirements that are to hold provided the environment and system context produce stimuli as prescribed in the environment and imported variables clauses.

Process scheduling clauses deal with timing requirements for that process only. A process schedule cannot prescribe the value of variables for another process. It may refer only to calls to its own exported transitions and references to the values of imported variables from another process.

The schedule clause for the phone process has three conjuncts that specify the relationships between the tones and rings of the phone.

The first conjunct

```
Dialtone → ~Ring & ~Ringback & ~Busytone
```

states that if a dialtone is present, then the ring, ringback, and busy tones are not. This is a schedule rather than an invariant because to prove that it holds it is necessary to have information about other processes in the system. In particular, the central control process enables these signals. The other two conjuncts of the phone system are similar.

The schedule clause for the central control process consists of seven conjuncts, each of them representing a scheduling requirement. The first one is very similar to the global schedule, which is presented below. The difference between the two is that the central control schedule is dealing only with phones in its area while the global schedule is dealing with all of the phones in the phone system.

```
FORALL P: Area_Phone, t, t1, t2: Time
   (  t ≤ t1 ≤ t2
   & Change2(Phone_State(P), t) & past(Phone_State(P), t) = Idle
   & P.End(Pickup, t1) & P.Offhook
   & Change(Phone_State(P), t2)
   → t2 ≤ t1 + 2
   & (  past(Phone_State(P), t2) = Ringing
      |  past(Phone_State(P), t2) = Ready_To_Dial))
```

It also shows the use of the ASTRAL predicate Change which is used to denote the last time that a variable has changed its value, that is Change(var, t) is true iff at time t variable var has changed value and there is no other time between t and the current time (Now) in which the variable value has changed.

The six remaining scheduling requirements that compose the schedule clause of the central control specification deal with the pulsing of the Enabled_Ring_Pulse and the Enabled_Ringback_Pulse.

The global schedule clause specifies timing requirements which involve multiple processes. It describes additional properties that the system must satisfy, when all its processes satisfy their own schedules and the assumptions on the behavior of the global environment hold. Only exported variables and the call, start, and end times of exported transitions can be used in the global schedule clause.

The global schedule clause for the example system specifies another of Dasarathy's timing requirements: "The caller shall receive a dialtone no later than 2 sec after lifting the phone receiver." The corresponding specification clause is

```
FORALL P: Phone_ID, t, t1, t2: Time
(  t ≤ t1 ≤ t2
& past(In_Area(P).Phone_State(P), t) = Idle
& Change2(In_Area(P).Phone_State(P), t)
& P.Offhook & P.End(Pickup, (t1))
& Change(In_Area(P).Phone_State(P), t2)
→ (  past(In_Area(P).Phone_State,t2) = Ringing
   |  past(In_Area(P).Phone_State, t2) = Ready_To_Dial)
& t2 ≤ t1 + 2)
```

The first conjunct of the antecedent is used to set up the time line for the schedule; that is t ≤ t1 ≤ t2. The next two conjuncts of the antecedent specify that the central control's view of phone P's previous mode was idle, and it changed to that mode at time t. The fourth conjunct specifies that phone P went offhook at time t1 and it is still offhook. The final conjunct of the antecedent specifies that the central control's view of phone P's mode last changed at time t2. The first conjunct of the consequent states that the mode of phone P (as viewed by the central control) at time t2 is either ready to dial or ringing. Furthermore, if it is ready to dial, then the mode change occurred within two secs of P going offhook. The possibility of P's mode going to ringing is necessary because of the race condition caused by another party being in the process of completing a call to P when P's phone was taken offhook.

Note that this expression does not specify that phone P actually gets a dialtone, but that it *can* get a dialtone. In order for the global schedule clause to express that phone P gets the dialtone the variable Dialtone would have to be exported by the phone process.

2.2.10 Further Assumptions and Restrictions

Schedules are not required to be proved using the basic elements of the ASTRAL specification. It is important, however, to know that the schedule is feasible. 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. Even though this kind of decision should often be postponed until a more detailed design phase, 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 could be satisfied. For this reason a further assumptions and restrictions clause can be included as part of a process specification. Unlike other components of an ASTRAL specification this clause is only used as guidance to the implementer; it is not a hard requirement.

The *further assumptions and restrictions clause* consists of two parts: 1) a further environment assumptions section and 2) a further process assumptions section. The *further environment assumptions* section obeys the same syntactic rules as the local environment. It states further hypotheses on the admissible behaviors of the environment interacting with the system. A *further process assumptions* section restricts the possible system implementations by specifying suitable selection policies in the case of nondeterministic choice between several enabled transitions, *transition selection*, or by further restricting constants, *constant refinement*. In general, the further process assumptions reduce the level of nondeterminism of the system specification. For more information on the further assumptions and restrictions clause and its role in the proof methodology see [15].

The further assumptions and restrictions clause for the central control process is:

```
FURTHER ASSUMPTIONS #1
   FURTHER PROCESS ASSUMPTIONS
      TRANSITION SELECTION
         enabled_transitions CONTAINS {Give_Dial_Tone }
      & TRUE
      → elgible_transitions = {Give_Dial_Tone}
   CONSTANT REFINEMENT
      2 > MAX(Tim1, …, Tim16) + (Max_Cust + 1)* Tim1
```

The above clause gives transition Give_Dial_Tone priority over the other transitions belonging to the central control process and states further constraints on the relationship of some constant values. These assumptions are needed to prove that the process schedule holds, that is, they describe a possible implementation choice that designers can follow to satisfy the critical requirements expressed by the schedule.

3 Composing ASTRAL Specifications

As seen in the previous section, the ASTRAL language contains structuring mechanisms that allow one to build modularized specifications of complex systems. It is often desirable to combine these system specifications into an even more complex system. Consider two ASTRAL specifications: each of these specifications contains one or more process type specifications and a global specification. Composing these specifications means to build a new specification, that is the specification of a system obtained by making the processes of the original specifications interact. The behavior, the environment, and the properties of the new system are obtained from those of its components, once their interaction is formally described. In order to compose ASTRAL specifications one needs to define how to formally describe the interconnection between two or more specifications, how the resulting new specification can be built starting from the specification of its components and the description of their interaction, and under what conditions the properties verified about the components will still be valid in the composed system.

In the next subsection a compose clause, which describes the interaction between two ASTRAL system specifications is introduced. Then, the specification of a long distance network system is introduced, and it is shown how this specification can be composed with the phone system specification to produce the specification of a long distance phone system.

3.1 The Compose Clause

A *compose clause* provides the information needed to compose two or more specifications into a single specification of the combined system. Let S_1 and S_2 denote two ASTRAL top level specifications. The interaction between the processes of S_1 and S_2 is described by specifying which exported transitions of the processes of S_1 (S_2) are no longer exported to the external environment, i.e., the stimuli needed to fire such transitions are produced by a process of the sibling system S_2 (S_1) rather than by the external environment.

Fig. 4a shows two systems, S_1 and S_2. S_1 exports transitions T1 and T2 and state variables x1, x2, and x3, while S_2 exports transition T3 and state variables y1 and y2.

When S_1 and S_2 are composed some transitions of S_1 (S_2) will not need an external call, since S_2 (S_1) is now providing part of the environment in which S_1 (S_2) works. For instance, in Fig. 4b transitions T1 and T3 are no longer exported since the events that trigger them are now represented by particular values of y2, x1, and x3, respectively. Thus, the composed system, C, will export only transition T2. That is, the external environment of C can call only transition T2 (See Fig. 4c).

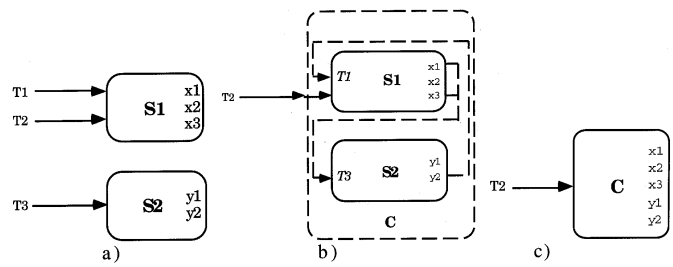


Fig. 4. The composition of S_1 and S_2 into C.

In general a compose clause contains the following parts:

- A process clause defining how many instances of each process type belonging to S_1 or S_2 are present in the resulting system.
- A set of clauses defining types, constants, and definitions that are used in the compose clause.
- A name clash resolution clause, which is used to solve any possible name clashes that can arise because of overloaded identifiers (i.e., the same identifier is used in both S_1 and S_2 with different meanings).
- A call generation clause which describes how exported transitions of S_1 (S_2) processes are triggered by events occurring in S_2 (S_1) processes. These events are described by formulas of the following form:

FORALL t: Time , … ($P(S_1) \leftrightarrow$ Call($T(a_1, \ldots a_n)$, t)),

where $P(S_1)$ is an ASTRAL well formed formula describing the occurrence of the events in S_1 that are equivalent to calling the exported transition T of S_2, and a_1, …, a_n are the actual parameters of T.

3.2 Long Distance Network

The long distance network specification is composed of a global specification and a single process type specification

(Long_Distance_Unit). Each area code is controlled by one process instance of type Long_Distance_Unit. For simplicity, it is assumed that each long distance unit instance is connected with all other instances, so that a direct communication between two long distance unit instances is always possible. The complete long distance network specification can be found in Appendix B.

The global type Line is used to represent the physical lines used for connecting one long distance unit to another and to the external environment.

Long_Distance_Unit has four variables all of which are parameterized by Line:

NetOut(Line), LocOut(Line): Connection,
NetStatus(Line), LocStatus(Line): Connection_Status

NetOut and NetStatus are used to communicate with another long distance unit, while LocOut and LocStatus are exported to send the information about the connection and the status occurring on a given line to the external environment.

The long distance unit process also exports four transitions:

Receive_Local_Req	called whenever a long distance call has been requested from the local area.
Local_Established	called to notify the unit that an incoming call to the local area has been received.
Started_Local_Talk	called to notify the unit that an incoming call, previously received has been answered.
End_Local	called to notify the unit that an incoming call has ended.

Fig. 5 shows the interface of the long distance process with the external environment, while Fig. 6 shows how the variables of Long_Distance_Unit are affected by its transitions when processing incoming or outgoing long distance calls.

The specification of transition Receive_Local_Req is as follows:

TRANSITION Receive_Local_Req(In_Line:Connection) Ti1
ENTRY
  In_Line[From_Area] = Get_Area(Self)
EXIT
  EXISTS L: Line
  (NetStatus'(L) = Available
  & Connect(NetOut(L), In_Line)
  & NetStatus(L) BECOMES In_Progress
  & FORALL L1:Line (L1 ≠ L → NOCHANGE(NetOut(L1))))

A complete discussion of the specification can be found in [14].

3.3 Composition Example

In this section the phone system specification is composed with the long distance network to obtain the specification of a long distance phone network. This requires the introduction of a compose section to define how the composition is carried out:

COMPOSITION OF Phone_System, Long_Distance_Network
AS Phone_Network

The compose section contains the declaration of the process type instances composing the new system:

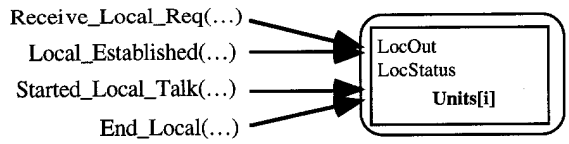


Fig. 5. Long_Distance_Unit Interface.

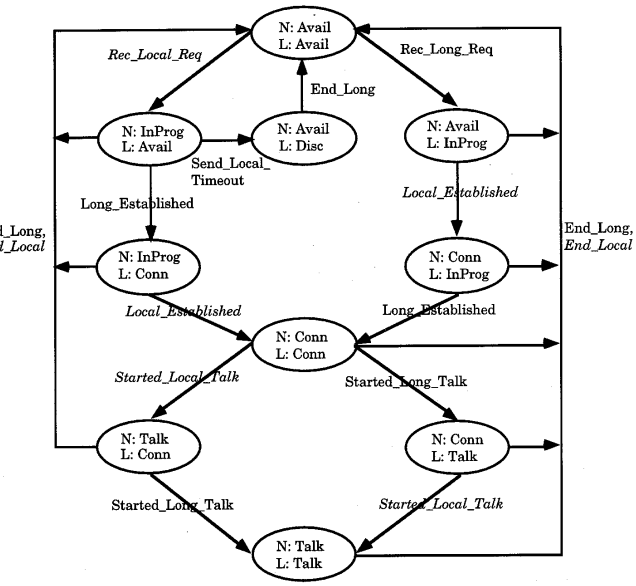


Fig. 6. The Long_Distance_Unit.

PROCESSES
Phones: array[1 .. Num_Phone] of Phone,
Centrals: array[1 .. Num_Area] of Central_Control
Units: array[1 .. Num_Area] of Long_Distance_Unit

It also contains the declaration of constant LD_Unit which defines the association of every Central_Control process instance with its corresponding Long_Distance_Unit process instance.

CONSTANTS
  LD_Unit(Central_Control_ID): LDU_ID

In order to specify that each long distance unit is associated with a single central control, the compose section contains the following axiom:

AXIOM
  FORALL C1, C2: Central_Control_ID
  (LD_Unit(C1) = LD_Unit(C2) → C1 = C2)

The above axiom is necessary to formally describe the topology of the composed system, and it is used to discharge the proof obligations of the composed system (See [14] for further information about proof obligations for the composed system).

Finally, the compose section describes how events occurring in the central control process instances are equivalent to calling the exported transitions of the long distance unit process instances and vice versa. As an example, long distance unit transition Receive_Local_Req for process instance LD_Unit(In_Area(P)) is triggered every time a customer P dials a long distance number; that is, whenever the status of the incoming line from a central control process instance (In_Area(P).LD_Out_Status) becomes "In_Progress":

FORALL t: Time, P: Phone_ID
(Change(In_Area(P).LDOut_Status(P), t)
& In_Area(P).LDOut_Status(P) = In_Progress
↔ LD_Unit(In_Area(P)).Call(Receive_Local_Req
(In_Area(P).LDOut_Line(P)), t))

In a similar manner, the central control transition Receive_LD is triggered every time an incoming long distance call is detected by a long distance unit. That is, the status of the outgoing line from a long distance unit (LD_Unit(C). Local_Status) becomes "In_Progress":

FORALL t: Time, C: Central_Control_ID, L: Line
(Change(LD_Unit(C).LocStatus(L), t)
& LD_Unit(C).LocStatus(L) = In_Progress
↔ C.Call(Receive_LD(LD_Unit(C).LocalOut(L)), t))

Fig. 7 shows how the variables and transitions from the phone system and the long distance network system interact in the composed system (For simplicity, the exported items of the phone process are not depicted). The complete composition section is presented in Appendix C.

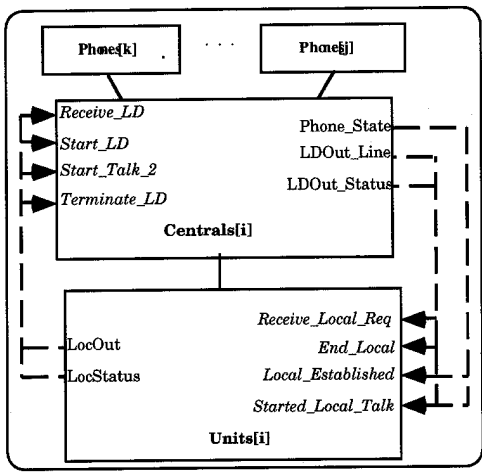


Fig. 7. The composed system.

As a result of the composition, the above transitions are no longer exported and their ENTRY / EXIT pair is extended to include the triggering mechanism whose aim is to enable a transition T only if the events described in the call generation clause occurred after the last firing of T. This will result in adding to the entry clause of such transitions a formula such as:

EXISTS t: Time …
(P(S_1) & FORALL t1: Time(t ≤ t1 < Now → ~Start(T, t1)))

where P(S_1) is the predicate used in the related call generation clause.
Note that a similar clause may also be needed in the exit assertion.

Moreover, if transition T had any formal parameter which is bound in the call generation clause to some exported variables of the triggering process, any occurrence thereof in the ENTRY / EXIT pair is substituted with the corresponding actual parameter, and the formal parameter is dropped from the transition header.

For example, transition Receive_Local_Req becomes (the italic part is what has been added):

TRANSITION Receive_Local_Req Ti1
ENTRY
EXISTS t: Time, P: Phone_ID
(*Change*(*In_Area*(*P*).*LDOut_Status*(*P*), *t*)
& *In_Area(P).LDOut_Status*(*P*) = *In_Progress*
& *FORALL t1:Time*
(*t* ≤ *t*1 < *Now*
→ ~*Start*(*Receive_Local_Req*, *t*1))
& *In_Area*(*P*).*LDOut_Line(P)*[*From_Area*] = *Get_Area*(*Self*))
EXIT
EXISTS t: Time, P: Phone_ID
(*past(Change*(*In_Area*(*P*).*LDOut_Status*(*P*), *t*)
& *In_Area*(*P*).*LDOut_Status*(*P*) = *In_Progress*
& *FORALL t1:Time*
(*t* ≤ t1 < *Now*
→ ~*Start(Receive_Local_Req*, *t*1))
& *In_Area*(*P*).*LDOut_Line*(*P*) [*From_Area*] =
Get_Area(*Self*), *Now - Ti*1)
& EXISTS L: Line
(NetStatus'(L) = Available
& Connect(NetOut(L), In_Area(P).LDOut_Line'(P))
& NetStatus(L) BECOMES In_Progress
& FORALL L1:Line
(L1 ≠ L → NOCHANGE (NetOut(L1)))))

3.4 Automatic Generation of Composed Specification

When composing two or more system specifications using the compose clause it is desirable to produce the specification of the composed system. Since an ASTRAL specification is composed of a global specification and a set of process type specification, this means that it is necessary to build the new global specification, and the new process specifications using the compose clause.

The new global specification contains a set of clauses defining types, constants and definitions built by taking the corresponding clauses declared in the compose clause and those belonging to both of the global specifications and using the name clash clause to resolve any name clashes.

Every process type specification in both of the systems is included in the composed system; however, the following transformations are needed:

- The import / export clause of each process type is modified since some transitions are no longer exported and some state variables are now imported instead. As a consequence both the local environment clause and the imported variables clause are modified in order to account for the no longer exported transition and the newly imported variables, respectively;
- Each transition belonging to a process type referred to in the call generation clause of the compose section is modified using the related call generation clause.
- The local schedule is modified using the related call generation clause if it refers to the call of a no longer exported transition.

Notice that the local invariant and the further process assumptions are not modified.

This composition can be constructed automatically. The details of how to construct the composite specifications can be found in [14].

4 Related Work

ASTRAL is an outgrowth of the authors' previous work on realtime formal specification languages. The Reliable Software Group at the Univeristy of California at Santa Barbara had already developed a layered formal specification language for sequential systems. This formal specification language, as well as the specification language processor, is called ASLAN. ASLAN's strengths include layered refinement, a state machine process model, and the ability to specify non-timing requirements and behavior. An overview of the ASLAN language can be found in [5], [6], and further information and use of the language processor are detailed in the ASLAN User's Manual [4]. ASTRAL builds on ASLAN by introducing time to the nondeterministic state machine modeling style of ASLAN, where states are specified via predicates, and state changes occur as a consequence of the application of atomic operations. Modularity and hierarchy of specifications are also inherited from ASLAN, and the syntactic flavor of ASLAN is retained.

RT-ASLAN is an earlier extension of ASLAN for specifying realtime systems [6]. In RT-ASLAN a system is modeled by a collection of state machine specifications. Processes are represented by communicating subspecifications, and shared data interfaces are represented by interface subspecifications. Each communicating subspecification defines the local state variables, state transitions and critical requirements of a single process. Each interface subspecification is essentially a data object protected by a monitor. The experience gained from specifying timing properties using RT-ASLAN were useful in designing the ASTRAL approach. The RT-ASLAN data sharing approach for modeling process interaction, however, was determined to be unsatisfactory for ASTRAL.

TRIO is a logic language designed at the Politecnico di Milano as a formal notation for specifying and verifying timing requirements [31]. The research and experimentation on TRIO initially addressed the issue of executing TRIO specifications [50]. Thus, TRIO can be considered as a realtime machine level formal language and this is why it was decided to build a high level language that could be translated to TRIO. The TRIO language was later extended with suitable object oriented mechanisms for modularizing a complex specification [53]. However, it still lacks many useful concepts which are specific to ASTRAL, such as assumptions about the environment, critical requirements, and a modular proof system.

Although the ASTRAL language was based on the authors' experience with ASLAN and TRIO, it was developed as a new language. The ASLAN state machine approach with layering is retained, but ASTRAL uses a novel approach to modeling interprocess communication, and many new specification ideas are introduced for expressing interaction with the environment and timing relationships. The experience of basing the ASTRAL language on the ASLAN and TRIO approaches prompted the selection of the name for the language: an **AS**lan based **TR**io **A**ssertion **L**anguage.

Several different approaches to formal specification of realtime systems can be found in the literature. Among them:

- Operational approaches based on different kinds of automata, including extensions to finite state machines, such as Statecharts [33], [36], Modecharts [41], different kinds of timed automata [2], [3] and various kinds of high level Petri nets, such as [51] and [30].
- Formalism based on temporal logics such as [40], [46], [44], [45], [39] and previous work on temporal logic [55], [54].
- Extensions to algebraic based languages like CCS [52] such as [58] and Constrained Expressions [19].

In [2] timed automata are used both for specifying the system and describing its properties. Thus, the problem of verifying whether a system satisfies a given property is reduced to the problem of checking the emptiness of the intersection between the languages accepted by the automata representing the system and the one describing the property respectively.[8] Alur, Feder, and Henzinger [3] show how to translate formulas written in the MITL temporal logic into timed automata. This approach allows one to check the correctness of a system modeled by timed automata against critical requirements expressed as MITL formulas. Instead, ASTRAL allows one to prove that a system meets its critical requirements by means of a proof system. That is, by providing an axiomatization of the behavior of the ASTRAL abstract machine. Moreover, timed automaton is a low level formalism when compared to ASTRAL.

An approach similar to ASTRAL's is presented in [7] where Abadi and Lamport show how TLA can be used to specify realtime systems. A TLA specification is a formula that describes the set of allowed behaviors of the system. The formula refers to a set of variables that can be viewed as the system state variables. Properties of the system are also expressed by TLA formulas, and to show that a given specification satisfies a given property one must prove that the former formula implies the latter. More recently, in [8] Abadi and Lamport show how to specify a system as a conjunction of the specifications of its components. As is the case with ASTRAL, properties of the system as a whole are proved by reasoning about the components. Furthermore, their presentation of conditional implementations, where certain assumptions are made, is similar to ASTRAL's use of environment clauses and imported variable clauses. The ASTRAL approach differs, however, in that invariants that depend on outside assumptions to be maintained are separated from those that don't (i.e., the distinction between invariants and schedules in ASTRAL).

ASTRAL also inherits some features from synchronous languages, such as ESTEREL [12], LUSTRE [34], [37], and SIGNAL [26]. These languages describe programs that react instantaneously to external events. The former is an imperative language, providing assignments, conditionals, loops, etc. The latter two are dataflow languages. For example, ESTEREL's intermodule communication via signals transferred in an instantaneous broadcast network is mirrored by ASTRAL's interprocess communication. ASTRAL differs from these languages, however, in that the actions

8. Actually the interesection is between the language accepted by the automata representing the system and the one accepted by the complement of the automata representing the property.

performed by the system in response to external stimuli take a nonnull time (i.e., there is a delay between external events and responses). Thus, ASTRAL's model is asynchronous, whereas ESTEREL, LUSTRE, and SIGNAL are synchronous. In the case of LUSTRE, validation of specifications is supported by model checking.

Like ASTRAL, the work of Gerber and Lee [29] provides a layered approach to the verification of realtime systems. With their approach the CSR application language is used to specify processes, and these processes are mapped to system resources using a configuration schema. A CSSR specification is then automatically generated. This approach is similar to the ASTRAL to TRIO translation; although, their approach is much more operational than the ASTRAL/TRIO approach.

Another language supporting layered specifications is HMS [25]. However the HMS notion of multilevel specifications is quite different from ASTRAL's. In HMS higher level specifications impose constraints on lower levels; the final specification is therefore composed of all levels. In ASTRAL each level is shown to be a correct refinement of the corresponding upper level; as a consequence the lowest level constitutes the final specification.

Recently, a number of approaches have been proposed to build formal proofs for realtime systems [1], [24], [25], [35], [41], [49], [54], [57]. In most cases, 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. In contrast, ASTRAL is provided with structuring mechanisms that allow one to build modularized specifications of complex systems with layering. As a result, one of the distinguishing feature of ASTRAL is that formal proofs are also structured [15], [16].

5 Conclusions and Future Work

This paper introduces ASTRAL, a new formal specification language for realtime systems. The language is an outgrowth of the authors' previous work on realtime formal specification languages.

ASTRAL was designed to support good software engineering practice. The structuring mechanisms that allow one to build modularized specifications of complex systems with layering are an example of this. This modular separation of processes allows for a better understanding of the system design by being able to consider each component separately. It also provides the lead designer with reasonable units to be assigned to different design teams. The modular approach also simplifies the proof process by allowing the process properties to be proved independently and then using these properties to prove the properties of the total system, as expressed in the global specification. The layering also allows the proof of the implementation of a process to be simplified.

ASTRAL also supports good software engineering practice by having both invariants and schedules, which contributes to the reuse of process specifications. That is, because properties expressed in the invariant are guaranteed to hold for the process, regardless of what environment it runs in, the process can be reused in any system design that requires only the properties expressed in the invariant. If in addition the properties expressed in the schedule were necessary, then the system for which the process was being considered would have to assure that the environment and imported variables properties were also satisfied. By splitting the environment dependent properties from the environment independent properties, processes are likely to be reused more than if they were grouped together.

Another feature of the language that supports good software engineering practice is the further assumptions clause. By including this clause the schedule requirements can be proved to be satisfiable without requiring the specifier to include implementation details that are inappropriate for the more abstract level of design. This avoids the common problem of over specifying a system in the early stages of design. In addition, by including more than one further assumptions clause the designer can document several possible implementation approaches in the higher level design.

This paper also describes how to compose two or more ASTRAL system specifications into a more complex realtime system. To accomplish this a compose clause was added to the basic specification language. It was also demonstrated how the compose clause can be used to transform two or more existing system specifications into a new specification for the composite system. This too contributes to reuse.

ASTRAL has been used to specify a number of interesting realtime systems. In [15] the results of using it to formally specify a CCITT system that consists of a packet assembler process and several input processes is reported. The use of ASTRAL as a hardware description language was demonstrated in [10]. In that paper it was used to formally specify a checksum generator and a universal asynchronous receiver transmitter (UART) between a modem and a microprocessor. The first of these examples had four distinct process types and the second had six. At Delft University of Technology (The Netherlands) ASTRAL was used to specify a robot control system [9]. These case studies as well as the phone system example presented in this paper demonstrate the expressiveness and the power of the language. They also show ASTRAL's usefulness for specifying varying types of realtime systems from basic hardware to complete communication systems.

In order to get designers to use formal methods to develop realtime systems it is necessary to provide them with an integrated set of tools for writing and analyzing their specifications. The ASTRAL Software Development Environment (SDE), which is an integrated set of tools based on the ASTRAL formal framework, is intended to meet this need. The tools that make up the support environment are a syntax-directed editor, a specification processor, a verification condition generator (vcg), a mechanical theorem prover, and a browser kit. The SDE offers features that reduce errors and facilitate use throughout all stages of the specification development process. In the initial specification phase, the editor prevents syntax errors and the formatter enhances readability. In the middle phase, the validation function reports type errors, scoping errors, missing parameters, etc., and the vcg component generates the proof obligations needed to prove the specification correct with respect to its critical requirements. Finally, the brows-

ers and compose/build features provide for easy maintenance and reuse of specifications. For more details on the SDE see [43].

Future work will proceed in several directions. One direction is to experiment further with the ASTRAL approach in more complex real life cases. Although the case studies have shown that the language is highly expressive and powerful, more experimentation is needed to further assess the approach.

Another research direction consists of enhancing and extending the theory behind the approach. The relationship between the target physical architecture and the ASTRAL computation model needs to be examined more closely. Which assumptions need to be relaxed or modified in order to fit target architectures more closely? A typical example is the assumption that as soon as a task terminates a transition, the changed values of the exported variables are immediately visible to all other tasks. Another assumption of the computation model that needs to be addressed is the global clock.

APPENDIX A—THE PHONE SYSTEM SPECIFICATION

```
GLOBAL SPECIFICATION Phone_System

PROCESSES
   Phones: array[1 .. Num_Phone] of Phone,
   Centrals: array[1 .. Num_Area] of Central_Control

TYPE
   Positive_Integer IS TYPEDEF p:Integer (p > 0),
   Digit IS TYPEDEF d: Integer (d ≥ 0 & d ≤ 9),
   Digit_List IS LIST OF Digit,
   Connection IS STRUCTURE OF (From_Area, From_Number, To_Area, To_Number: Digit_List),
   Phone_ID IS TYPEDEF pid: ID (IDTYPE(pid) = Phone),
   Central_Control_ID IS TYPEDEF pid: ID (IDTYPE(pid) = Central_Control),
   Enabled_State IS (Idle, Ready_To_Dial, Dialing, Ringing, Waiting, Talk, Calling, Disconnecting, Busy, Alarm),
   Connection_Status IS (Available, In_Progress, Disconnect, Connect, Talking)

CONSTANT
   In_Area(Phone_ID): Central_Control_ID,
   Max_Cust, Num_Phone, Num_Area : Positive_Integer

ENVIRONMENT          /* No more than Max_Cust phone calls are initiated in any 2 sec interval */
   FORALL C: Central_Control_ID
   (SET_SIZE({ SETDEF P: Phone_ID (In_Area(P) = C & Now - 2 ≤ P.Call(Pickup) ≤ Now)} ) ≤ Max_Cust)

SCHEDULE                    /* The phone should receive a dial tone within 2 sec */
   FORALL P:Phone_ID, t, t1, t2: Time
      (  t ≤ t1 ≤ t2
      & past(In_Area(P).Phone_State(P), t) = Idle & Change2(In_Area(P).Phone_State(P), t)
      & P.Offhook & P.End(Pickup, t1) & Change(In_Area(P).Phone_State(P), t2)
      → (past(In_Area(P).Phone_State(P), t2) = Ringing | past(In_Area(P).Phone_State(P), t2) = Ready_To_Dial) & t2 ≤ t1 + 2)

END Phone_System

SPECIFICATION Phone
LEVEL Top_Level

IMPORT
   Digit, Phone_ID, Central_Control_ID, Enabled_State, In_Area,
   Centrals.Phone_State, Centrals.Enabled_Ring_Pulse, Centrals.Enabled_Ringback_Pulse

EXPORT
   Offhook, Next_Digit, Pickup, Enter_Digit, Hangup

CONSTANT
   T1, T2, T3, T4, T5, T6, T7, T8, T9, T10: Time

VARIABLE
   Offhook, Dialtone, Ring, Ringback, Busytone: Boolean,
   Next_Digit: Digit
```

```
DEFINE
   My_Central: Central_Control_ID == In_Area(Self)

ENVIRONMENT
      FORALL t: Time (Call(Pickup, t) → ~past(Offhook, t))          /* To pick up the phone it must not be offhook */
   &  FORALL t: Time (Call(Hangup, t) → past(Offhook, t))           /* To hang up the phone it must be offhook */
   &  FORALL t: Time                                                /* Digits are entered after the dialtone is received, */
      (   Call(Enter_Digit, t)                                      /* LD numbers begins with 1 and have 11 digits, */
      →  (   past(Dialtone, t)                                      /* local numbers have 7 digits, and */
           |  EXISTS t1: Time, n: Integer, D: Digit                 /* the phone is offhook while digits are entered */
                (  2 ≤ n & Call_n(Enter_Digit(D), t1)
                 &  past(Dialtone, t1) & (n ≤ 7 & D ≠ 1 | n ≤ 11 & D = 1) & FORALL t2: Time (t1 ≤ t2 ≤ t → past(Offhook, t2)))))
   &  FORALL t: Time (Call_2(Pickup, t) → Call(Pickup) - Call_2(Pickup) ≥ 1) /* The phone is picked up no more than once a */
                                                                    /* second */

INITIAL
      ~Offhook & ~Dialtone & ~Busytone & ~Ring & ~Ringback

INVARIANT
      ((Dialtone | Ringback | Busytone)→ Offhook)
   &  (Ring → (~Offhook & ~DialTone & ~Ringback & ~Busytone))

SCHEDULE
      (Dialtone → ~Ring & ~Ringback & ~Busytone)
   &  (Busytone →  ~Dialtone & ~Ring & ~Ringback)
   &  (Ringback →  ~Dialtone & ~Ring & ~Busytone)

IMPORTED VARIABLE
      (  My_Central.Phone_State(Self) = Busy                        /* A phone's state can only change to Busy from Dialing */
      →  past(My_Central.Phone_State(Self), Change_2(My_Central.Phone_State(Self))) = Dialing
      &  EXISTS t: Time (Change_2(My_Central.Phone_State(Self)) < t & past(End(Enter_Digit) = Now, t)))

   &  (  My_Central.Phone_State(Self) = Waiting                     /* A phone's state can only change to Waiting from Dialing */
         →  past(My_Central.Phone_State(Self), Change_2(My_Central.Phone_State(Self))) = Dialing
         &  EXISTS t: Time (Change_2(My_Central.Phone_State(Self)) < t & past(End(Enter_Digit) = Now, t)))

   &  (  My_Central.Phone_State(Self) = Dialing                     /* A phone's state can only change to Dialing from Ready_To_Dial */
      →  past(My_Central.Phone_State(Self), Change_2(My_Central.Phone_State(Self))) = Ready_To_Dial
      &  EXISTS t: Time (Change_2(My_Central.Phone_State(Self)) < t & past(End(Enter_Digit) = Now, t)))

   &  (  My_Central.Phone_State(Self) = Ready_To_Dial               /* A phone's state can only change to Ready_To_Dial from Idle */
      →  past(My_Central.Phone_State(Self), Change_2(My_Central.Phone_State(Self))) = Idle)

   &  (  EXISTS t, t1: Time                                         /* From the time a phone's state is idle until it is waiting */
         (  past(My_Central.Phone_State(Self), t) = Idle            /* the phone's ringback pulse must be disabled */
         &  FORALL t1: time
         &           (t > t1 ≤ Now
                  →  past ( My_Central.Phone_State ( Self ), t1 ) ≠ Waiting ) )
      →  ~My_Central.Enabled_Ringback_Pulse(Self))

FURTHER ASSUMPTIONS #1
   FURTHER PROCESS ASSUMPTIONS
      TRANSITION SELECTION
         enabled_transitions CONTAINS any_subset({Stop_Ringback, Stop_Busytone})
      &  TRUE
      →  elgible_transitions = {Stop_Ringback, Stop_Busytone} INTERSECT enabled_transitions

TRANSITION Pickup      T1
ENTRY
      ~Offhook
EXIT
      Offhook & ~Dialtone & ~Busytone
   &  ~Ring & ~Ringback

TRANSITION Start_Ring  T4
ENTRY
~Offhook & ~Ring
& My_Central.Phone_State(Self) = Ringing
& My_Central.Enabled_Ring_Pulse(Self)
EXIT
Ring
```

```
TRANSITION Start_Tone  T2
ENTRY
      Offhook & ~Dialtone
   &  My_Central.Phone_State(Self) = Ready_To_Dial
   &  FORALL t: Time
         (Change(Dialtone,t) → t < Change(Offhook))
EXIT
      Dialtone

TRANSITION Enter_Digit(D:Digit) T3
ENTRY
      Offhook
   & (  My_Central.Phone_State(Self) = Ready_To_Dial
        & Dialtone
      |  My_Central.Phone_State(Self) = Dialing)
EXIT
      Next_Digit = D & ~Dialtone

TRANSITION Stop_Ringback T7
ENTRY
      Ringback
   & ~My_Central.Enabled_Ringback_Pulse(Self)
EXIT
         ~Ringback

TRANSITION Start_Busytone T8
ENTRY
         Offhook & ~Busytone
   &  My_Central.Phone_State(Self) = Busy
EXIT
         Busytone

END Top_Level
END Phone

TRANSITION Stop_Ring T5
ENTRY
       Ring
   & ~My_Central.Enabled_Ring_Pulse(Self)
EXIT
  ~Ring
TRANSITION Start_Ringback T6

ENTRY
      Offhook & ~Ringback
   &  My_Central.Phone_State(Self) = Waiting
   &  My_Central.Enabled_Ringback_Pulse(Self)
EXIT
   Ringback

TRANSITION Stop_Busytone T9
ENTRY
      Busytone
   &  My_Central.Phone_State(Self) ≠ Busy
EXIT
      ~Busytone

TRANSITION Hangup      T10
ENTRY
      Offhook
EXIT
      ~Offhook & ~Dialtone & ~Busytone
   &   ~Ring & ~Ringback
```

```
SPECIFICATION Central_Control
LEVEL Top_Level

IMPORT
   Digit, Digit_List, Connection, Phone_ID, Central_Control_ID, Enabled_State, Connection_Status,
   In_Area, Max_Cust, Phones.Offhook, Phones.Next_Digit, Phones.Pickup, Phones.Enter_Digit

EXPORT
   Phone_State, Enabled_Ring_Pulse, Enabled_Ringback_Pulse, LDOut_Line, LDOut_Status,
   Receive_LD, Start_LD, Start_Talk_2, Terminate_LD_2

TYPE
   Area_Phone IS TYPEDEF p: Phone_ID (In_Area(p) = Self)

CONSTANT
   Uptime_Ring, Downtime_Ring, Uptime_Ringback, Downtime_Ringback, LD_Timeout, Delta: Time,
   Tim1, Tim2, Tim3, Tim4, Tim5, Tim6, Tim7, Tim8, Tim9, Tim10, Tim11, Tim12, Tim13, Tim14, Tim15, Tim16: Time,
   Get_ID(Digit_List): Area_Phone,
   Get_Number(Area_Phone), Get_Area(Central_Control_ID), Pick_Area(Digit_List), Pick_Number(Digit_list): Digit_List

VARIABLE
   Phone_State(Area_Phone): Enabled_State,
   Long_Distance(Area_Phone): Boolean,
   Enabled_Ring_Pulse(Area_Phone): Boolean,
   Enabled_Ringback_Pulse(Area_Phone): Boolean,
   Connected_To(Area_Phone): Area_Phone,
   Number(Area_Phone): Digit_List,
   LDOut_Line(Area_Phone): Connection,
   LDOut_Status(Area_Phone): Connection_Status

AXIOM
   FORALL d: Digit_List (LIST_LEN(Pick_Number(d)) = 7 & LIST_LEN(Pick_Area(d)) = 3)
```

```
DEFINE
   Plug(L1, L2: Connection):Boolean ==                          /* L1 is connected to L2 */
         L1[From_Area] = L2[To_Area]
      & L1[From_Number] = L2[To_Number]
      & L1[To_Area] = L2[From_Area]
      & L1[To_Number] = L2[From_Number]

   Count(P: Area_Phone): Integer == LIST_LEN(Number(P)),

   Calling_Out(P: Area_Phone, L: Connection): Boolean ==        /* Phone P is making a long distance call through line L */
         P.Offhook
      & Long_Distance(P)
      & Get_Area(Self) = L[To_Area]
      & Get_Number(P) = L[To_Number]
      & Plug(LDOut_Line(P), L)

ENVIRONMENT
   FORALL t: Time, L: Connection                    /* The termination of a LD call implies that someone has initiated it */
   (  Call(Terminate_LD_Call_2(L), t)
   → EXISTS t1: Time, LS: Connection_Status (t1 < t & (Call(Receive_LD(L), t1) | Call(Start_LD(L,LS), t1))))

&  FORALL t: Time, L: Connection                    /* For any LD call, starting to talk requires that the call has occurred */
   (Call(Start_Talk_2(L), t) → EXISTS t1: Time, LS: Connection_Status (t1 < t & Call(Start_LD(L, LS), t1)))

&  FORALL t: Time, L: Connection, LS: Connection_Status
   (  Call(Start_LD(L, LS), t)
   →  EXISTS t1: Time, P: Area_Phone (t1 < t & Start(Connect_Long_Distance(P), t1) & past(Calling_Out(P, L), t)))

&  FORALL t: Time (Call2(Receive_LD, t) → Call(Receive_LD) –  t > LD_Timeout)    /* LD_timeout is the time between two */
                                                                                 /* subsequent LD calls received by central control*/

INITIAL
   FORALL P: Area_Phone
      (  Phone_State(P) = Idle & Number(P) = NIL
      & ~Enabled_Ring_Pulse(P)
      & ~Enabled_Ringback_Pulse(P)
      & ~Long_Distance(P)
      & LDOut_Status(P) = Available)

INVARIANT
   FORALL P: Area_Phone
   (  (Long_Distance(P)  →  Count(P) ≥ 0 & Count(P) ≤ 11 )
   & (~Long_Distance(P)  →  (  Count(P) ≥ 0 & Count(P) ≤ 7 )
                            & (Phone_State(P) = Waiting  →  Phone_State(Connected_To(P)) = Ringing)
                            & (Phone_State(P) = Ringing  →  Phone_State(Connected_To(P)) = Waiting)
                            & (Phone_State(P) = Talk →  Phone_State(Connected_To(P)) = Talk))))

CONSTRAINT                                          /* The only way in which P can change its state is by becoming idle */
   FORALL P: Area_Phone
   (  (Phone_State'(P) = Busy | Phone_State'(P) = Alarm | Phone_State'(P) = Disconnect)
   &  Phone_State(P) ≠ Phone_State'(P)
   → Phone_State(P) = Idle )

SCHEDULE
   FORALL P: Area_Phone, t, t1, t2: Time            /* The maximum time to give the dial tone to a phone */
   (  t ≤ t1 ≤ t2
   &  Change2(Phone_State(P), t) & past(Phone_State(P), t) = Idle
   &  P.End(Pickup, t1) & P.Offhook
   &  Change(Phone_State(P), t2)
   → t2 ≤ t1 + 2
   &  (past(Phone_State(P), t2) = Ringing | past(Phone_State(P), t2) = Ready_To_Dial))

&  FORALL P: Area_Phone                            /* Phone ringing is enabled every Downtime_Ring time units */
   (  Phone_State(P) = Ringing
   &  Now - Change(Phone_State(P)) ≥ Downtime_Ring
   →  EXISTS n: Integer
      (  Endn(Enable_Ring(P)) > Change(Phone_State(P))
      &  Endn(Enable_Ring(P)) ≤ Change(Phone_State(P)) + Downtime_Ring ) )
```

```
& FORALL P: Area_Phone                          /* Phone ringing is disabled every Uptime_Ring time units */
  ( Phone_State(P) = Ringing
  & End(Enable_Ring(P)) > Change(Phone_State(P))
  & Now ≥ End(Enable_Ring(P)) + Uptime_Ring + Delta
  →   End(Disable_Ring_Pulse(P)) ≥ End(Enable_Ring(P)) + Uptime_Ring
    & End(Disable_Ring_Pulse(P)) ≤ End(Enable_Ring(P)) + Uptime_Ring + Delta)

& FORALL P: Area_Phone                          /*Ringback tone is enabled every Downtime_Ringback time units */
( Phone_State(P) = Ringing
& End(Disable_Ring_Pulse(P)) > Change(Phone_State(P))
& Now ≥ End(Disable_Ring_Pulse(P)) + Downtime_Ring + Delta
→   End(Enable_Ring(P)) ≥ End(Disable_Ring_Pulse(P)) + Downtime_Ring
  & End(Enable_Ring(P)) ≤ End(Disable_Ring_Pulse(P)) + Downtime_Ring+ Delta)

& FORALL P: Area_Phone   /* The phone rings no later than Downtime_ring time units after the connection is established */
  ( ~Long_Distance(P)
  & Phone_State(P) = Waiting
  & Now - End(Process_Local_Call(P)) ≥ Downtime_Ring
  → EXISTS n, m: Integer
    ( End_n(Enable_Ring(Connected_To(P))) > End(Process_Local_Call(P))
    & End_n(Enable_Ring(Connected_To(P))) ≤ End(Process_Local_Call(P)) + Downtime_Ring
    & End_m(Enable_Ringback(P)) > End(Process_Local_Call(P))
    & End_m(Enable_Ringback(P)) ≤ End_n(Enable_Ring(Connected_To(P))) + 0.5))

& FORALL P: Area_Phone                          /*Ringback tone is disabled every Uptime_Ringback time units */
  ( Phone_State(P) = Waiting
  & End(Enable_Ringback(P)) > Change(Phone_State(P))
  & Now ≥ End(Enable_Ringback(P)) + Uptime_Ringback + Delta
  → End(Disable_Ringback_Pulse(P)) ≥ End(Enable_Ringback(P)) + Uptime_Ringback
  & End(Disable_Ringback_Pulse(P)) ≤ End(Enable_Ringback(P)) + Uptime_Ringback + Delta )

& FORALL P: Area_Phone                          /* The ringback tone starts no later than Downtime_ringback time units */
  ( Phone_State(P) = Waiting                    /* after the connection is established */
  & End(Disable_Ringback_Pulse(P)) > Change(Phone_State(P))
  & Now ≥ End(Disable_Ringback_Pulse(P)) + Downtime_Ringback + Delta
  → End(Enable_Ringback(P)) ≥ End(Disable_Ringback_Pulse(P)) +Downtime_Ringback
  & End(Enable_Ringback(P)) ≤ End(Disable_Ringback_Pulse(P)) +Downtime_Ringback + Delta )

IMPORTED VARIABLE CLAUSE                        /* No more than Max_Cust phones are picked up in a 2" interval */
  SET_SIZE( { SETDEF P: Area_Phone (Now – 2 ≤ P.Start(Pickup) ≤ Now)} ) ≤ Max_Cust

FURTHER ASSUMPTIONS #1
  FURTHER PROCESS ASSUMPTIONS
    TRANSITION SELECTION                        /* Gives priority to Give_Dial_Tone over all other transitions */
    enabled_transitions CONTAINS {Give_Dial_Tone }
    & TRUE
    → elgible_transitions = {Give_Dial_Tone}
  CONSTANT REFINEMENT
    2 > MAX(Tim1, …, Tim16) + (Max_Cust + 1)*Tim1

TRANSITION Give_Dial_Tone(P:Area_Phone) Tim1     /* P has picked up the phone */
ENTRY
    P.Offhook & Phone_State(P) = Idle
EXIT
    Phone_State(P) BECOMES Ready_To_Dial & Number(P) BECOMES NIL

TRANSITION Process_Digit(P:Area_Phone) Tim2      /* P has entered a digit */
ENTRY
    P.Offhook
  & ( (Long_Distance(P) & Count(P) < 11) | (~Long_Distance(P) & Count(P) < 7))
  & ( ( Phone_State(P) = Ready_To_Dial & P.End(Enter_Digit) > End(Give_Dial_Tone(P)))
    | ( Phone_State(P) = Dialing & P.End(Enter_Digit) > End(Process_Digit(P))))
EXIT
    IF Count'(P) = 0
    THEN
      IF P.Next_Digit' = 1
```

```
            THEN Long_Distance(P) BECOMES True
            ELSE Long_Distance(P) BECOMES False
            FI
            & Phone_State(P) BECOMES Dialing
        FI
    &  Number(P) BECOMES Number'(P) CONCAT LISTDEF(P.Next_Digit')

TRANSITION Process_Local_Call(P: Area_Phone)   Tim3                    /* P has dialed a local number */
ENTRY
        P.Offhook & ~Long_Distance(P) & Count(P) = 7 & Phone_State(P) = Dialing
    &  ~Get_ID(Number(P)).Offhook & Phone_State(Get_ID(Number(P))) = Idle
EXIT
        Phone_State(Get_ID(Number'(P))) = Ringing & Phone_State(P) = Waiting & ~Long_Distance(Get_ID(Number'(P)))
    &  Connected_To(P) = Get_ID(Number'(P)) & Connected_To(Get_ID(Number'(P))) = P
    &  FORALL P1: Area_Phone
            (P1 ≠ P & P1 ≠ Get_ID(Number'(P)) → NOCHANGE(Phone_State(P1)) & NOCHANGE(Connected_To(P1)))
EXCEPT
        P.Offhook & ~Long_Distance(P) & Count(P) = 7 & Phone_State(P) = Dialing
        &  (Get_ID(Number(P)).Offhook | Phone_State(Get_ID(Number(P))) ≠ Idle)
EXIT
        Phone_State(P) BECOMES Busy

TRANSITION Connect_Long_Distance(P:Area_Phone) Tim4                /*P has dialed a LD number */
ENTRY
        P.Offhook & Long_Distance(P) & Count(P) = 11
    &  Phone_State(P) = Dialing & Pick_Area(Number(P)) ≠ Get_Area(Self)
EXIT
        LDOut_Line(P)[From_Area] = Get_Area(Self) & LDOut_Line(P)[From_Number] = Get_Number(P)
    &  LDOut_Line(P)[To_Area] = Pick_Area(Number'(P)) & LDOut_Line(P)[To_Number] = Pick_Number(Number'(P))
    &  LDOut_Status(P) BECOMES In_Progress & Phone_State(P) BECOMES Calling
    &  FORALL P1: Area_Phone (P1 ≠ P → NOCHANGE(LDOut_Line(P1)))
EXCEPT
        P.Offhook & Long_Distance(P) & Count(P) = 11
    &  Phone_State(P) = Dialing & Pick_Area(Number(P)) = Get_Area(Self)
EXIT
        Long_Distance(P) BECOMES False & Number(P) BECOMES Pick_Number(Number'(P))

TRANSITION Enable_Ring(P:Area_Phone) Tim5                          /* Makes phone P ringing */
ENTRY
        ~P.Offhook & Phone_State(P) = Ringing & ~Enabled_Ring_Pulse(P)
    &  FORALL t: Time
        (  End(Disable_Ring_Pulse(P), t) & FORALL t1 : Time (t ≤ t1 ≤ Now → past(Phone_State(P), t1) = Ringing)
        →  Now – t ≥ Downtime_Ring)
EXIT
        Enabled_Ring_Pulse(P) BECOMES True

TRANSITION Disable_Ring_Pulse(P: Area_Phone)     Tim6              /* Prevent phone P from ringing */
ENTRY
        Enabled_Ring_Pulse(P) & (P.Offhook | Now – End(Enable_Ring(P)) ≥ Uptime_Ring)
EXIT
        Enabled_Ring_Pulse(P) BECOMES False

TRANSITION Enable_Ringback(P:Area_Phone) Tim7  /* The ringback tone is started */
ENTRY
        P.Offhook & Phone_State(P) = Waiting & ~Enabled_Ringback_Pulse(P)
    &  FORALL t: Time
        (  End(Disable_Ringback_Pulse(P), t) & FORALL t1 : Time (t ≤ t1 ≤ Now → past(Phone_State(P), t1) = Waiting)
        →  Now - t ≥ Downtime_Ringback)
EXIT
        Enabled_Ringback_Pulse(P) BECOMES True

TRANSITION Disable_Ringback_Pulse(P: Area_Phone) Tim8              /* Prevent the ringback tone */
ENTRY
        Enabled_Ringback_Pulse(P) & (~P.Offhook | Now - End(Enable_Ringback(P)) ≥ Uptime_Ringback)
EXIT
        Enabled_Ringback_Pulse(P) BECOMES False
```

```
TRANSITION Receive_LD (LDIn_Line: Connection)  Tim9          /* An incoming LD call has arrived from line LDIn_Line */
ENTRY
      LDIn_Line[To_Area] = Get_Area(Self)
   &  Phone_State(Get_ID(LDIn_Line[To_Number])) = Idle
   &  ~Get_ID(LDIn_Line[To_Number]).Offhook
EXIT
      Phone_State(Get_ID(LDIn_Line[To_Number])) BECOMES Ringing
   &  LDOut_Status(Get_ID(LDIn_Line[To_Number])) BECOMES Connect
   &  Long_Distance(Get_ID(LDIn_Line[To_Number]))
   &  Plug(  LDOut_Line(Get_ID(LDIn_Line[To_Number])), LDIn_Line)
   &  FORALL P: Area_Phone (P ≠ Get_ID(LDIn_Line[To_Number]) → NOCHANGE(LDOut_Line(P)))

TRANSITION Start_Talk_1(P: Area_Phone)  Tim10                /* Phone P was ringing and someone picked it up */
ENTRY
      P.Offhook & Phone_State(P) = Ringing
EXIT
      Phone_State(P) = Talk
   &  IF ~Long_Distance'(P)
      THEN
         Phone_State(Connected_To'(P)) = Talk
         &  FORALL P1:Area_Phone (P1 ≠ P & P1 ≠ Connected_To'(P) → NOCHANGE(Phone_State(P1)))
      ELSE LDOut_Status(P) BECOMES Talking
      FI

TRANSITION Start_Talk_2(LDIn_Line:Connection)  Tim11          /* Phone P has made a LD call through LDIn_Line */
ENTRY                                                        /* and someone has answered the call */
      EXISTS P: Area_Phone (Calling_Out(P, LDIn_Line) & Phone_State(P) = Waiting & LDOut_Status(P) = Connect)
EXIT
      EXISTS P: Area_Phone
      (  Calling_Out'(P,LDIn_Line) & Phone_State'(P) = Waiting & LDOut_Status'(P) = Connect
      &  LDOut_Status(P) BECOMES Talking & Phone_State(P) BECOMES Talk)

TRANSITION Start_LD(LDIn_Line: Connection, LDIn_Status: Connection_Status)  Tim12  /* Phone P made a LD call and the */
ENTRY                                                                             /* connection is established */
      LDIn_Status = Connect
   &  EXISTS P: Area_Phone (Calling_Out(P, LDIn_Line) & Phone_State(P) = Calling & LDOut_Status(P) = In_Progress)
EXIT
      EXISTS P: Area_Phone
      (  Calling_Out'(P, LDIn_Line) & Phone_State'(P) = Calling & LDOut_Status'(P) = In_Progress
      &  LDOut_Status(P) BECOMES Connect & Phone_State(P) BECOMES Waiting)
EXCEPT
      LDIn_Status = Disconnect
   &  EXISTS P: Area_Phone (Calling_Out(P, LDIn_Line) & Phone_State(P) = Calling & LDOut_Status(P) = In_Progress)
EXIT
      EXISTS P: Area_Phone
      (  Calling_Out'(P, LDIn_Line) & Phone_State'(P) = Calling & LDOut_Status'(P) = In_Progress
      &  LDOut_Status(P) BECOMES Disconnect & Phone_State(P) BECOMES Busy)

TRANSITION Terminate_LD_1(P: Area_Phone)  Tim13               /* P has ended the LD call that it initiates */
ENTRY
      ~P.Offhook & Long_Distance(P) & Phone_State(P) ≠ Idle & Phone_State(P) ≠ Ringing
   &  LDOut_Line(P)[From_Area] = Get_Area(Self) & LDOut_Line(P)[From_Number] = Get_Number(P)
   &  LDOut_Status(P) ≠ Available
EXIT
      Phone_State(P) BECOMES Idle & LDOut_Status(P) BECOMES Available & ~Enabled_Ringback_Pulse(P)

TRANSITION Generate_Alarm(P: Area_Phone)  Tim14
ENTRY
      P.Offhook
   &  (  Phone_State(P) = Ready_To_Dial
      |  (Phone_State(P) = Dialing & P.Call(Enter_Digit) < Start(Process_Digit(P))))
   &  (  Count(P) = 0 & (Now-End(Give_Dial_Tone(P))) > 30
      |  (  Count(P) > 0
         &  ( ~Long_Distance(P) & Count(P) < 7 | Long_Distance(P) & Count(P) < 11)
         &  Now - End(Process_Digit(P)) > 20)
```

```
      | Now – End(Give_Dial_Tone(P))) > 100 )
EXIT
      Phone_State(P) BECOMES Alarm

TRANSITION Terminate_Local_Call(P: Area_Phone)  Tim15          /* P has terminated a local call */
ENTRY
      ~P.Offhook & ~Long_Distance(P) & Phone_State(P) ≠ Idle & Phone_State(P) ≠ Ringing
EXIT
      Phone_State(P) = Idle
   &  ~Enabled_Ringback_Pulse(P)
   &  IF Phone_State'(P) = Talk | Phone_State'(P) = Waiting
      THEN
         IF Phone_State'(P) = Talk
         THEN Phone_State(Connected_To'(P)) = Disconnecting
         ELSE Phone_State(Connected_To'(P)) = Idle & ~Enabled_Ring_Pulse(Connected_To'(P))
         FI
      &  FORALL P1: Area_Phone (P1 ≠ P & P1 ≠ Connected_To'(P) → NOCHANGE(Phone_State(P1)))
      ELSE FORALL P1: Area_Phone (P1 ≠ P → NOCHANGE(Phone_State(P1)))
      FI

TRANSITION Terminate_LD_2(LDIn_Line:Connection) Tim16          /* P's correspondent terminated the LD call */
ENTRY
      EXISTS P: Area_Phone (Calling_Out(P, LDIn_Line) & Phone_State(P) = Talk & LDOut_Status(P) = Talking)
EXIT
      EXISTS P: Area_Phone
      (  Calling_Out'(P, LDIn_Line) & Phone_State'(P) = Talk & LDOut_Status'(P) = Talking
      &  LDOut_Status(P) BECOMES Disconnect & Phone_State(P) BECOMES Disconnecting)
EXCEPT
      EXISTS P: Area_Phone (Calling_Out(P, LDIn_Line) & Phone_State(P) = Ringing & LDOut_Status(P) = Connect)
EXIT
      EXISTS P: Area_Phone
      (  Calling_Out'(P,LDIn_Line) & Phone_State'(P) = Ringing & LDOut_Status'(P) = Connect
      &  LDOut_Status(P) BECOMES Available & Phone_State(P) BECOMES Idle & ~Enabled_Ring_Pulse(P))

END Top_Level
END Central_Control
```

APPENDIX B – THE LONG DISTANCE NETWORK

```
GLOBAL SPECIFICATION Long_Distance_Network

PROCESSES
   Units: array[1 .. Num_Area] of Long_Distance_Unit

TYPE
   Digit IS TYPEDEF d: Integer (d ≥ 0 & d ≤ 9),
   Positive_Integer IS TYPEDEF p: Integer (p > 0),
   Line,
   LDU_ID IS TYPEDEF pid: ID (IDTYPE(pid) = Long_Distance_Unit),
   Digit_List IS LIST OF Digit,
   Connection_Status = (Available, In_Progress, Disconnected, Connected, Talk),
   Connection IS STRUCTURE OF (From_Area, From_Number, To_Area, To_Number: Digit_List)

CONSTANT
   Num_Area: Positive_Integer

DEFINE
      Connect(L1, L2: Connection): Boolean ==
         L1[From_Area] = L2[From_Area]
      &  L1[From_Number] = L2[From_Number]
      &  L1[To_Area] = L2[To_Area]
      &  L1[To_Number] = L2[To_Number],

   Disconnect(L: Connection): Boolean ==
         L[From_Area] = NIL
      &  L[To_Area] = NIL
      &  L[From_Number] = NIL
```

```
        & L[To_Number] = NIL,

    Plug(L1, L2: Connection): Boolean ==
            L1[From_Area] = L2[To_Area]
        & L1[From_Number] = L2[To_Number]
        & L1[To_Area] = L2[From_Area]
        & L1[To_Number] = L2[From_Number]
END Long_Distance_Network

SPECIFICATION Long_Distance_Unit
LEVEL Top_Level

IMPORT
    Digit, Digit_List, Line, LDU_ID, Connection_Status, Connection, Plug, Connect, Disconnect

EXPORT
    NetOut, LocOut, NetStatus, LocStatus, Receive_Local_Req, Local_Established, Started_Local_Talk, End_Local

VARIABLE
    NetOut(Line), LocOut(Line): Connection,
    NetStatus(Line), LocStatus(Line): Connection_Status

CONSTANT
    Ti1, Ti2, Ti3, Ti4, Ti5, Ti6, Ti7, Ti8, Ti9, Timeout, Max_Calls: Time,
    Get_Area(LDU_ID): Digit_List

ENVIRONMENT
    FORALL t: Time, L: Connection  /* Relates the calls to exported variables to the value of NetOut,LocOut and NetStatus */
    (  Call(End_Local(L), t) → EXISTS L1: Line (Connect(past(NetOut(L1), t), L) & past(NetStatus(L1), t) ≠ Available)
    &  Call(Local_Established(L), t) → EXISTS L1:Line (Plug(past(LocOut(L1), t), L) & past(NetStatus(L1), t) ≠ Connected)
    &  Call(Started_Local_Talk(L), t)→ EXISTS L1:Line (Plug(past(LocOut(L1), t), L) & past(NetStatus(L1), t) = Connected))

&  FORALL t:Time (Call2(Receive_Local_Req, t) → Call(Receive_Local_Req) - Call2(Receive_Local_Req) > Max_Calls)

INITIAL
    FORALL L: Line
    (Disconnect(NetOut(L)) & Disconnect(LocOut(L)) & NetStatus(L) = Available & LocStatus(L) = Available)

TRANSITION Receive_Local_Req (In_Line: Connection) Ti1
ENTRY
    In_Line[From_Area] = Get_Area(Self)
EXIT
    EXISTS L: Line
    (  NetStatus'(L) = Available
    &  Connect(NetOut(L), In_Line)
    &  NetStatus(L) BECOMES In_Progress
    &  FORALL L1:Line(L1 ≠ L → NOCHANGE(NetOut(L1))))

TRANSITION Receive_Long_Req Ti2
ENTRY
    EXISTS U: LDU_ID, L: Line
    (  U.NetOut(L)[To_Area] = Get_Area(Self)
    &  U.NetStatus(L) = In_Progress
    &  FORALL L1:Line(~Connect(U.NetOut(L), LocOut(L1))))
EXIT
    EXISTS U: LDU_ID, L: Line
    (  U.NetOut'(L)[To_Area] = Get_Area(Self)
    &  U.NetStatus'(L) = In_Progress
    &  FORALL L1: Line(~Connect(U.NetOut'(L), LocOut'(L1))
    &  Connect(LocOut(L), U.NetOut'(L))
    &  LocStatus(L) BECOMES In_Progress
    &  FORALL L1: Line(L1 ≠ L → NOCHANGE(LocOut(L1))))

TRANSITION Local_Established (In_Line: Connection) Ti3
ENTRY
    In_Line[From_Area] = Get_Area(Self)
&  EXISTS L: Line (Plug(LocOut(L), In_Line) & NetStatus(L) ≠ Connected)
EXIT
```

```
    EXISTS L: Line
    (  Plug(LocOut'(L), In_Line)
    &  NetStatus'(L) ≠ Connected
    &  NetStatus(L) BECOMES Connected
    &  Connect(NetOut(L), In_Line)
    &  FORALL L1: Line(L1 ≠ L → NOCHANGE(NetOut(L1))))

TRANSITION Long_Established Ti4
ENTRY
    EXISTS U: LDU_ID, L: Line
    ( U.NetOut(L)[To_Area] = Get_Area(Self)
    &  U.NetStatus(L) = Connected & LocStatus(L) ≠ Connected)
EXIT
    EXISTS U: LDU_ID, L: Line
    ( U.NetOut'(L)[To_Area] = Get_Area(Self)
    &  U.NetStatus'(L) = Connected & LocStatus'(L) ≠ Connected
    &  LocStatus(L) BECOMES Connected
    &  Connect(LocOut(L), U.NetOut'(L))
    &  FORALL L1:Line (L1 ≠ L → NOCHANGE(LocOut(L1))))

TRANSITION Started_Local_Talk (In_Line: Connection) Ti6
ENTRY
    EXISTS L: Line (Plug(LocOut(L), In_Line) & NetStatus(L) = Connected)
EXIT
    EXISTS L: Line
    (  Plug(LocOut'(L), In_Line' & NetStatus'(L) = Connected
    &  NetStatus(L) BECOMES Talk)

TRANSITION Send_Local_Timeout Ti5
ENTRY
    EXISTS L: Line
    (  NetStatus(L) = In_Progress & LocStatus(L) = Available
    &  Now-End(Receive_Loc_Req(NetOut(L))) ≥ Timeout)
EXIT
    EXISTS L: Line
    (  NetStatus'(L) = In_Progress
    &  Now-End(Receive_Local_Req(NetOut'(L))) ≥ Timeout + Ti5
    &  LocStatus(L) BECOMES Disconnected
    &  NetStatus(L) BECOMES Available)

TRANSITION Started_Long_Talk Ti7
ENTRY
    EXISTS U: LDU_ID, L: Line
    (  Connect(U.NetOut(L), LocOut(L))
    &  U.NetStatus(L) = Talk
    &  LocStatus(L) = Connected)
EXIT
    EXISTS U: LDU_ID, L: Line
    (  Connect(U.NetOut'(L), LocOut'(L))
    &  U.NetStatus'(L) = Talk
    &  LocStatus(L) BECOMES Talk)

TRANSITION End_Local (In_Line: Connection) Ti8
ENTRY
    EXISTS L: Line (Connect(NetOut(L), In_Line))
EXIT
    EXISTS L: Line
    ( Connect(NetOut'(L), In_Line')
    &  Disconnect(NetOut(L))
    &  Disconnect(LocOut(L))
    &  NetStatus(L) BECOMES Available
    &  LocStatus(L) BECOMES Available
    &  FORALL L1: Line (L1 ≠ L → NOCHANGE(LocOut(L1)) & NOCHANGE(NetOut(L1))))

TRANSITION End_Long Ti9
```

```
ENTRY
    EXISTS L: Line
    ( ~Disconnect(LocOut(L))
    & LocStatus(L) ≠ Available
    & FORALL U: LDU_ID, L1: Line (~Connect(U.NetOut(L1), LocOut(L))))
EXIT
    EXISTS L: Line
    ( ~Disconnect(LocOut'(L))
    & LocStatus'(L) ≠ Available
    & FORALL U: LDU_ID, L1: Line (~Connect(U.NetOut'(L1), LocOut'(L)))
    & Disconnect(LocOut(L)) & Disconnect(NetOut(L))
    & LocStatus(L) BECOMES Available
    & NetStatus(L) BECOMES Available
    & FORALL L1: Line (L1 ≠ L → NOCHANGE(LocOut(L1)) & NOCHANGE(NetOut(L1))))

END Top_Level
END Long_Distance_Unit
```

APPENDIX C—THE COMPOSE SECTION

```
COMPOSITION OF Phone_System, Long_Distance_Network AS Phone_Network

PROCESSES
    Phones: array[1 .. Num_Phone] of Phone,
    Centrals: array[1 .. Num_Area] of Central_Control
    Units: array[1 .. Num_Area] of Long_Distance_Unit

CONSTANT
LD_Unit(Central_Control_ID): LDU_ID

AXIOM
    FORALL C1, C2: Central_Control_ID (LD_Unit(C1) = LD_Unit(C2) → C1 = C2)

TRIGGERS
    CALL GENERATION FOR Long_Distance_Unit.Receive_Local_Req
        FORALL t: Time, P: Phone_ID
        ( Change(In_Area(P).LDOut_Status(P),t) & In_Area(P).LDOut_Status(P) = In_Progress
        ↔ LD_Unit(In_Area(P)).Call(Receive_Local_Req(In_Area(P).LDOut_Line(P)), t))

    CALL GENERATION FOR Long_Distance_Unit.Local_Established
        FORALL t: Time, P: Phone_ID
        ( Change(In_Area(P).LDOut_Status(P), t) & In_Area(P).LDOut_Status(P) = Connected
        & Change(In_Area(P).Phone_State(P), t) & In_Area(P).Phone_State(P) = Ringing
        ↔ LD_Unit(In_Area(P)).Call(Local_Established(In_Area(P).LDOut_Line(P)), t))

    CALL GENERATION FOR Long_Distance_Unit.Started_Local_Talk
        FORALL t: Time, P: Phone_ID
        ( Change(In_Area(P).LDOut_Status(P), t) & In_Area(P).LDOut_Status(P) = Talk
        & EXISTS t1: Time
        ( FORALL t2 : Time (t2 ≥ t1 & t2 < t → ~past(Change(In_Area(P).LDOut_Status(P), t2), t2))
        & past(In_Area(P).LDOut_Status(P), t1) = Connected & past(In_Area(P).Phone_State(P), t1) = Ringing)
        ↔ LD_Unit(In_Area(P)).Call(Started_Local_Talk(In_Area(P).LDOut_Line(P)), t))

    CALL GENERATION FOR Long_Distance_Unit.End_Local
        FORALL t: Time, P: Phone_ID
        ( Change(In_Area(P).LDOut_Status(P), t) & In_Area(P).LDOut_Status(P) = Available
        ↔ LD_Unit(In_Area(P)).Call(End_Local(In_Area(P).LDOut_Line(P)), t))

    CALL GENERATION FOR Central_Control.Receive_LD
        FORALL t: Time, C: Central_Control_ID, L: Line
        ( Change(LD_Unit(C).LocStatus(L), t) & LD_Unit(C).LocStatus(L) = In_Progress
        ↔ C.Call(Receive_LD(LD_Unit(C).LocOut(L)), t))

    CALL GENERATION FOR Central_Control.Start_LD
        FORALL t: Time, C: Central_Control_ID, L: Line
        ( Change(LD_Unit(C).LocStatus(L), t) & past(LD_Unit(C).LocStatus(L), t) = Connected & ~Change(LD_Unit(C).NetOut(L), t)
        & (LD_Unit(C).NetStatus(L) = Connected | LD_Unit(C).NetStatus(L) = Available)
```

↔ C.Call(Start_LD(LD_Unit(C).LocOut(L), LD_Unit(C).LocStatus(L)), t))

CALL GENERATION FOR Central_Control.Terminate_LD_2
FORALL t: Time, L: Line, P: Phone_ID
(Change(LD_Unit(In_Area(P)).LocOut(L), t) & past(LD_Unit(In_Area(P)).LocStatus(L), t) = Available
& Change(LD_Unit(In_Area(P)).NetOut(L), t) & past(LD_Unit(In_Area(P)).NetStatus(L), t) = Available
& (past(In_Area(P).LDOut_Status(P), t) = Talk | past(In_Area(P).LDOut_Status(P), t) = Connected)
↔ In_Area(P).Call(Terminate_LD_Call_2(LD_Unit(In_Area(P)).LocOut(L)), t))

END COMPOSITION

6 Acknowledgments

The authors would like to thank Steve Eckmann, Miguel Felder, Dino Mandrioli, Angelo Morzenti, and Pierluigi San Pietro who reviewed earlier versions of this paper and provided many useful suggestions. Thanks also to Paul Kolano, Richard Lee, and Marco Mussini who worked on the ASTRAL software development environment. Finally, thanks go to the anonymous reviewers who gave many useful suggestions and comments on the first version of this paper.

This paper is an extended version of two previous papers—Ghezzi and Kemmerer [27], and Coen-Porisini and Kemmerer [14]. This research was started while Carlo Ghezzi was visiting the Reliable Software Group at the University of California at Santa Barbara. This research was partially supported by the National Computer Security Center under Grant No. MDA904-88-C-6006, by the National Science Foundation under Grant No. CCR-9204249, CNR–Comitato Nazionale per la Scienza e le Tecnologie dell'Informazione, and CNR–Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo.

References

[1] R. Alur, C. Courcoubetis, and D. Dill, "Model-Checking for Real-time Systems," *Proc. Fifth IEEE LICS '90*, pp. 414-425, IEEE, 1990.

[2] R. Alur and D. Dill, "The Theory of Timed Automata," *Lecture Notes in Computer Science 600*, pp. 45–73. Springer-Verlag, 1991.

[3] R. Alur, T. Feder, and T. Henzinger, "The Benefits of Relaxing Punctuality," *Proc. 10th ACM Symp. Principles of Distributed Computing*, pp. 139–152, 1991.

[4] B. Auernheimer and R.A. Kemmerer, "ASLAN User's Manual," TRCS84-10, Dept. of Computer Science, Univ. of California, Santa Barbara, Apr. 1992.

[5] B. Auernheimer and R.A. Kemmerer, "Procedural and Nonprocedural Semantics of the ASLAN Formal Specification Language," *Proc. 19th Ann. Hawaii Int'l Conf. System Sciences*, Honolulu, Hawaii, Jan. 1986.

[6] B. Auernheimer and R.A. Kemmerer, "RT-ASLAN: A Specification Language for Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 12, no. 9, Sept. 1986.

[7] M. Abadi and L. Lamport, "An Old-Fashioned Recipe for Real-time." *Lecture Notes on Computer Science 600*, pp. 1–27. Springer-Verlag, 1991.

[8] M. Abadi and L. Lamport, "Conjoining Specifications" *ACM Trans. Programming Languages and Systems*, vol. 17, no. 3, pp. 507–534, May 1995.

[9] K. Brink, L. Bun, J. van Katwijk, and W.J. Toetenel, "Hybrid Specification of Control Systems," *First IEEE Int'l Conf. Eng. Complex Computer Systems*, Ft. Lauderdale, Fla., Nov. 1995.

[10] G. Buonanno, A. Coen-Porisini, and W. Fornaciari, "Hardware Specification Using the Assertion Language ASTRAL," *Proc. Advanced Research Workshop Correct Hardware Design Methodologies*, Torino, Italy, June 1991.

[11] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, vol. 21, no. 3, Sept. 1989.

[12] F. Boussinot and R. de Simone, "The ESTEREL Language." *Proc. IEEE*, vol. 79, no 9, Sept. 1991.

[13] K.M. Chandi and J. Misra, *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[14] A. Coen-Porisini and R.A. Kemmerer, "The Composability of ASTRAL Realtime Specifications," *Proc. Int'l Symp. Software Testing and Analysis*, Cambridge, Mass., July 1993.

[15] A. Coen-Porisini, R.A. Kemmerer, and D. Mandrioli, "A Formal Framework for ASTRAL Intra-Level Proof Obligations," *IEEE Trans. Software Eng.*, vol. 20, no. 8, Aug. 1994.

[16] A. Coen-Porisini, R.A. Kemmerer, and D. Mandrioli, "A Formal Framework for ASTRAL Inter-level Proof Obligations," *Proc. Fifth European Software Eng. Conf.*, Barcelona, Spain, Sept. 1995.

[17] A. Coen-Porisini, P. San Pietro, and R. Kemmerer, "Formal Semantics Definition for ASTRAL," Report No. TRCS 94-15, Dept. of Computer Science, Univ. of California, Santa Barbara, Sept. 1994.

[18] B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods for Validating Them," *IEEE Trans. Software Eng.*, vol. 11, no. 1, Jan. 1985.

[19] L.K. Dillon, G.S. Avrunin, and J.C. Wileden, "Constrained Expressions: Toward Broad Applicability of Analysis Methods for Distributed Software Systems," *ACM Trans. Programming Languages and Systems*, vol. 10, no. 3, pp. 374–402, July 1988.

[20] J. Douglas and R.A. Kemmerer, "Aslantest: A Symbolic Execution Tool for Testing Aslan Formal Specifications," *Proc. Int'l Symp. Software Testing and Analysis*, Seattle, Wash., Aug. 1994.

[21] G. De Pietro and U. Villano, "A Clock Synchronization Algorithm for the Performance Analysis of Multicomputer Systems," *Concurrency: Practice and Experience*, vol. 6, no. 8, pp. 653–671, Dec. 1994.

[22] S.R. Faulk and D.L. Parnas, "On Synchronization in Hard Real-Time Systems," *Comm. ACM*, vol. 31, no. 3, Mar. 1988.

[23] M. Felder and A. Morzenti, "Validating Real-Time Systems by History Checking TRIO Specifications," *ACM Trans. Software Eng. and Methodologies*, vol. 3, no. 4 , Oct. 1994.

[24] M. Felder, D. Mandrioli, and A. Morzenti, "Proving Properties of Realtime Systems through Logical Specifications and Petri Net Models," *IEEE Trans. Software Eng.*, vol. 20, no. 2, pp. 127–141, Feb. 1994.

[25] A. Gabrielian and M. Franklin, "Multilevel Specification of Real-time Systems," *Comm. ACM*, vol. 34, no. 5, pp. 51–60, May 1991.

[26] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming Real-Time Applications with SIGNAL," *Proc. IEEE*, vol. 79, no. 9, Sept. 1991.

[27] C. Ghezzi and R.A. Kemmerer, "ASTRAL: An Assertion Language for Specifying Realtime Systems," *Proc. Third European Software Eng. Conf.*, Milano, Italy, Oct. 1991.

[28] C. Ghezzi and R.A. Kemmerer, "Executing Formal Specifications: The ASTRAL to TRIO Translation Approach," *Proc. Symp. Testing, Analysis, and Verification*, Victoria, B.C., Canada, Oct. 1991.

[29] R. Gerber and I. Lee, "A Layered Approach to Automating the Verification of Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 18, no. 9, pp. 768–784, Sept. 1992.

[30] C. Ghezzi et al., "A General Way to Put Time in Petri Nets," *Proc. Fourth Int'l Workshop Software Design and Specification*, Monterey, Calif., Apr. 1987.

[31] C. Ghezzi, D. Mandrioli, and A. Morzenti, "TRIO: A Logic Language for Executable Specifications of Real-Time Systems," *J. Systems and Software*, June 1990.

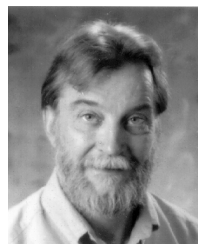
[32] H. Gomaa, "Software Design Methods for the Design of Large-Scale Real-Time Systems," *J. Systems and Software*, vol. 25, no. 2, pp. 127–146, May 1994.

[33] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[34] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, Sept. 1991.
[35] C. Heitmeyer and N. Lynch, "The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems," *Proc. Real-Time Systems*, San Juan, Puerto Rico, Dec. 1994.
[36] D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *Proc. 10th Int'l Conf. Software Eng.*, pp. 396–406, Singapore, Apr. 1988.
[37] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and Verifying Real-Time Systems by Means of the Synchronous Data Flow Language LUSTRE," *IEEE Trans. Software Eng.*, vol. 18, no. 9, Sept. 1992.
[38] D.J. Hatley and I.A. Pirbai, *Strategies for Real-Time System Specification*. Dorset House, 1988.
[39] M.S. Jaffe and N.G. Leveson, "Completeness, Robustness, and Safety in Real-Time Software Requirements Specification," *Proc. 11th Int'l Conf. Software Eng.*, Pittsburgh, Penn., May 1989.
[40] F. Jahanian and A.K. Mok, "Safety Analysis of Timing Properties of Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 12, no. 9, Sept. 1986.
[41] F. Jahanian and A.K. Mok, "Modechart: A Specification Language for Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 20, no. 10, pp. 879–889, Oct. 1994.
[42] R.A. Kemmerer, "Testing Software Specifications to Detect Design Errors," *IEEE Trans. Software Eng.*, vol. 11, no. 1, Jan. 1985.
[43] P.Z. Kolano, "ASTRAL Software Development Environment User's Manual," TRCS96-31, Dept. of Computer Science, Univ. of California, Santa Barbara, July 1996.
[44] R. Koymans, R. Kuiper, and E. Zijlstra, "Specifying Message Passing and Real-Time Systems with Real-Time Temporal Logic," *ESPRIT '87 Achievement and Impact*, North Holland, 1987.
[45] R. Koymans, "Specifying Message Passing and Time-Critical Systems with Temporal Logic," PhD thesis, Eindhoven Univ. of Technology, 1989.
[46] R. Koymans and W.P. de Roever, "Examples of a Realtime Temporal Logic Specification," *Lecture Notes in Computer Science 207*. Berlin: Springer-Verlag, 1985.
[47] L. Lamport, "Concurrent Reading and Writing of Clocks," *ACM Trans. Computer Systems*, vol. 8, no. 4, pp. 305–310, Nov. 1990.
[48] B. Liskov, "Practical Uses of Synchronized Clocks in Distributed Systems," *Distributed Computing*, vol. 6, no. 4, pp. 211–219, 1993.
[49] F. Lagnier, P. Raymond, and C. Dubois, "Formal Verification of Critical Systems Written in Saga/Lustre," *Workshop of Formal Methods, Modeling and Simulation for System Eng.*, St. Quentin en Yvelines (F), Feb. 1995.
[50] A. Morzenti, D. Mandrioli, and C. Ghezzi, "A Model Parametric Real-Time Logic," *ACM Trans. Programming Languages and Systems*, vol. 14, no. 4, pp. 521–573, Oct. 1992.
[51] P.M. Merlin and D.J. Farber, "Recoverability of Communication Protocols, Implications of a Theoretical Study," *IEEE Trans. Comm.*, vol. 24, Sept. 1976.
[52] R. Milner, "Calculi for Synchroni and Asynchroni," *Theoretical Computer Science*, vol. 25, 1983.
[53] A. Morzenti and P. San Pietro, "Object Oriented Logic Specification of Time-Critical Systems," *ACM Transactions on Software Engineering and Methodology*, vol. 3, no. 1, pp. 56–98, Jan. 1994.
[54] J. Ostroff, *Temporal Logic For Realtime Systems*, Taunton, Advanced Software Development Series, 1. Somerset, England: Research Studies Press Ltd., 1989.
[55] A. Pnueli, "The Temporal Semantics of Computer Programs," *Theoretical Computer Science*, vol. 13, 1981.
[56] W.J. Quirk, *Verification and Validation of Real-Time Software*. Berlin: Springer-Verlag, 1985.
[57] I. Suzuki, "Formal Analysis of Alternating Bit Protocol by Temporal Petri Nets," *IEEE Trans. Software Eng.*, vol. 16, no. 11, pp. 1,273–1,281, Nov. 1990.
[58] Y. Wang, "Realtime Behavior of Asynchronous Agents," *Lecture Notes in Computer Science 458*, pp. 502–520. Springer-Verlag, 1990.
[59] J. Zwiers, "Compositionality, Concurrency, and Partial Correctness," *Lecture Notes in Computer Science 321*. Berlin: Springer-Verlag, 1989.

Alberto Coen-Porisini received the Laurea degree in electrical engineering and the PhD degree in computer science from the Politecnico di Milano, Italy, in 1987 and 1992, respectively. He was a visiting scholar at the University of California at Santa Barbara from June 1992 to August 1993. He is currently an assistant professor of computer science at the Politecnico di Milano. His main research interests are in software engineering, with particular reference to formal specification languages and object-oriented systems. Dr. Coen-Porisini is a member of the IEEE Computer Society and the Association for Computer Machinery.

Carlo Ghezzi received his DrEng degree in electrical engineering from the Politecnico di Milano, Italy, where he holds the position of full professor of computer science. Prior to that, he taught at the Universities of Padova, Italy, and at North Carolina at Chapel Hill. He has been a visitor at the University of California, Los Angeles, the University of California, Santa Barbara, ESLAI, Argentina, and TUW, Austria. His research interests are in software specification and design, software processes, human-centered systems, and programming languages, with emphasis on the emerging context of network computing. He has been program chair or co-chair of several international conferences. He is a member of the editorial boards of *IEEE Transactions on Software Engineering*; *Trends in Software, Theory and Practice of Object Systems*; *Software Process Improvement and Practice*; and *Annals of Software Engineering*. Ghezzi is the author of more than 120 technical papers and eight books. He is a member of the IEEE.



Richard A. Kemmerer received the BS degree in mathematics from Pennsylvania State University in 1966, and the MS and PhD degrees in computer science from the University of California, Los Angeles, in 1976 and 1979, respectively. He is a professor and past chair of the Department of Computer Science at the University of California, Santa Barbara. He also leads the Reliable Software Group at the University of California at Santa Barbara. 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. 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 UCLA. His research interests include formal specification and verification of systems, computer system security and reliability, programming and specification language design, and software engineering. He 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*.

Dr. Kemmerer has served as a member of the National Academy of Science's Committee on Computer Security in the Department of Energy (1987-1988); the System Security Study Committee (1989-1991); the Committee for Review of the Oversight Mechanisms for Space Shuttle Flight Software Processes (1992-1993); and the Committee on Maintaining Privacy and Security in Health Care Applications of the National Information Infrastructure (1995-1996). 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 chair of the IEEE Technical Committee on Security and Privacy and a past member of the advisory board for the ACM's Special Interest Group on Security, Audit, and Control. Dr. Kemmerer has served on the editorial boards of *the IEEE Transactions on Software Engineering* and the *ACM Computing Surveys*. He is currently editor-in-chief of the *IEEE Transactions on Software Engineering*. He is a fellow of the IEEE, a fellow of the Association for Computing Machinery, a member of the IFIP Working Group 11.3 on Database Security, and the International Association for Cryptologic Research.