The Composability of ASTRAL Realtime Specifications

Alberto Coen-Porisini[§]
Richard A. Kemmerer[‡]
Reliable Software Group
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

ASTRAL is a formal specification language for realtime systems. It is intended to support formal software development, and therefore has been formally defined. In ASTRAL a realtime system is modeled by a collection of state machine specifications and a single global specification.

This paper focuses on extending the ASTRAL methodology to allow the composition of ASTRAL system specifications into specifications of larger and more complex systems.

The ASTRAL language includes structuring mechanisms that allow one to build modularized specifications of complex systems with layering. In this paper we concentrate on how to combine these complex system specifications into specifications of even more complex realtime systems. This is accomplished by adding a COMPOSE section to the language that provides the needed information to combine two or more ASTRAL specifications into a single new one.

In this paper we also introduce the necessary proof obligations to assure that the assumptions of each of the components of the larger system are satisfied by the other components of the system that replace the previous external environment. We also discuss how some exported transitions become internal transitions of the new system. A telephony example with local central controls that interface to long distance units is used to motivate the extensions.

---

§ Alberto Coen-Porisini is supported by Consiglio Nazionale delle Ricerche - Comitato Nazionale per la Scienza e le Tecnologie dell'Informazione

‡ This research was partially funded by the National Science Foundation under grant CCR-9204249

ACM-ISSTA'93-6/93/Cambridge, MA, USA
© 1993 ACM 0-89791-608-5/93/0006/0128...$1.50

1. Introduction

ASTRAL is a formal specification language for realtime systems. It is intended to support formal software development, and therefore has been formally defined. [GK 91a] discusses the rational of ASTRAL's design and demonstrates how the language builds on previous language experiments. [GK 91b] discusses how ASTRAL's semantics are specified in the TRIO formal realtime logic and outlines how ASTRAL specifications can be formally analyzed by translating them into TRIO. This paper focuses on the composability of ASTRAL specifications.

A compositional specification method allows one to reason about the behavior of a system in terms of the specifications of its components. That is, the behavior of a system comprised of 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. The main benefit of compositional specification is that it often makes it easier to write and reason about designs.

Recently, compositionality has been the focus of much research in concurrent and distributed systems, and, to a lesser extent, in realtime systems. Barringer, Kuiper and Pnueli [BKP 86] were among the first to develop a compositional proof system for concurrent programs. Their work is noteworthy for considering both shared variable and message passing models. However, their goal was only to demonstrate feasibility of a compositional proof system based on temporal

logic, so their published results are very preliminary.

One of the most thorough treatments of compositionality in concurrent systems appears in [Zwi 89]. The work of Hooman [Hoo 87], and of Hooman and Widom [HW 88] both focus on developing a compositional proof system for realtime systems.

The ASTRAL language as it has been defined and used in the previous papers lends itself to a top-down design strategy. That is, a system is modeled by a collection of state machine specifications (one per process type) and a single global specification that contains the global invariants for the system. Each of the state machine specifications consists of a sequence of levels. The first (or top) level is a very abstract model of the process. Lower levels are increasingly more detailed with the lowest level corresponding closely to high level code. A proof theory that prescribes how the proofs of the individual state machine specifications can be combined to produce a proof of the entire system is presented in [CKM 92].

An issue that arises is how to compose two or more ASTRAL system specifications (i.e., a global specification and its associated collection of state machine specifications) to derive the specification for a more complex realtime system. In this paper we concentrate on this composition. This is accomplished by adding a COMPOSE section to the language, which provides the needed information to combine two or more ASTRAL specifications into a single specification of the total system.

In this paper we also introduce the necessary proof obligations to assure that the assumptions of each of the components of the larger system are satisfied by the other components of the system that replace what was previously the external environment. We also discuss how some exported transitions become internal transitions of the new system.

In the next section a brief overview of ASTRAL is presented. In section 3 an example system, which is used for illustrating the composability features of ASTRAL is introduced. Section 4 presents the components of the COMPOSE clause, demonstrates how a single system specification can be constructed from the component system specifications using the information in the compose clause, and presents the additional proof obligations that are needed to show that the composition is sound. Finally, in section 6 some conclusions from this research are presented and possible future directions are proposed.

2. Overview of ASTRAL

ASTRAL uses a state machine process model and has types, variables, constants, transitions, and invariants. A realtime system is modeled by a collection of state machine specifications and a single global specification. Each state machine specification represents a process type of which there may be multiple instances in the realtime system. State variables and transitions may be explicitly exported by a process. This makes the variable values readable by other processes and the transitions executable by the external environment; exported transitions cannot be executed by another process. Interprocess communication is via the exported variables, and is accomplished by inquiring about the value of an exported variable for a particular instance of the process. A process can inquire about the value of any exported variable of a process type or about the start or end time of an exported transition. Start(Op_i, t) is a predicate that is true if and only if transition Op_i starts at time t and there is no other time after t and before the current time when Op_i starts (i.e., t is the time of the last occurrence of Op_i). For simplicity, the functional notation Start(Op_i) is adopted as a shorthand for "time t such that Start(Op_i, t)", whenever the quantification of the variable t (whether existential or universal) is clear from the context. Start-k(Op_i) is used to give the start time of the kth previous occurrence of Op_i. Inquiries about the end time of a transition Op_i may be specified similarly using End(Op_i) and End-k(Op_i).

In ASTRAL one can either refer to the current time, which is denoted by Now, or to an absolute

value for time that must be less than Now. That is, in ASTRAL one cannot express values of time that are to occur in the future. To specify the value that an exported variable var had at time t, ASTRAL provides a past(var,t) function.

The 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". For inquiries where there is more than one instance of that type, the inquiry is preceded by the unique id of the desired instance, followed by a period, and process instance ids that are used in a process specification must be explicitly imported. For example, i.Start(Op) gives the last start time that transition Op was executed by the process instance whose unique id is i. However, when the process instance performing the inquiry is the same as the instance being queried, the preceding id and period may be dropped.

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

ASTRAL also allows assumptions about the external environment and the other processes in the system to be specified in an environment clause and an imported variable clause, respectively. The optional environment clause describes the pattern of invocation of external transitions. If Op_i is an exported transition, Call(Op_i) may be used in the environment clause to denote the time of the last occurrence of the call to Op_i (with the same syntactic conventions as Start(Op_i)). The imported variable clause describes patterns of value changes to imported variables, including timing information about any transitions exported by other processes that may be used by the process being specified (e.g., Start(Op_i) and End(Op_i)).

Critical requirements for the system being designed are represented as invariants and schedules in an ASTRAL specification. Invariants represent the properties holding in every state the system can be in, regardless of the environment, while schedules represent stronger properties holding in every state the system can be in, provided that the environment produces stimuli as prescribed in the environment clauses. Invariants and schedules can be both local and global.

A detailed description of ASTRAL and of its underlying motivations is provided in [GK 91a]. In this paper, due to space limitations, only the concepts of ASTRAL that are needed to present the composability issues are discussed in detail. These concepts are illustrated via an extension of the phone example presented in [GK 91a] that includes long distance dialing and area codes.

3. Example System

The example system used in this paper is a simple phone network. The system is composed of two different ASTRAL specifications, the first one being the specification of a local area phone system, while the second one is the specification of a long distance system. The composition of the two specifications will lead to a phone network in which there are several area codes, and in each area code there is a local phone system composed of many phones and one central control.

The system is a simplification of a real phone network, every local phone number is seven digits long, area codes are three digits long, a customer can be connected at most with one other phone (either local or in another area) and ongoing calls cannot be interrupted.

3.1 The local phone system

The local phone system specified is derived from a previous ASTRAL specification of a phone system presented in [GK 91a]. This system is composed of a set of process instances of type Phone and a set of Central_Control units that provide all the functionalities needed to set up a local call, i.e., a call between two phones connected to the same Central_Control unit. This is modeled in ASTRAL by the following process declaration:

```
PROCESSES
    Phone: array[0 .. Num_Phone] of Phone,
    Centrals: array[0 .. Num_Area] of Central_Control
```

The constant

```
Is_In_Area(P:Phone, C:Central_Control): Boolean
```

is used to describe that each phone is associated with a central control, and that such a binding cannot change.

The Central_Control unit, also provides the functionality that allows it to send out (receive) data to (from) an external environment whenever a call to a different area code is made. Figure 1 shows the architecture of a local phone system.

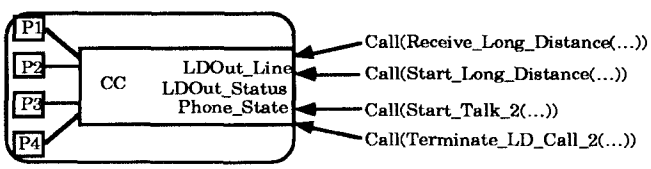

Figure 1: A Local Phone System

In what follows we will focus on the behavior of the central control with respect to long distance calls.

The global specification contains several type declarations among which there are:

```
Enabled_State = (Idle, Ready_To_Dial, Dialing, Ringing,
                 Waiting, Calling, Disconnect, Busy, Alarm),
```

representing the states that a customer's phone could be in,

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

representing the data structure needed to store the information related to a long distance call, and

```
Connection_Status = (Available, In_Progress,
                     Disconnected, Connected, Talk)
```

representing the status of a long distance line .

The Central_Control exports three variables (LDOut_Line, LDOut_Status and Phone_State) to send out data to the external environment and four transitions (Receive_Long_Distance, Start_Talk_2, Start_Long_Distance and Terminate_LD_Call_2) to receive data from the environment.

The variable:

```
Phone_State(P:Phone): Enabled_State
```

indicates the central control's view of the mode of each of its customer's phones.

The variables:

```
LDOut_Line(P: Phone): Connection
```

and

```
LDOut_Status(P:Phone): Connection_Status
```

indicate during a long distance call, whom the phone P is connected to and what is the status of the connection, respectively.

The exported transitions of the central control are called from the environment whenever an incoming call to a phone connected to the central control occurs (Receive_Long_Distance), to notify the central control that an outgoing call has been received by the called central control (Start_Long_Distance), to notify the central control that an outgoing call has been answered by the called customer (Start_Talk_2) and to notify the central control that an outgoing call has been ended by the called central control (Terminate_LD_Call_2). For instance the specification of transition Receive_Long_Distance is as follows:

```
TRANSITION Receive_Long_Distance (
    LDIn_Line:Connection,
    LDIn_Status:Connection_Status) Tim5
 ENTRY
      LDIn_Line.To_Area = Get_Area(Self)
    & LDIn_Status = In_Progress
    & Phone_State(Get_Phone_ID(
                      LDIn_Line.To_Number)) = Idle
    & Available_lines > 0
 EXIT
    Phone_State(Get_Phone_ID(
        LDIn_Line'.To_Number)) BECOMES Ringing
    & LDOut_Status(Get_Phone_ID(
        LDIn_Line'.To_Number)) BECOMES Connected
    & Plug(LDOut_Line(Get_Phone_ID(
            LDIn_Line'.To_Number)), LDIn_Line')
    & FORALL P: My_Phone
      (   P ~= Get_Phone_ID(LDIn_Line'.To_Number)
      → NOCHANGE(LDOut_Line(P)))
```

Figure 2 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.

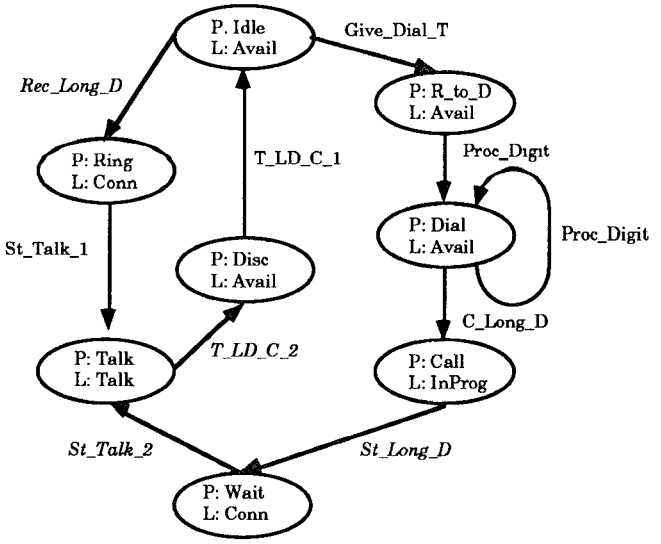

Figure 2: The Central_Control

3.2 The 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 instance of type Long_Distance_Unit. For simplicity, we assume that each Long_Distance_Unit instance is connected with all the other instances, so that a direct communication between two Long_Distance_Unit instances is always possible (See figure 3).

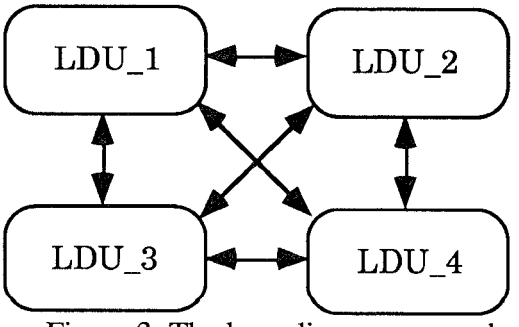

Figure 3: The long distance network

Long_Distance_Unit has four variables:

NetworkOut(L:Line): Connection

and

NetworkStatus(L:Line): Connection_Status

are used to communicate with another Long_Distance_Unit, while

LocalOut(L:Line): Connection

and

LocalStatus(L:Line): Connection_Status

are used to send the information about the connection and the status occurring on a given line L to the external environment.

Long_Distance_Unit also exports four transitions:

- Receive_Local_Request: which is called whenever a long distance call has been requested from the local area,
- Local_Connection_Established, which is called to notify the unit that an incoming call to the local area has been received,
- Started_Local_Talk, which is called to notify the unit that an incoming call, previously received has been answered, and
- End_Local_Connection, which is called to notify the unit that an incoming call has ended.

Figure 4 shows the interface of a Long_Distance_Unit with the external environment.

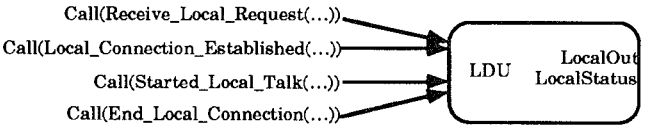

Figure 4: The long distance unit

For instance the specification of transition Receive_Local_Request is as follows:

```
TRANSITION Receive_Local_Request(
    In_Line: Connection,
    In_Status: Connection_Status) Ti1
 ENTRY
      In_Line.From_Area = Get_Area(Self)
    & In_Status = In_Progress
 EXIT
    EXISTS L: Line
      (  NetworkStatus'(L) = Available
      &  Connect(NetworkOut(L), In_Line')
      &  FORALL L1:Line
         (L1 ~= L → NOCHANGE(NetworkOut(L1))
      &  NetworkStatus(L) BECOMES In_Progress)
```

Figure 5 shows how the variables of Long_Distance_Unit are affected by its transitions

when processing incoming or outgoing long distance calls.

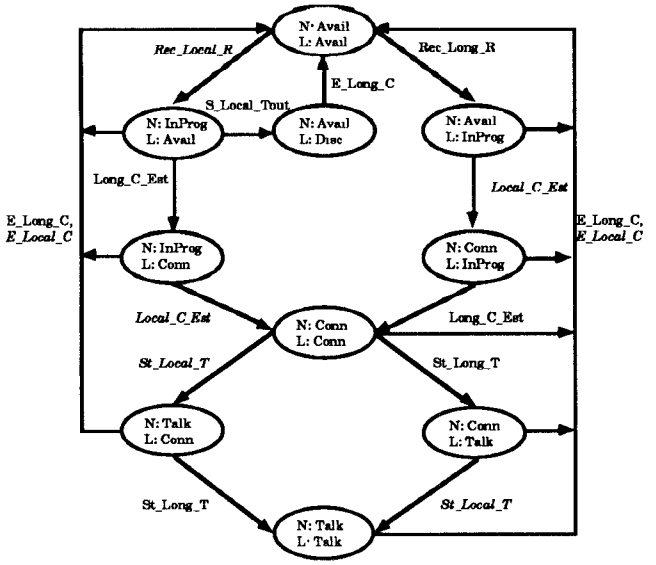

Figure 5: The Long_Distance_Unit

The complete details of the specifications can be found in [CK 92].

4. Composing ASTRAL Specifications

Consider two ASTRAL top level specifications S' and S". Composing S' and S" means to build a new top level specification C, that is the specification of a system obtained by making S' and S" interact. The behavior, the environment and the properties of C are obtained from those of S' and S", once their interaction is formally described. Thus, in order to compose S' and S" we have to define:

1) how the interconnection between two or more ASTRAL specifications can be formally described,

2) how the specification C can be built starting from S', S" and the description of their interaction.

3) under which conditions the properties stated in S' and S" will still be valid in C.

4.1 The COMPOSE clause

A COMPOSE clause describes the interaction between S' and S", and thus provides the information needed to compose two or more global specifications into a single specification of the combined system.

The interaction between S' and S" is described by specifying which exported transitions of S' (S") are no longer exported to the external environment, i.e., the stimuli needed to fire such transitions are produced by the system S" (S') rather than by the external environment.

For instance, in Figure 6 S' exports transitions T1 and T2 and state variables x1, x2 and x3, while S" exports transition T3 and state variables y1 and y2.

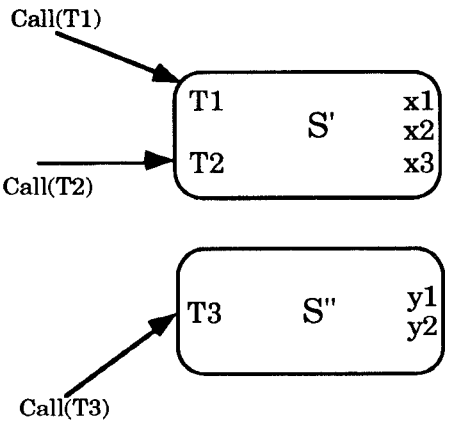

Figure 6: S' and S"

When S' and S" are composed some transitions of S' (S") will not need an external call, since S" (S') is now providing part of the environment in which S' (S") works. For instance, in Figure 7 transitions T1 and T3 are no longer exported since the events that trigger them are now represented by particular values of y2 and x1, x3, respectively.

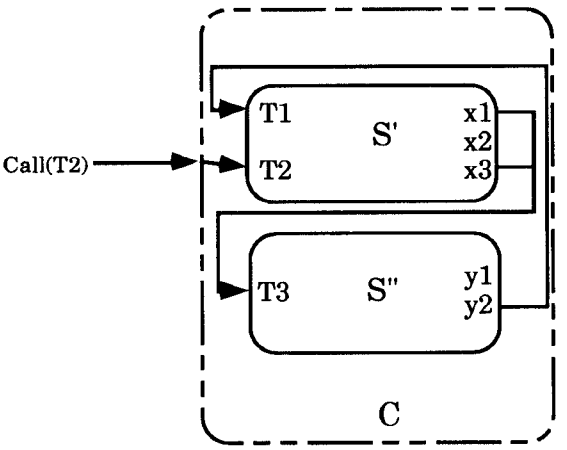

Figure 7: Composing S' and S"

Thus, the system C will export only transition T2, i.e., the external environment of C can call only transition T2 (See Figure 8).

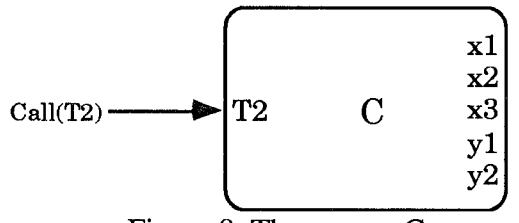

Figure 8: The system C

In general a compose clause contains the following parts:

- A set of clauses defining types, constants, and definitions that are used in the compose clause. For instance, the following declarations are introduced in the compose clause of the example:

```
CONSTANTS
    Is_LD_Unit(Central_Control,
               Long_Distance_Unit): Boolean

DEFINES
    Change(L: Connection_Status, t: Time) ==
      EXISTS e: Time
      (   e > 0 & e ≤ t
      &   FORALL d: Time
            (d ≥ t - e & d < t → past(L,d) ~= past(L,t))),
    LastChg(L: Connection_Status, t: Time) ==
          Change(L,t)
      &   FORALL t1: Time
            (t1 > t & t1 ≤ Now → ~Change(L,t1))
```

- 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' and S" with different meanings).
- A call generation clause which describes how exported transitions of S' (S") are triggered by events occurring in S" (S'). Such events are described by means of formulas of the following form:

```
FORALL t: Time ,... (P(S') ↔ Call(T) = t),
```

where P(S') is an ASTRAL well-formed formula describing the occurrence of the events in S', which are equivalent to calling the exported transition T of S".

For instance, the following formula describes when the behavior of process Central_Control is such that a call to transition Receive_Local_Request of process Long_Distance_Unit occurs:

```
FORALL  t: Time, C: Central_Control, P: Phone,
          U: Long_Distance_Unit
    ( LastChg(C.LDOut_Status(P),t)
    & C.LDOut_Status(P) = In_Progress
    & Is_In_Area(P,C) & Is_LD_Unit(C,U)
    ↔ Call(U.Receive_Local_Request(C.LDOut_Line(P),
                                   C.LDOut_Status(P))) = t)
```

The generation of calls for transition Receive_Long_Distance of process Central_Control is specified in the following way:

```
FORALL t: Time, C: Central_Control, L: Line,
         U: Long_Distance_Unit
    (  LastChg(U.LocalStatus(L),t)
    & U.LocalStatus(L) = In_Progress
    & Is_LD_Unit(C,U)
    ↔ Call(C.Receive_Long_Distance(U.LocalOut(L),
                                   U.LocalStatus(L))) = t)
```

4.2 Building the new 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, it is necessary to:

1) build the new Global specification, and
2) build the new Process specifications

according to the guidelines described in the compose clause.

4.2.1 The global specification

A global specification G is made up of a set of clauses defining types, constants and definitions, as well as global invariant I_G, global schedule Sc_G and global environment Env_G clauses.

The clauses defining types, constants and definitions are built by taking the corresponding clauses from the compose clause and the corresponding clauses belonging to specifications

S' and S", using the name clash clause to resolve any name clashes.

The global invariant I is constructed in the following way:

Let I' and I" denote the global invariants of S' and S", respectively:

1) Rewrite I' and I" using the name clash clause. Let RI' and RI" be the result of these rewritings;
2) Substitute any occurrences of the operators Start and End referring to a no longer exported transition with an equivalent predicate referring to exported variables. Let SRI' and SRI" be the result of these substitutions.
3) I is the conjunction of SRI' & SRI"

The global schedule Sc is constructed in the same manner as the global invariant. However, at point 2, any occurrence of the operator Call referring to a no longer exported transition has to be substituted by using the predicate in the corresponding call generation clause.

The global environment clause has to be modified since some transitions are no longer exported. In particular all formulas referring to the transitions that are used in the call generation clause of the compose clause should be eliminated from the global environment clause.

4.2.2 The process type specification

A process type specification P is composed of a set of transitions $Op_1, \ldots, Op_n$, a local invariant I, a local schedule Sc a local environment Env, imported variable assumptions IV, a further local environment FEnv and a further process assumption FPA. Moreover, every transition Op_j is described by entry and exit clauses denoted EN_j and EX_j, respectively.

Each process type specification in either S' or S" should be included in C; however, the following transformations have to be made:

- The local environment clauses (Env, and FEnv) have to be modified since some transitions are no longer exported. This transformation is identical to the one described for the global environment clause;
- The export/import clauses have to be modified: the transitions belonging to the processes of S' (S") that are referred to in the call generation clause of the compose section, should no longer be exported by S' (S"); moreover each state variable in S' (S") referred to in the call generation clause of a transition belonging to the processes in S" (S') has to be imported by the processes of S" (S'). For instance, the export clause of the process type Central_Control before the composition is:

EXPORT
Phone_State, Enabled_Ring_Pulse, LDOut_Line, Enabled_Ringback_Pulse, LDOut_Status, Receive_Long_Distance, Start_Long_Distance, Start_Talk_2, Terminate_LD_Call_2

while, after the composition is:

EXPORT
Phone_State, Enabled_Ring_Pulse, LDOut_Line, Enabled_Ringback_Pulse, LDOut_Status

- Each transition T belonging to the processes of S' (S") referred to in the call generation clause of the compose section, has to be modified using the related call generation clause. This will result in adding to the entry clause of such transitions a formula such as:

EXISTS t: Time ...(P(S') & start(T) < t)

where P(S') is the predicate used in the related call generation clause.

Note that a similar clause may also be needed in the exit assertion.

For instance, the transition Receive_Local_Request of process Long_Distance_Unit is transformed as follows in the composed specification (the italicized portions indicate changes or additions):

```
TRANSITION Receive_Local_Request Ti1
    ENTRY
        EXISTS t: Time, C: Central_Control, P: Phone
        ( LastChg(C.LDOut_Status(P),t)
        & C.LDOut_Status(P) = In_Progress
        & Is_In_Area(P,C) & Is_LD_Unit(C,Self)
        & Start(Receive_Local_Request) < t)
    EXIT
        EXISTS t: Time, C: Central_Control, P: Phone
        ( LastChg'(C.LDOut_Status(P),t)
        & C.LDOut_Status'(P) = In_Progress
        & Is_In_Area(P,C) & Is_LD_Unit(C,Self)
        & Start-2(Receive_Local_Request) < t
        &  EXISTS L: Line
          ( NetworkStatus'(L) = Available
          &  Connect(NetworkOut(L), C.LDOut_Line(P))
          &  FORALL L1:Line (L1 ~= L
                IMPLIES NOCHANGE(NetworkOut(L1))
          &  NetworkStatus(L) BECOMES In_Progress))
```

- The imported variables clause (IV) has to be modified in order to describe the behavior of the newly imported variables. The new assumptions are generated from the related call generation clauses and the old environment clauses;
- The local schedule (Sc) has to be modified only if it refers to the call of no longer exported transitions. In such a case the same substitution defined for the global schedule is applied;
- The local invariant (I) and the further process assumptions (FPA) are not modified.

The complete specification of the composed system can be found in [CK 92].

4.3 Proof obligations for system composition

Again consider two ASTRAL system specifications S' and S". Both S' and S" may have a local and/or global schedule and/or invariant, representing some of the properties of S' and S". In [CKM 92] it was shown how such properties can be formally proved.

When S' and S" are composed, the "environment" in which S' (S") runs is given by the external environment and the exported features of S" (S'). As a consequence, the properties of S' (S") might not be valid in the composed system because of the changes in the environment in which S' (S") runs. However, if we prove that from the viewpoint of S' (S") the behavior of the environment (i.e., the way in which it produces stimuli) has not changed then we can conclude that the properties of S' (S") are still valid.

For simplicity assume that the specification resulting from the composition of S' and S" is a closed system, that is, there are no exported transitions[1]. Note that weakening this hypothesis requires one to partition each environmental assumption into two parts; the first part being the assumptions that do not involve the environment of the composed specification, and the second part being the assumptions that continue to involve the environment. In general the proof obligations related to composability affect only the first part.

4.3.1 Invariants

The invariants of S' (S") represent properties that hold for every environment in which S' (S") may run. The main effect of composing S' and S" is the modification of the environment in which they run, and therefore all the local invariants and the global invariant belonging to either S' or S" are still valid in the composed system.

4.3.2 Schedules

The schedules of S' (S") represent properties that hold when the environment in which S' (S") runs behaves according to the assumptions stated in the global environment (Env_G), local environment (Env) and further environment (FEnv) clauses.

Since in the composed system S' (S") provides the environment for S" (S'), the schedules of S" (S') will continue to hold only if the behavior of S' (S") implies what is stated in the environment clauses of S" (S'). Thus, the following proof obligations ensure that the global schedule of S' (S") is still valid .

[1]This is not the case for the example discussed in the paper

$$A1 \;\&\; A2'' \;\&\; A3 \;\&\; A4 \;\&\; Env_G' \;\&\; CG' \vdash F_\sigma' \rightarrow Env_G'', \text{ for } S''$$

$$A1 \;\&\; A2'' \;\&\; A3 \;\&\; A4 \;\&\; Env_G'' \;\&\; CG'' \vdash F_\sigma'' \rightarrow Env_G', \text{ for } S'$$

A1, A2", A3 and A4 are the axioms describing the ASTRAL abstract machine as in the global schedule proof obligation [CKM 92]; Env_G' and Env_G'' are the global environment clauses of S' and S", respectively. F_σ' and F_σ'' are sequences of events of S' and S", respectively. CG' (CG") is the call generated clause that binds events occurring in S' (S") to the generation of a call for the system S" (S').

Similarly, the following proof obligations ensure that the local schedule of process p of S' (S") is still valid.

$$A1 \;\&\; A2'' \;\&\; A3 \;\&\; A4 \;\&\; Env_G' \;\&\; \tilde{CG}' \vdash F_{\sigma p}' \rightarrow Env_p'' \;\&\; FEnv_p'', \text{ for } S''$$

$$A1 \;\&\; A2'' \;\&\; A3 \;\&\; A4 \;\&\; Env_G'' \;\&\; \tilde{CG}'' \vdash F_{\sigma p}'' \rightarrow Env_p' \;\&\; FEnv_p', \text{ for } S'$$

Env_p' and $FEnv_p'$ (Env_p'' and $FEnv_p''$) are the local assumptions about the environment and the further assumptions made by process p of S' (S"), and $\tilde{CG}'$ ($\tilde{CG}''$) is obtained from CG' (CG") by freezing the value of process p.

For instance, when proving that the local schedule of process Central_Control is still valid, one of the clauses of CG" is:

```
FORALL  t: Time, C: Central_Control, P: Phone,
          U: Long_Distance_Unit
    ( LastChg(C.LDOut_Status(P),t)
    & C.LDOut_Status(P) = In_Progress
    & Is_In_Area(P,C) & Is_LD_Unit(C,U)
    ↔ Call(U.Receive_Local_Request(C.LDOut_Line(P),
                                   C.LDOut_Status(P))) = t)
```

Since we are considering a single instance of process type Central_Control, the variable C should be considered as a constant. As a consequence, this clause in $\tilde{CG}''$ is transformed into:

```
FORALL t: Time, P: Phone, U: Long_Distance_Unit
    ( LastChg(C.LDOut_Status(P),t)
    & C.LDOut_Status(P) = In_Progress
    & Is_In_Area(P,C) & Is_LD_Unit(C,U)
    ↔ Call(U.Receive_Local_Request(C.LDOut_Line(P),
                                   C.LDOut_Status(P))) = t)
```

5. Conclusions and Future Directions

In this paper we have described 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 ASTRAL specification language. We also described how the compose clause can be used to transform the existing system specification into a new ASTRAL specification for the composite system. Finally, we introduced the additional proof obligations that are necessary to assure that the resulting specification is sound.

By adding the compose clause to the ASTRAL language and introducing a compositional specification method a system designer can now reason about the behavior of a composite system in terms of the specifications of its components. The size of the composite specification grows linearly with the size of the component specifications. However, because the composite specification is completely determined by the component specifications, the resulting system specification is easy to comprehend.

The composite specification approach coupled with the previously defined top-down specification approach of ASTRAL allows a system designer to specify his/her system using either a bottom-up or a top-down approach, or some combination of the two. The composability of specifications also promotes the reuse of existing specifications.

Future work in this area will concentrate on applying the composition approach to more varied and complex realtime systems. The ASTRAL specification processor will also be updated to process the compose clause and generate the additional proof obligations.

References

[BKP 86] Barringer H., Kuiper R. and Pnueli A., "Now You May Compose Temporal Logic Specifications," *Proceedings of 18th POPL*, pp. 173-183. ACM, 1986.

[CK 92] Coen-Porisini A. and Kemmerer R. "The Composability of ASTRAL Realtime Specifications", Technical Report TRCS 92-25, Department of Computer Science, University of California Santa Barbara, December 1992.

[CKM 92] Coen-Porisini A., Kemmerer R. and Mandrioli D., "A Formal Framework for ASTRAL Intra-Level Proof Obligations", *Proceedings of the Fourth European Software Engineering Conference*, Garmisch, Germany, September 1993.

[GF 91] Gabrielian A. and Franklin M., "Multilevel Specification of Realtime Systems," *CACM 34*, 5, pp. 51-60, May 1991.

[GK 91a] Ghezzi C. and Kemmerer R., "ASTRAL: An Assertion Language for Specifying Realtime Systems," *Proceedings of the Third European Software Engineering Conference*, Milano, Italy, pp. 122-146, October 1991.

[GK 91b] Ghezzi C. and Kemmerer R., "Executing Formal Specifications: the ASTRAL to TRIO Translation Approach, "Proceedings *of TAV4: the Symposium on Testing, Analysis, and Verification*, Victoria, B.C., Canada, pp. 112-119, October 1991.

[Hoo 87] Hooman J., "A Compositional Proof System for an Occam-like Real-Time Language," Technical Report CSN 87/14, Department of Mathematics and Computer Science, Eindhoven University of Technology, November 1987.

[HW 88] Hooman J. and Widom J., "A Temporal-Logic Based Compositional Proof System for Real-Time Message Passing," Technical Report 88-919, Department of Computer Science, Cornell University, June 1988.

[Ost 88] Ostroff J.S., "Modular Reasoning in the ESM/RTTL Framework for Real-Time Systems," Technical Report CS-88-03, Department of Computer Science, York University, April 1988.

[Zwi 89] Zwiers J., Compositionality, Concurrency, and Partial Correctness, LNCS 321, Springer Verlag, Berlin, 1989.