A Formal Framework for ASTRAL Inter-level Proof Obligations

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

Dino Mandrioli
Dipartimento di Elettronica e Informazione
Politecnico di Milano
20133 Milano, Italia

Abstract

ASTRAL is a formal specification language for realtime systems. It is intended to support formal software development, and therefore has been formally defined. This paper focuses on how to formally prove the mathematical correctness of ASTRAL specifications.

ASTRAL is provided with structuring mechanisms that allow one to build modularized specifications of complex systems with layering. A realtime system is modeled by a collection of process specifications and a single global specification. Each process specification consists of a sequence of levels; each level is an abstract data type view of the process being specified. In this paper further details of the ASTRAL refinement process, which were not fully developed in previous papers, are presented.

Formal proofs in ASTRAL can be divided into two categories: *inter-level* proofs and *intra-level* proofs. The former deal with proving that the specification of level i+1 is consistent with the specification of level i, while the latter deal with proving that the specification of level i is consistent and satisfies the stated critical requirements. This paper concentrates on inter-level proofs. The necessary proof obligations to assure that a refinement is a correct implementation are presented. The approach is motivated with a communication example.

Key Words: Formal methods, Formal specification and verification, Realtime systems, Timing requirements, Proof obligations, State machines, ASTRAL.

A Formal Framework for ASTRAL Inter-level Proof Obligations

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

Dino Mandrioli
Dipartimento di Elettronica e Informazione
Politecnico di Milano
20133 Milano, Italia

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, and [CKM 92] gives further details of the ASTRAL environment and critical requirement components. [GK 91b] discusses how ASTRAL's semantics are specified in the TRIO formal realtime logic, and it also outlines how ASTRAL specifications can be formally analyzed by translating them into TRIO and then using the TRIO validation theory.

Recently, a number of approaches have been proposed to build formal proofs for realtime systems [Ost 89, FMM 91, ACD 90, Suz 90, CHS 90, GF 91]. Many of these exploit the so called "dual language approach" [Ost 89, Pnu 77] where a system is modeled as an abstract machine (e.g., a finite state machine or a Petri net) and its properties are described through some assertion language (e.g., a logic or an algebraic language). However, they are based on low level formalisms, i.e., abstract machines and/or assertion languages that are not provided with modularization and abstraction mechanisms. As a consequence, the proofs lack structure, which makes them unsuitable for dealing with complex real-life systems.

On the contrary, ASTRAL is provided with structuring mechanisms that allow one to build modularized specifications of complex systems with layering [GK 91a, GK 91b]. In this paper further details of the ASTRAL refinement process, which were not fully developed in previous papers, are presented.

Formal proofs in ASTRAL can be divided into two categories: *inter-level* proofs and *intra-level* proofs. The former deal with proving that the specification of level i+1 is consistent with the specification of level i, while the latter deal with proving that the specification of level i is consistent and satisfies the stated critical requirements. The intra-level proofs are discussed in detail in [CKM 92]. This paper concentrates on inter-level proofs.

In the next section a brief overview of ASTRAL is presented. Section 3 discusses layering and the implementation statement. Section 4 presents the proof obligations

that are necessary to prove that a lower level implementation is a correct refinement of the upper level; that is, if the critical requirements were satisfied in the upper level specification, then they will be satisfied by the lower level specification. Section 5 presents an example of a two level ASTRAL specification along with a proof of the inter-level proof obligations for the specification. 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, t) is a predicate that is true if and only if transition Op starts at time t and there is no other time after t and before the current time when Op starts (i.e., t is the time of the last occurrence of Op). For simplicity, the functional notation Start(Op) is adopted as a shorthand for "time t such that Start(Op, t)", whenever the quantification of the variable t (whether existential or universal) is clear from the context. Start-k(Op) is used to give the start time of the kth previous occurrence of Op. Inquiries about the end time of a transition Op may be specified similarly using End(Op) and End-k(Op).

The ASTRAL computation model views the values of all variables being modified by a transition as being changed by the transition in a single atomic action that occurs when the transition completes execution. Thus, if a process is inquiring about the value of an exported variable while a transition is being executed by the process being queried, the value obtained is the value the variable had when the transition commenced.

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; 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 nonprimitive 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 is an exported transition, Call(Op) may be used in the environment clause to denote the time of the last occurrence of the call to Op (with the same syntactic conventions as Start(Op)). 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 (eg., Start(Op) and End(Op)).

Critical requirements for the system being specified are represented as invariants and schedules in an ASTRAL specification. Invariants represent the properties that are required to hold in every state the system can be in, regardless of the environment, while schedules represent stronger properties that are required to hold in every state the system can be in, provided that the environment produces stimuli according to the description contained in the environment clauses. Invariants and schedules can be both local and global.

The computation model for ASTRAL is based on nondeterministic state machines and assumes maximal parallelism, noninterruptable and nonoverlapping transitions in a single process instance, and implicit one-to-many (multicast) message passing communication, which is instantaneous.

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 layering issues are discussed in detail. These concepts are illustrated via a refinement of the communication example presented in [GK 91b].

3. Layering in ASTRAL

As mentioned earlier, in ASTRAL a realtime system is modelled by a collection of process type specifications and a single global specification. Each process specification consists of a sequence of levels, where each level is an abstract data type view of the process being specified. The first (Top level) view is a very abstract model of what constitues a process (types, constants, variables), what the system does (state transitions), what the environment in which the process works behaves like (Environment clause and Imported variable clause), and what the critical requirements are (Invariants and Schedules). Lower levels are increasingly more detailed; the lowest level may correspond closely to high level code. The relationship between each successively more detailed level and the abstract level it refines is specified in implementation statements.

Given two ASTRAL process specification P_U and P_L, where P_L is a refinement of P_U, the implementation statement IMPL defines a mapping from all the types, constants, variables and transitions of P_U into their corresponding terms in P_L. A type, constant, variable or transition of P_L representing the implementation of a

corresponding term in P_U is referred to as a mapped type, constant, variable or transition. P_L can also introduce types, constants and/or variables, that are not mapped; these are referred to as the new types, constants or variables of P_L. Note that P_L cannot introduce any new transition; i.e., each transition of P_L must be a mapped transition.

Each transition of P_L is defined by an entry assertion, an exit assertion and a nonnull duration in the same manner as top level ASTRAL transitions. All scope rules that apply to top level ASTRAL transitions apply to lower level transitions as well.

There are two basic ways to implement a transition of P_U:

- Selection, or
- Sequence

That is, a transition of P_U can be mapped into a sequence of transitions, a selection of transitions, or any combinations thereof. In the following sections each of the basic implementation mechanisms is discussed.

3.1 Selection Mapping

A transition T_U of P_U can be refined by choosing from a number of different transitions of P_L, $T_{L.1}$, …, $T_{L.n}$. That is, every time T_U fires one and only one $T_{L.j}$ fires. This is specified in the following way:

$$TU == A1 \ \& \ TL.1 \mid A2 \ \& \ TL.2 \mid \ldots \mid An \ \& \ TL.n,$$

where A_j, $1 \leq j \leq n$, are boolean expressions involving P_L's terms and parameters of T_U, which are essentially Dijkstra's "guards". The A_j need not be disjoint, but when they are not the specification is nondeterministic.

The semantics of the selection statement is such that transition $T_{L.j}$ is enabled whenever the logical conjunction of its entry condition and its associated guard A_j evaluates to *true*.

3.2 Sequence Mapping

A transition T_U of P_U can also be implemented as a sequence of transitions of P_L that are executed in a pre-determined order. This is specified by means of the WHEN-DO-BEFORE statement:

$$TU == \text{WHEN } Entry_L \text{ DO } T_{L.1} \text{ BEFORE } T_{L.2} \text{ BEFORE } \ldots \text{ BEFORE } T_{L.n} \text{ OD}$$

where T_U belongs to P_U, $T_{L.j}$, $1 \leq j \leq n$, belongs to P_L and $Entry_L$ is a well-formed formula involving P_L's terms and the parameters of T_U.

The semantics of the WHEN-DO-BEFORE statement is such that the sequence of transitions $T_{L.1}$; $T_{L.2}$; …; $T_{L.n}$ is enabled (i.e., can start) whenever $Entry_L$ evaluates to *true*. Once the sequence has started it cannot be interrupted until all of its transitions have been executed. Moreover, the WHEN-DO-BEFORE statement requires transitions to be executed following a left to right order, i.e., the first

transition to be executed is $T_{L.1}$ while the last one is $T_{L.n}$. Thus, transition $T_{L.j}$ fires as soon as its entry condition evaluates to *true*, provided that transition $T_{L.j-1}$ has ended.

The starting time of transition T_U of P_U corresponds to the starting time of the sequence
$T_{L.1}$; $T_{L.2}$; …; $T_{L.n}$ (which is not necessarily equal to the starting time of $T_{L.1}$ because of a possible delay between the time when the sequence starts and the time when $T_{L.1}$ becomes enabled), while the ending time of the transition T_U of P_U corresponds to the ending time of the last transition in the sequence ($T_{L.n}$).

The exit clause of any $T_{L.j}$ cannot modify the value of a mapped variable of P_L unless $T_{L.j}$ always appears as the last transition of a sequence. This further constraint is a consequence of the ASTRAL communication model. That is, in the upper level, the new values of the variables affected by T_U are broadcast when T_U terminates. Thus, mapped variables of P_L can be modified only when the sequence implementing T_U ends (i.e., when the last transition of the sequence terminates).

4. Inter-level Proof Obligations in ASTRAL

As mentioned earlier, in ASTRAL the system critical requirements are expressed by means of invariants and/or schedules. Invariants represent the properties that must hold in every state the system can be in, regardless of the environment. Schedules are additional system properties that are required to hold under more restrictive hypotheses than invariants. Unlike invariants, the validity of a schedule may be proved using the assumptions about the external environment (environment clauses) and/or the behavior of the other processes composing the system (imported variable clauses).

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

Thus, a correct ASTRAL top level specification is such that:

1) the invariant is satisfied, and
2) the schedule is satisfied or can be satisfied by making some further assumptions about the environment and/or the behavior of the process being designed; in the latter case the schedule is said to be satisfiable.

A correct implementation of a process specification must have the same properties as the top level specification. That is, the invariant has to be satisfied by the implementation and the schedule has to be either satisfied or satisfiable by the

implementation depending on whether it was satisfied or satisfiable in the top level specification.

Thus, given an upper level process specification P_U, a lower level process specification P_L and an implementation statement IMPL defining the mapping between P_U and P_L, the inter-level proof obligations guarantee that P_L is a correct implementation of P_U. That is, P_L satisfies the invariant if P_U does, and the schedule is satisfied (satisfiable) by P_L if it is by P_U.

The proof obligations involve the implementation of each transition belonging to the upper level. Note that no new transition (i.e., a transition not belonging to any mapping) is allowed in a lower level process specification. If a new transition that was not mapped were introduced all of the upper level timing analysis would be invalidated, which would jeopardize the incremental design approach of ASTRAL.

In what follows the proof obligations required for sequence and selection mappings without further assumptions or restrictions are discussed first followed by the proof obligations that include the further assumptions or restrictions clauses.

4.1 Proof Obligations for Selection Mappings

Let T_U be a transition of P_U refined by choosing from a number of transitions of P_L,
$T_{L.1},\ldots, T_{L.n}$ in the following way:

TU == A1 & TL.1 | A2 & TL.2 | … | An & TL.n.

In order to prove that the selection A_1 & $T_{L.1}$ | A_2 & $T_{L.2}$ | … | A_n & $T_{L.n}$ is a correct implementation of transition T_U one has to prove that:

1) At least one $T_{L.j}$, $1 \le j \le n$, fires when and only when T_U does, and

2) The effect of the $T_{L.j}$, $1 \le j \le n$, is equivalent to T_U, and its duration equals T_U's duration.

In order to prove the first point one has to show that:

1) T_U is enabled if and only if at least one $T_{L.j}$ is enabled and the corresponding A_j evaluates to *true*, $1 \le j \le n$.

$$IMPL(Entry_U) \leftrightarrow A_1 \,\&\, Entry_{L.1} \mid \ldots \mid A_n \,\&\, Entry_{L.n} \qquad [S0]$$

In order to prove the second point one has to show that whenever $T_{L.j}$ fires its effect is equivalent to the effect of T_U. This requires the following logical implication to hold for all $T_{L.j}$:

2) If both T_U and $T_{L.j}$ are enabled and $T_{L.j}$ fires then its effect implies the effect of T_U, for $1 \le j \le n$.

$$IMPL(Entry_U') \,\&\, A_j' \,\&\, Entry_{L.j}' \,\&\, Exit_{L.j} \rightarrow IMPL(Exit_U) \qquad [S1j]$$

In ASTRAL Exit assertions, variable names followed by a ' indicate the value that the variable had when the transition fired. Thus, IMPL($Entry_U$'), A_j' and $Entry_{L.j}$' refer to the starting time of T_U and $T_{L.j}$, while IMPL($Exit_U$) and $Exit_{L.j}$ refer to the ending time of T_U and $T_{L.j}$.

Finally, the duration of $T_{L.j}$ must equal the duration of T_U, for $1 \le j \le n$.

$$Dur_{L.1} = Dur_{L.2} = \ldots = Dur_{L.n} = Dur_U \qquad [S2]$$

4.2 Proof Obligations for Sequence Mappings

Let us assume that an upper level transition T_U, defined by an entry assertion $Entry_U$, exit assertion $Exit_U$ and duration Dur_U is mapped into a sequence of transitions in the lower level:

TU == WHEN $Entry_L$ DO $T_{L.1}$ BEFORE $T_{L.2}$ BEFORE … BEFORE $T_{L.n}$ OD

When $Entry_L$ holds the sequence of transitions $T_{L.1}$; $T_{L.2}$; …; $T_{L.n}$ may start. The semantics of the WHEN-DO-BEFORE clause is such that the sequence is non-interruptable, i.e., after transition $T_{L.j}$ the only transition that can fire is $T_{L.j+1}$, $1 \le j < n$.

In order to prove that the sequence $T_{L.1}$; $T_{L.2}$; …; $T_{L.n}$ is a correct implementation of transition T_U one has to prove that:

1) The sequence $T_{L.1}$; $T_{L.2}$; …; $T_{L.n}$ is enabled if and only if T_U is, and
2) The effect of the sequence $T_{L.1}$; $T_{L.2}$; …; $T_{L.n}$ is equivalent to the effect of T_U, and its duration (including any initial delay after $Entry_L$ is *true*) equals T_U's duration.

Thus, the first proof obligation requires the entry condition of T_U to be logically equivalent to the entry condition of the sequence $T_{L.1}$; $T_{L.2}$; …; $T_{L.n}$. That is:

$$IMPL(Entry_U) \leftrightarrow Entry_L \qquad [P0]$$

In order to prove the second point one can assume that transition T_U ends Now (i.e., has fired at time t_0 = Now - Dur_U) and prove that if the sequence starts at time t_0 then it ends Now and the result of the sequence is equivalent to the result of T_U.

This requires the following logical implications to hold:

1) If the sequence starts at time t_0 = Now - Dur_U (i.e., $Entry_L$ evaluates to *true*) then the first transition $T_{L.1}$ will become enabled (and therefore will fire) at time t_1 and the interval [t_1, Now] is large enough to allow all the transitions to be executed within it:

$$past(Entry_L, t_0) \rightarrow \exists\, t_1\text{: Time } (t_1 \geq t_0 \;\&\; t_1 + \sum_{k=1}^{n} Dur_{L.k} \leq Now \;\&\; past(Entry_{L.1}, t_1))$$

[P1]

The above proof obligation allows one to conclude that the first transition of the sequence will eventually be executed within the appropriate time boundaries.

In general, however it is necessary to show that all n transitions will eventually be executed. This requires n-1 additional proof obligations.

Thus, for transitions $T_{L.2}$ to $T_{L.n-1}$ one must prove that:

2) If the sequence starts at time $t_0 = Now - Dur_U$ and $T_{L.1}$, …, $T_{L.j}$ fire at time t_1, …, t_j then the j+1 transition fires at time t_{j+1} and the interval $[t_{j+1}, Now]$ must be large enough to allow the execution of transitions $T_{L.j+1}$, …, $T_{L.n}$.

$\exists\, t_1, \ldots, t_j$: Time [Pj+1]

$$\begin{aligned}
&(\quad t_j \geq t_{j-1} + Dur_{L.j-1} \;\&\; \ldots \;\&\; t_2 \geq t_1 + Dur_{L.1} \;\&\; t_1 \geq t_0 \\
&\;\&\; past(Entry_L, t_0) \\
&\;\&\; past(Entry_{L.1}, t_1) \;\&\; Start(T_{L.1}) = t_1 \\
&\;\&\; past(Exit_{L.1}, t_1 + Dur_{L.1}) \;\&\; End(T_{L.1}) = t_1 + Dur_{L.1} \\
&\ldots \\
&\;\&\; past(Entry_{L.j}, t_j) \;\&\; Start(T_{L.j}) = t_j \\
&\;\&\; past(Exit_{L.j}, t_j + Dur_{L.j}) \;\&\; End(T_{L.j}) = t_j + Dur_{L.j} \\
&\rightarrow \quad \exists\, t_{j+1}\text{: Time } (t_{j+1} \geq t_j + Dur_{L.j} \;\&\; t_{j+1} + \sum_{k=j+1}^{n} Dur_{L.k} \leq Now \;\&\; past(Entry_{L.j+1}, t_{j+1})))
\end{aligned}$$

In this way we obtain n-1 proof obligations ensuring that once the sequence has started every transition but the last one will be executed within the appropriate time boundaries.

For the last transition one must prove that:

3) If the sequence starts at time $t_0 = Now - Dur_U$ and $T_{L.1}$, …, $T_{L.n-1}$ fire at time t_1, …, t_{n-1} then the last transition fires at time t_n such that $T_{L.n}$ will end exactly Dur_U after the sequence starts, which is Now.

$\exists\, t_1, \ldots, t_{n-1}$: Time [Pn]

$$\begin{aligned}
&(\quad t_{n-1} \geq t_{n-2} + Dur_{L.n-2} \;\&\; \ldots \;\&\; t_2 \geq t_1 + Dur_{L.1} \;\&\; t_1 \geq t_0 \\
&\;\&\; past(Entry_L, t_0) \\
&\;\&\; past(Entry_{L.1}, t_1) \;\&\; Start(T_{L.1}) = t_1 \\
&\;\&\; past(Exit_{L.1}, t_1 + Dur_{L.1}) \;\&\; End(T_{L.1}) = t_1 + Dur_{L.1} \\
&\ldots
\end{aligned}$$

& past($Entry_{L.n-1}$, t_{n-1}) & Start($T_{L.n-1}$) = t_{n-1}

& past($Exit_{L.n-1}$, t_{n-1}+ $Dur_{L.n-1}$) & End($T_{L.n-1}$) = t_{n-1} + $Dur_{L.n-1}$

→ ∃ t_n: Time ($t_n \geq t_{n-1} + Dur_{L.n-1}$ & $t_n + Dur_{L.n}$ = Now & past($Entry_{L.n}$, t_n)))

Finally, the last proof obligation requires the effect of the execution of the sequence
$T_{L.1}$; …; $T_{L.n}$ to logically imply the effect of transition T_U:

4) If the sequence starts at time t_0 = Now - Dur_U and $T_{L.1}$, …, $T_{L.n}$ fire at time t_1, …, t_n then the effect of the execution of the sequence $T_{L.1}$; …; $T_{L.n}$ logically implies the effect of transition T_U.

∃ t_1, …, t_n: Time [Pn+1]

($t_n \geq t_{n-1} + Dur_{L.n-1}$ & … & $t_2 \geq t_1 + Dur_{L.1}$ & $t_1 \geq t_0$

& past($Entry_L$, t_0)

& past($Entry_{L.1}$, t_1) & Start($T_{L.1}$) = t_1

& past($Exit_{L.1}$, t_1+ $Dur_{L.1}$) & End($T_{L.1}$) = t_1 + $Dur_{L.1}$

• • •

& past($Entry_{L.n}$, t_n) & Start($T_{L.n}$) = t_n

& $Exit_{L.n}$ & End($T_{L.n}$) = Now

→ IMPL($Exit_U$)

Note that the proof obligations are meant to be carried out in an incremental way, i.e., [Pj] should be carried out after [Pj-1], $1 \leq j \leq n + 1$. In this way, if one succeeds in proving [Pj-1], he/she can (re)use the values of t_1, …, t_{j-1} in proving [Pj].

4.3 Implementing Further Assumptions and Restrictions

As mentioned before, schedules can be satisfied by exploiting further assumptions about the environment or restrictions on the system behavior. These assumptions constitute a separate part of the process specification, the further assumptions and restrictions clause. It consists of two parts: a further environment assumptions section and a further process assumptions section. Each process specification can contain one or more further assumptions and restrictions clauses. They may be used to prove that the specification satisfies the schedule.

The further environment assumptions section states further hypotheses on the admissible behaviors of the environment interacting with the system. It can be realized in a lower level only by another further environment assumptions clause.

The further process assumptions section, FPA, restricts the possible system implementations by specifying suitable selection policies in the case of nondeterministic choice between several enabled transitions or by further restricting constants. Thus, the further process assumptions represent possible implementation decisions that can make the implementation satisfy the schedule.

An FPA section is composed of a constant refinement part and a transition selection part. The constant refinement part, CR, is a sequence of clauses that may restrict the values that system constants can assume.

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

{OpSet} <Boolean Condition> {ROpSet}

where

- {OpSet} defines a set of transitions.
- {ROpSet} defines a restricted but nonempty set of transitions that must be included in the set defined by {OpSet}.
- <Boolean Condition> is a boolean condition on the state of process p.

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

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

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

The proof obligations introduced in the previous sections state that P_L is a correct implementation of P_U if every transition T_U of P_U is mapped into a sequence (selection) $T_{L.1}…T_{L.n}$ of transitions of P_L whose behavior is similar to T_U's. That is,

1) $T_{L.1}…T_{L.n}$ is enabled if and only if T_U is, and
2) the effect of $T_{L.1}…T_{L.n}$ implies the effect of $T_{U.}$

These proof obligations guarantee that the set of possible behaviors of P_L is equivalent to the set of possible behaviors of P_U, that is {B}. Showing that P_L is a correct implementation of P_U and FPA requires some modifications to the above proof obligations, since the set of possible behaviors of P_L has to be equivalent to a subset of {B}, specifically {RB}.

A transition T_U fires as soon as its entry condition evaluates to *true*, provided that no other transition is executed, i.e., whether or not T_U is enabled depends only on its entry condition. However, as a result of the TS part of FPA, T_U may not be enabled even though its entry condition evalutes to *true*. That is, whether or not T_U is

enabled may depend on its entry condition and the entry condition of other transitions of P_U. For example, consider a P_U that contains only transitions T_1 and T_2 and the FPA clause $\{T_1, T_2\}$ TRUE $\{T_1\}$. This allows T2 to fire only when its entry condition evaluates to *true* and the entry condition of T_1 evaluates to *false*. Therefore, to deal with the FPA clauses the first proof obligation should reflect this kind of global analysis of the upper level entry conditions.

Let T_{U1}, …, T_{Un} be the transitions of P_U, let $Entry_{U1}$, …, $Entry_{Un}$ be the corresponding entry conditions and let TS and CR be the further assumption clause of P_U. TS is composed of rules R_1 through R_k, where $R_i = \{OpSet_i\}\ Bool_i\ \{ROpSet_i\}$, $1 \le i \le k$. Also, let $REntry_{Uj}$ denote a new entry condition for transition T_{Uj} such that $REntry_{Uj}$ evaluates to *true* if and only if $Entry_{Uj}$ evaluates to *true* and T_{Uj} is enabled under the restrictions stated by TS. $REntry_{Uj}$ can be automatically derived by using the following algorithm:

```
Foreach TUj in PU do
  REntryUj := true
  Foreach Ri in TS do
     TMP := true
     if TUj ∈ {OpSeti} ∧ TUj ∉ {ROpSeti} then
       Foreach TUl in PU do
          case TUl ∈ {OpSeti} ∧ TUl ≠ TUj      do TMP := TMP & EntryUl od
          case TUl ∉ {OpSeti}:                 do TMP := TMP & ~EntryUl od
       od
       TMP := TMP & Booli
     fi
     REntryUj := REntryUj | TMP
  od
  REntryUj := EntryUj & ~(REntryUj)
od
```

The above algorithm builds the new entry condition for transition T_{Uj} by looking at the rules that contain T_{Uj} in the left hand side but not in the right hand side, i.e., rules that prevent T_{Uj} from firing when its entry condition evaluates to *true*.

Whenever such a rule is found, a formula representing the conditions that cause the rule to apply is built: transitions belonging to the l. h. s. of the rule must be enabled and transitions not belonging to the l. h. s. of the rule must be disabled. Furthermore, the boolean expression of the rule has to evaluate to *true* in order to apply the rule.

Each of the above formula defines a state in which transition T_{Uj} must not fire even though it is enabled. Thus, the logical disjunction of all such formulae represent all and only the states in which T_{Uj} must not fire even though $Entry_{Uj}$ evaluates to *true*.

The new entry condition $REntry_{Uj}$ is obtained as the logical conjunction of the $Entry_{Uj}$ and the negation of the formula representing all and only the states in which T_{Uj} must not fire.

Given an upper level transition T_U and a lower level sequence WHEN $Entry_L$ DO $T_{L.1}$ BEFORE … BEFORE $T_{L.n}$ OD, the proof obligations

required to show that the sequence is a correct implementation of T_U are [P0'], [P1], …, [Pn], [Pn+1'], where [P0'] and [Pn+1'] are as follows:

$$\text{IMPL}(\text{REntry}_U \,\&\, \text{CR}) \leftrightarrow \text{Entry}_L \qquad \text{[P0']}$$

$$\exists\, t_1, \ldots, t_n\text{: Time} \qquad \text{[Pn+1']}$$
$$(\quad t_n \geq t_{n-1} + \text{Dur}_{L.n-1} \,\&\, \ldots \,\&\, t_2 \geq t_1 + \text{Dur}_{L.1} \,\&\, t_1 \geq t_0$$
$$\&\ \text{past}(\text{Entry}_L, t_0)$$
$$\&\ \text{past}(\text{Entry}_{L.1}, t_1) \,\&\, \text{Start}(T_{L.1}) = t_1$$
$$\&\ \text{past}(\text{Exit}_{L.1}, t_1 + \text{Dur}_{L.1}) \,\&\, \text{End}(T_{L.1}) = t_1 + \text{Dur}_{L.1}$$
$$\ldots$$
$$\&\ \text{past}(\text{Entry}_{L.n}, t_n) \,\&\, \text{Start}(T_{L.n}) = t_n$$
$$\&\ \text{Exit}_{L.n} \,\&\, \text{End}(T_{L.n}) = \text{Now}$$
$$\rightarrow\ \text{IMPL}(\text{Exit}_U \,\&\, \text{CR})$$

Similarly, given an upper level transition T_U and a lower level selection $A_1 \,\&\, T_{L.1} \mid \ldots \mid A_n \,\&\, T_{L.n}$ the proof obligations required to show that the selection is a correct implementation of T_U are [S0'], [S1'] and [S2], where [S0'] and [S1'] are as follows:

$$\text{IMPL}(\text{REntry}_U \,\&\, \text{CR}) \leftrightarrow A_1 \,\&\, \text{Entry}_{L.1} \mid \ldots \mid A_n \,\&\, \text{Entry}_{L.n} \qquad \text{[S0']}$$

$$\text{IMPL}(\text{REntry}_U' \,\&\, \text{CR}) \,\&\, A_j' \,\&\, \text{Entry}_{L.j}' \,\&\, \text{Exit}_{L.j} \rightarrow \text{IMPL}(\text{Exit}_U \,\&\, \text{CR}) \qquad \text{[S1']}$$

5 An Example of a Refined ASTRAL Specification

The example used in this paper is a variation of the packet assembler described in [Zav 87].

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

maker should send an incomplete packet to the environment rather than wait for data to complete it."

In the remainder of this paper this system is referred to as the CCITT system.

The complete ASTRAL two level specification of the CCITT system is presented in [CKM 93]. Due to space limitations only the relevant portions of the specification are presented here. It consists of a packet maker process specification, an input process specification (of which there are N instances), and the global specification.

5.1 Process Input

The top level specification of process Input contains two variables Msg of type Message and Channel_Closed of type Boolean. It also contains two transitions New_Info and Notify_Timeout. Transition New_Info, which is exported, prepares a message to be sent to the packet maker process through a channel. The message contains a data part, which is provided by the external environment when the transition is invoked, and two other parts that allow the system to unequivocally identify which instance of process Input has produced that message and how many messages have been produced so far by that particular process instance.

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

Transition Notify_Timeout is executed when no datum is received from the external environment for more than Input_Tout time units. It prepares a message to be sent to the packet maker process containing the information that no datum has been received. Moreover, Notify_Timeout marks the channel through which messages are usually sent as being closed.

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

The second level specification of process Input contains three variables Msg, Channel_Closed and Temp_Msg and five transitions Set_Msg, Set_Proc, Set_Count, Set_Result, and Close_Channel, with durations Time1, Time2, Time3, Time4 and Time5, respectively. The variables Msg and Channel_Closed from the top level specification are identity mappings onto the second level Msg and Channel_Closed, respectively, while variable Temp_Msg in the second level specification is a "new" state variable used for holding partially constructed messages. The constants of the top level are also identically mapped to the second level.

The transitions in the second level build a temporary message by constructing each piece via a single transition. The specification for transition Set_Msg is:

```
TRANSITION Set_Msg(x: Info)          Time1
  EXIT
      Temp_Msg[Data_Part] = x
  &   Temp_Msg[Count] = Temp_Msg'[Count]
  &   Temp_Msg[ID_Part] = Temp_Msg'[ID_Part]
```

The transitions for Set_Proc, Set_Count and Close_Channel are similar, and Transition Set_Result copies the contents of Temp_Msg to Msg and sets Channel_Closed appropriately depending on the value of the Data_Part component of Temp_Msg. In addition, to relate the symbolic durations to the upper level durations, N_I_Dur and N_T_Dur, the following axiom is included in the second level specification:

```
AXIOM
      Time1 + Time2 + Time3 + Time4 = N_I_Dur
  &   Time2 + Time3 + Time4 + Time5 = N_T_Dur
```

The transitions in the second level specifications are used to implement the two top level transitions as shown by the following mappings:

```
New_Info(x) == DO Set_Msg(x) BEFORE Set_Proc BEFORE Set_Count BEFORE Set_Result OD


Notify_Timeout ==
  WHEN EXISTS t1: Time (Start(Set_Msg, t1) & Now - t1 ≥ Input_Tout) & ~Channel_Closed
  DO Close_Channel BEFORE Set_Proc BEFORE Set_Count BEFORE Set_Result OD
```

In order to prove that the second level specification is a correct implementation of the top level, one must prove that both transition New_Info and Notify_Timeout are correctely implemented.

For transition New_Info this requires the following proofs:

Proof obligation P0

Since neither New_Info nor the lower level sequence has an explicit entry condition this proof is trivially true;

Proof obligation P1

```
True → EXISTS t1: Time (t1 ≥ Now - N_I_Dur & t1 + Time1 + Time2 + Time3 + Time4 ≤ Now),
```

Using the first conjunct of the second level axiom (Time1 + Time2 + Time3 + Time4 = N_I_Dur) we conclude that t1 = Now - N_I_Dur;

Proof obligation P2

```
     past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1) = x
&    past(Temp_Msg[Count], Now - N_I_Dur + Time1) = past(Temp_Msg[Count], Now - N_I_Dur)
&    past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1) = past(Temp_Msg[ID_Part], Now -
```

N_I_Dur)
→ EXISTS t2: Time (t2 ≥ Now - N_I_Dur + Time1 & t2 + Time2 + Time3 + Time4 ≤ Now),

which is true since t2 exists and is equal to Now - N_I_Dur + Time1;

Proof obligation P3

past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1) = x
& past(Temp_Msg[Count], Now - N_I_Dur + Time1) = past(Temp_Msg[Count], Now - N_I_Dur)
& past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1) = past(Temp_Msg[ID_Part], Now - N_I_Dur)
& past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1 + Time2) = Self
& past(Temp_Msg[Count], Now- N_I_Dur+Time1+Time2) = past(Temp_Msg[Count], Now- N_I_Dur+Time1)
& past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1 + Time2) =
past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1)
→ EXISTS t3: Time (t3 ≥ Now - N_I_Dur + Time1 + Time2 & t3 + Time3 + Time4 ≤ Now),

which is true since t3 exists and is equal to Now - N_I_Dur + Time1+ Time2;

Proof obligation P4

past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1) = x
& past(Temp_Msg[Count], Now - N_I_Dur + Time1) = past(Temp_Msg[Count], Now - N_I_Dur)
& past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1) = past(Temp_Msg[ID_Part], Now - N_I_Dur)
& past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1 + Time2) = Self
& past(Temp_Msg[Count], Now- N_I_Dur+Time1+Time2) = past(Temp_Msg[Count], Now- N_I_Dur+Time1)
& past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1 + Time2) =
past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1)
& past(Temp_Msg[Count], Now - N_I_Dur + Time1 + Time2 + Time3) =
past(Temp_Msg[Count], Now - N_I_Dur + Time1 + Time2) + 1
& past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1 + Time2 + Time3) =
past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1 + Time2)
& past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1 + Time2 + Time3) =
past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1 + Time2)
→ EXISTS t4 (t4 ≥ Now - N_I_Dur + Time1+ Time2 + Time3 & t4 + Time4 = Now),

which is true since t4 exists and is equal to Now - N_I_Dur + Time1+ Time2 + Time3

Proof obligation P5

past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1) = x
& past(Temp_Msg[Count], Now - N_I_Dur + Time1) = past(Temp_Msg[Count], Now - N_I_Dur)
& past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1) = past(Temp_Msg[ID_Part], Now - N_I_Dur)
& past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1 + Time2) = Self
& past(Temp_Msg[Count], Now- N_I_Dur+Time1+Time2) = past(Temp_Msg[Count], Now- N_I_Dur+Time1)
& past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1 + Time2) =
past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1)
& past(Temp_Msg[Count], Now - N_I_Dur + Time1 + Time2 + Time3) =
past(Temp_Msg[Count], Now - N_I_Dur + Time1 + Time2) + 1
& past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1 + Time2 + Time3) =

past(Temp_Msg[Data_Part], Now - N_I_Dur + Time1 + Time2)
& past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1 + Time2 + Time3) =
past(Temp_Msg[ID_Part], Now - N_I_Dur + Time1 + Time2)
& Msg = past(Temp_Msg, Now - N_I_Dur + Time1 + Time2 + Time3)
& IF Msg[Data_Part] = Closed THEN Channel_Closed ELSE ~Channel_Closed FI
→ Msg[Data_Part] =x
& Msg[Count] = past(Msg[Count], Now - N_I_Dur) + 1
& Msg[ID_Part] = Self
&~Channel_Closed

Since all proof obligations are satisfied one can conclude that transition New_Info is implemented correctly.

For transition Notify_Timeout this requires the following proofs:

Proof obligation P0

IMPL(EXISTS t1: Time (Start(New_Info, t1) & Now - t1 ≥ Input_Tout) & ~Channel_Closed)
↔ EXISTS t1: Time (Start(Set_Msg, t1) & Now - t1 ≥ Input_Tout) & ~Channel_Closed

which is true if FORALL t (Start(Seq,t) ↔ Start(Set_Msg, t)), where Seq is the sequence into which transition New_Info is mapped. Since there is no delay between the starting time of Seq and its first transition (Set_Msg), and since Set_Msg belongs only to Seq, we conclude that the starting time of Set_Msg equals the starting time of New_Info.

Proof obligation P1

past(EXISTS t: Time (Start(Set_Msg, t) & Now - t ≥ Input_Tout) & ~Channel_Closed, Now - N_T_Dur)
→ EXISTS t1: Time (t1 ≥ Now - N_T_Dur & t1 + Time5 + Time2 + Time3 + Time4 ≤ Now),

Using the second conjunct of the second level axiom (Time1 + Time2 + Time3 + Time4 = N_I_Dur) we conclude that t1 = Now - N_I_Dur;

Proof obligation P2

past(EXISTS t: Time (Start(Set_Msg, t) & Now - t ≥ Input_Tout) & ~Channel_Closed, Now - N_T_Dur)
& past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5) = Closed
& past(Temp_Msg[Count], Now - N_T_Dur + Time5) = past(Temp_Msg[Count], Now - N_T_Dur)
& past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5) = past(Temp_Msg[ID_Part], Now - N_T_Dur)
→ EXISTS t2: Time (t2 ≥ Now - N_T_Dur + Time5 & t2 + Time2 + Time3 + Time4 ≤ Now),

which is true since t2 exists and is equal to Now - N_T_Dur + Time5;

Proof obligation P3

past(EXISTS t: Time (Start(Set_Msg, t) & Now - t ≥ Input_Tout) & ~Channel_Closed, Now - N_T_Dur)
& past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5) = Closed
& past(Temp_Msg[Count], Now - N_T_Dur + Time5) = past(Temp_Msg[Count], Now - N_T_Dur)
& past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5) = past(Temp_Msg[ID_Part], Now - N_T_Dur)

& past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5 + Time2) = Self
& past(Temp_Msg[Count], Now- N_T_Dur+Time5+Time2) = past(Temp_Msg[Count], Now-N_T_Dur+Time5)
& past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5 + Time2) =
past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5)
→ EXISTS t3: Time (t3 ≥ Now - N_T_Dur + Time5 + Time2 & t3 + Time3 + Time4 ≤ Now),

which is true since t3 exists and is equal to Now - N_T_Dur + Time5+ Time2;

Proof obligation P4

past(EXISTS t: Time (Start(Set_Msg, t) & Now - t ≥ Input_Tout) & ~Channel_Closed, Now - N_T_Dur)
& past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5) = Closed
& past(Temp_Msg[Count], Now - N_T_Dur + Time5) = past(Temp_Msg[Count], Now - N_T_Dur)
& past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5) = past(Temp_Msg[ID_Part], Now - N_T_Dur)
& past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5 + Time2) = Self
& past(Temp_Msg[Count], Now- N_T_Dur+Time5+Time2) = past(Temp_Msg[Count], Now-N_T_Dur+Time5)
& past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5 + Time2) =
past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5)
& past(Temp_Msg[Count], Now - N_T_Dur + Time5 + Time2 + Time3) =
past(Temp_Msg[Count], Now - N_T_Dur + Time5 + Time2) + 1
& past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5 + Time2 + Time3) =
past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5 + Time2)
& past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5 + Time2 + Time3) =
past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5 + Time2)
→ EXISTS t4 (t4 ≥ Now - N_T_Dur + Time5+ Time2 + Time3 & t4 + Time4 = Now),

which is true since t4 exists and is equal to Now - N_T_Dur + Time5+ Time2 + Time3

Proof obligation P5

past(EXISTS t: Time (Start(Set_Msg, t) & Now - t ≥ Input_Tout) & ~Channel_Closed, Now - N_T_Dur)
& past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5) = Closed
& past(Temp_Msg[Count], Now - N_T_Dur + Time5) = past(Temp_Msg[Count], Now - N_T_Dur)
& past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5) = past(Temp_Msg[ID_Part], Now - N_T_Dur)
& past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5 + Time2) = Self
& past(Temp_Msg[Count], Now- N_T_Dur+Time5+Time2) = past(Temp_Msg[Count], Now-N_T_Dur+Time5)
& past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5 + Time2) =
past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5)
& past(Temp_Msg[Count], Now - N_T_Dur + Time5 + Time2 + Time3) =
past(Temp_Msg[Count], Now - N_T_Dur + Time5 + Time2) + 1
& past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5 + Time2 + Time3) =
past(Temp_Msg[Data_Part], Now - N_T_Dur + Time5 + Time2)
& past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5 + Time2 + Time3) =
past(Temp_Msg[ID_Part], Now - N_T_Dur + Time5 + Time2)
& Msg = past(Temp_Msg, Now - N_T_Dur + Time5 + Time2 + Time3)
& IF Msg[Data_Part] = Closed THEN Channel_Closed ELSE ~Channel_Closed FI
→ Msg[Data_Part] =Closed

```
& Msg[Count] = past(Msg[Count], Now - N_T_Dur ) + 1
& Msg[ID_Part] = Self
& Channel_Closed
```

Since all proof obligations are satisfied one can conclude that transition Notify_Timeout is implemented correctly.

Thus, one can conclude that the second level specification of process Input is a correct implementation of the top level specification.

5.2 Process Packet_Maker

The packet maker specification has three variables Packet and Output of type Message_List, and Previous(Receiver_ID) of type Time; and it has two transitions: Process_Msg and Deliver, which correspond to processing a message from an input channel and delivering a packet, respectively.

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


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

Furthermore, the specification contains two different sets of further assumptions and restrictions that were used to prove the schedule. Those assumptions were regarded in the top level specification as possible ways of satisfying the schedule [CKM 92]. Neither of them was explicitly chosen, since the designer felt that such a decision was not appropriate at that design level. However in the second level specification he/she chose to implement the first set of process assumptions, which is reported below:

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

The second level specification is composed of three variables, one constant and two transitions all of which are identically mapped from their upper level counterparts.

It is therefore necessary to prove that the entry conditions of both lower level transitions are logically equivalent to their upper level counterparts when FURTHER ASSUMPTIONS #1 is taken into account. Thus, one has first to derive the new entry conditions for transitions Process_Msg and Deliver by means of the algorithm introduced in the previous section.

Because there is only one rule in the TS part and Process_Msg appears in both the l.h.s. and the r.h.s. of the rule $REntry_{UProc_Msg}$ is equal to $Entry_{UProc_Msg}$. Transition Deliver, in contrast, appears in the l.h.s. of the rule but not in the r.h.s.; therefore, $REntry_{UDel}$ is equal to ($Entry_{UDel}$ & ~($Entry_{UProc_Msg}$)).

Let us consider P_L's transition Deliver:

```
TRANSITION Deliver  Del_Dur
  ENTRY
      LIST_LEN(Packet) = N/L
  |   (    LIST_LEN(Packet) > 0
      &    Receiver[R_id].End(New_Info) ≤ Previous(R_id)
      &    (    Receiver[R_id].Msg[Data_Part]≠Closed
           |    past(Receiver[R_id].Msg[Data_Part],Previous(R_id)) = Closed)
  EXIT
      Output = Packet'
  &   Packet = EMPTY
```

Because the implementation mapping is a selection proof obligation [S0'] requires the following logical equivalence to hold:

Proof Obligation S0'

$IMPL(REntry_{UDel}$ & Del_Tout = 0 & Maximum = N/L) ↔ $Entry_{LDel}$

that is,

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

} IMPL ($Entry_{UDel}$)

```
& ~(   LIST_LEN(Packet) < Maximum
    &   (    Receiver[R_id].End(New_Info) > Previous(R_id)
        |(     Receiver[R_id].Msg[Data_Part] = Closed
           &  past(Receiver[R_id].Msg[Data_Part],
                                    Previous(R_id)) ≠ Closed)))
& Del_Tout = 0 & Maximum = N/L

↔ LIST_LEN(Packet) = N/L
   |  (   LIST_LEN(Packet) > 0
      &  Receiver[R_id].End(New_Info) ≤ Previous(R_id)
      &  ( Receiver[R_id].Msg[Data_Part] ≠ Closed
         |   past(Receiver[R_id].Msg[Data_Part],
                                    Previous(R_id)) = Closed)),
```

} ~ IMPL (Entry UProc _ Msg)
} IMPL(CR)

} Entry LDel

which can be shown to hold simply by substituting for Del_Tout and Maximum and then applying DeMorgan's formulae.

Proof Obligation S1'

```
        LIST_LEN(Packet') = Maximum
   |      (    LIST_LEN(Packet') > 0
      &   ( EXISTS t:Time (past(Start(Deliver, t), Now- Del_Dur) &  Now - t ≥ Del_Tout - Del_Dur)
          | Now = Del_Tout + N_I_Dur))

&  Del_Tout = 0 & Maximum = N/L

&    LIST_LEN(Packet') = N/L
   |   (  LIST_LEN(Packet') > 0
      &   past(Receiver[R_id].End(New_Info), Now- Del_Dur) ≤ Previous(R_id)'
      &   (    Receiver[R_id].Msg[Data_Part]' ≠ Closed
          |    past(Receiver[R_id].Msg[Data_Part],Previous(R_id)') = Closed))

&       Output = Packet'
    &   Packet = EMPTY

→       Output = Packet'
    &   Packet = EMPTY

&   Del_Tout = 0 & Maximum = N/L
```

which is trivially true

Proof Obligation S2

```
Del_Dur = Del_ Dur
```

which is trivially true

Thus one conclude that transition Deliver is implemented correctely in the secondlevel specification.

Let us consider transition Process_Msg:

Proof Obligation S0'

```
        LIST_LEN(Packet) < Maximum
    &  (  Receiver[R_id].End(New_Info) > Previous(R_id)
```

```
          |    (   Receiver[R_id].Msg[Data_Part]=Closed
               &     past(Receiver[R_id].Msg[Data_Part],Previous(R_id))≠Closed))

&  Del_Tout = 0 & Maximum = N/L

↔      LIST_LEN(Packet) < N/L
   &  (  Receiver[R_id].End(New_Info) > Previous(R_id)
          |    (  Receiver[R_id].Msg[Data_Part]=Closed
               &   past(Receiver[R_id].Msg[Data_Part],Previous(R_id))≠Closed))
```

Proof Obligation S1'

```
          LIST_LEN(Packet') < Maximum
   &  (  past(Receiver[R_id].End(New_Info), Now- PM_Dur) > Previous'(R_id)
          |    (   Receiver[R_id].Msg'[Data_Part]=Closed
               &     past(Receiver[R_id].Msg[Data_Part],Previous'(R_id))≠Closed))

&  Del_Tout = 0 & Maximum = N/L

&         LIST_LEN(Packet') < N/L
   &  (  past(Receiver[R_id].End(New_Info), Now- PM_Dur) > Previous'(R_id)
          |    (   Receiver[R_id].Msg'[Data_Part]=Closed
               &     past(Receiver[R_id].Msg[Data_Part],Previous'(R_id))≠Closed))

&      Packet = Packet' CONCAT LIST(Receiver[R_id].Msg)
   &  Previous(R_id) BECOMES Now

→      Packet = Packet' CONCAT LIST(Receiver[R_id].Msg)
   &  Previous(R_id) BECOMES Now

&  Del_Tout = 0 & Maximum = N/L
```

which is trivially true

Thus, one can conclude that the second level specification of the CCITT system is a correct implementation of the top level specification. Furthermore, since in the second level specification a set of further assumptions has been "hardwired" into the transition specifications, the second level specification satisfies both the invariants and the schedules.

6. Conclusions and Future Directions

In this paper the refinement of process specifications using layering and implementation mappings, which were only briefly sketched in previous papers, were presented in detail. The inter-level proof obligations were also presented and several example proofs were demonstrated.

All of the inter-level proofs for the CCITT specification have been completed. The details of these proofs as well as the complete two-level CCITT specification can be found in [CKM 93].

Future work will concentrate on applying ASTRAL to more varied and complex realtime systems. Work will also continue on a formal semantic definition for

ASTRAL and on building a tool suite for formally designing realtime systems using ASTRAL.

References

[ACD 90] Alur, R., C. Courcoubetis, and D. Dill, "Model-Checking for Realtime Systems," *5th IEEE LICS 90,* IEEE, pp. 414-425, 1990.

[CKM 92] Coen-Porisini, A., R. Kemmerer and D. Mandrioli, "Formal Verification of Realtime Systems in ASTRAL", Report no. TRCS 92-22, Department of Computer Science, University of California, Santa Barbara, California, September 1992.

[CKM 93] Coen-Porisini, A., R. Kemmerer and D. Mandrioli, "A Formal Framework for ASTRAL Inter-level Proof Obligations", Report no. TRCS 93-09, Department of Computer Science, University of California, Santa Barbara, California, April 1993.

[CHS 90] Chang, C., H. Huang, and C. Song, "An Approach to Verifying Concurrency Behavior of Realtime Systems Based On Time Petri Net and Temporal Logic," *InfoJapan 90,* IPSJ, pp. 307-314, 1990.

[FMM 91] Felder, M., D. Mandrioli, and A. Morzenti, "Proving Properties of Realtime Systems through Logical Specifications and Petri Net Models," Tech. Rept. 91-72, Dip. di Elettronica-Politecnico di Milano, December 1991.

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

[GK 91a] Ghezzi, C. and R. Kemmerer, "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 R. Kemmerer, "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.

[Ost 89] Ostroff, J., *Temporal Logic For Realtime Systems,* Research Studies Press LTD., Taunton, Somerset, England , Advanced Software Development Series, 1, 1989.

[Pnu 77] Pnueli, A., "The Temporal Logic of Programs," *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pp. 46-57, 1977.

[Suz 90] Suzuki, I., "Formal Analysis of Alternating Bit Protocol by Temporal Petri Nets," *IEEE-TSE 16*, 11, pp. 1273-1281, November 1990.

[Zav 87] Zave, P., PAISLey User Documentation Volume 3: Case Studies, Computer Technology Research Laboratory Report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1987.

Appendix ASTRAL Formal Specification for the CCITT System

```
GLOBAL Specification CCITT

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

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

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

AXIOM
  N MOD L = 0

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

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

INVARIANT
/* Every data output was received sometime in the past */
    FORALL k:Integer (k > 0 & k ≤ LIST_LEN(Output) & Output[k] [Data_Part] ≠ Closed
    → EXISTS i:Receiver_ID, t:Time, j:Integer (t < Now & Receiver[i].Start-j(New_Info(Output[k]
[Data_Part])=t))
```

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

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

SCHEDULE

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

FORALL i:Receiver_ID, t_1:Time, x:Info
(t_1 ≤ Now - N/L*P_M_Dur - N_I_Dur - Del_Dur
& past(Receiver[i].Call(New_Info(x)),t_1)=t_1
→ EXISTS t_2:Time, k:Integer
(t_2 = t_1 + N/L*P_M_Dur + N_I_Dur + Del_Dur & 0 < k & k ≤ LIST_LEN(past(Output,t2))
& Change(Output,t_2) & past(Output[k][Data_Part],t_2)=x & past(Output[k][ID_Part],t_2) = Receiver[i].Id))

END CCITT

```
SPECIFICATION Input

LEVEL Top_Level

IMPORT
  Data, Message, Info, Closed, N_I_Dur

EXPORT
  New_Info, Msg

VARIABLE
  Msg: Message,
  Channel_Closed:Boolean

CONSTANT
  Input_Tout, N_T_Dur: Time

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

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

INVARIANT
/* After Input_Tout time units have elapsed without receiving any new message a timeout occurs */
    FORALL t1: Time (Start(New_Info, t1) & Now - t1 > Input_Tout
    → EXISTS t2: Time (Start(Notify_Timeout, t2) & t2 = t1 + Input_Tout))

/* The last received message is kept until a timeout occurs */
&   FORALL t1: Time, x: Info
    (    (End(New_Info(x), t1) & Now - t1 < Input_Tout - N_I_Dur + N_T_Dur → Msg[Data_part] = x)
    &   (End(New_Info(x), t1) & Now - t1 ≥ Input_Tout - N_I_Dur + N_T_Dur → Msg[Data_part] =
Closed))

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

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

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

END Top_Level
```

```
LEVEL Second_Level

IMPORT
  Data, Message, Info, Closed, N_I_Dur

EXPORT
  New_Info, Msg

VARIABLE
  Msg, Temp_Msg: Message,
  Channel_Closed:Boolean

CONSTANT
  Input_Tout, N_T_Dur, Time1, Time2, Time3, Time4, Time5: Time

AXIOM
  Time1 + Time2 + Time3 + Time4 = N_I_Dur
& Time2 + Time3 + Tim4 + Time5 = N_T_Dur

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

TRANSITION Set_Msg(x:Info)              Time1
  EXIT
      Temp_Msg[Data_Part] = x
  &   Temp_Msg[Count] = Temp_Msg'[Count]
  &   Temp_Msg[ID_Part] = Temp_Msg'[ID_Part]

TRANSITION Set_Proc                     Time2
  EXIT
      Temp_Msg[Data_Part] = Temp_Msg'[Data_Part]
  &   Temp_Msg[Count] = Temp_Msg'[Count]
  &   Temp_Msg[ID_Part] = Self

TRANSITION Set_Count            Time3
  EXIT
      Temp_Msg[Data_Part] = Temp_Msg'[Data_Part]
  &   Temp_Msg[Count] = Temp_Msg'[Count] + 1
  &   Temp_Msg[ID_Part] = Temp_Msg'[ID_Part]

TRANSITION Set_Result           Time4
  EXIT
      Msg = Temp_Msg'
  &   IF Msg[Data_Part] = Closed
        THEN Channel_Closed
        ELSE ~Channel_Closed
      FI

TRANSITION Close_Channel        Time5
  EXIT
      Temp_Msg[Data_Part] = Closed
  &   Temp_Msg[Count] = Temp_Msg'[Count]
  &   Temp_Msg[ID_Part] = Temp_Msg'[ID_Part]
```

IMPLEMENTATION
  Msg == Msg,
  Channel_Closed == Channel_Closed,
  Input_Tout == Input_Tout,
  N_T_Dur == N_T_Dur,
  New_Info(x) == DO Set_Msg(x) BEFORE Set_Proc BEFORE Set_Count BEFORE Set_Result OD,
  Notify_Timeout == WHEN EXISTS t1: Time (Start(Set_Msg,t1) & Now - t1 ≥ Input_Tout) &
~Channel_Closed
                    DO Close_Channel BEFORE Set_Proc BEFORE Set_Count BEFORE
Set_Result OD

END Second_Level

END Input

```
SPECIFICATION Packet_Maker

LEVEL Top_Level

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

EXPORT
  Output

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

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

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

&   FORALL t:Time ((t - N_I_Dur) MOD (N/L*P_M_Dur + Del_Dur) ≠ 0
    → FORALL i:Receiver_ID (~Receiver[i].End(New_Info) = t))

&   FORALL i:Receiver_ID (Receiver[i].Msg[Data_Part] ≠ Closed)

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

INVARIANT
/*Changes in Output occur at and only at the end of a Deliver*/
    FORALL t: Time (Change(Output, t) ↔ past(End(Deliver), t) = t)

/* No new messages are generated by the packet assembler */
&   FORALL k:Integer (k>0 & k≤LIST_LEN(Output)
    → EXISTS i:Receiver_ID, t:Time (t<Now & past(Receiver[i].Msg,t)=Output[k]) )

/*The order that messages appear in an output packet is the one in which they were processed from
the channels*/
&   FORALL k:Integer (k>0 & k<LIST_LEN(Output)
    → EXISTS t1,t2:Time
        (    t1 < t2 < Now & past(End(Process_Message), t1) = t1 & past(End(Process_Message), t2) = t2
        &   Output[k] = past(Packet[past(LIST_LEN(Packet), t1)], t1)
        &   Output[k+1] = past(Packet[past(LIST_LEN(Packet), t2)], t2))

/* The order is also preserved across output packets */
&   EXISTS t:Time  (Start-2(Deliver, t) & End(Deliver) > Start(Deliver))
    → EXISTS t1,t2:Time
        (    t1 < t2 < Now & past(End(Process_Message, t1) = t1 & past(End(Process_Message, t2) = t2
        &   past(Output[past(LIST_LEN(Output), Start(Deliver)], Start(Deliver)) =

  past(Packet[past(LIST_LEN(Packet),t1)], t1)
        &   Output[1] = past(Packet[past(LIST_LEN(Packet),t2)], t2))
```

```
/* Every message in Output was previously in Packet and all of the elements of Packet have not
changed from
   when they were put into the packet until the packet is output*/
&  FORALL k:Integer (k>0 & k≤LIST_LEN(Output)
   ↔ EXISTS t:Time
      (   t<End(Deliver) & past(End(Process_Message,t)=t &
past(Packet[past(LIST_LEN(Packet),t)],t)=Output[k]
      &  FORALL t1:Time (t1≥t & t1<End(Deliver)
         → past(Packet[past(LIST_LEN(Packet), t)], t) = past(Packet[past(LIST_LEN(Packet), t)],
t1))))

&  FORALL t1:Time (t1≤Now-H3 & past(End(Process_Msg),t1)=t1
   → EXISTS t2:Time
      (   t2>t1 & t2 ≤ Now & past(End(Deliver),t2)=t2
      &  past(Packet[past(LIST_LEN(Packet), t1)], t1) = past(Output[past(LIST_LEN(Packet), t1)],
t2)
      & FORALL t:Time (t≥t1 & t<t2
         → past(Packet[past(LIST_LEN(Packet), t1)], t1) = past(Packet[past(LIST_LEN(Packet),
t1)], t))))

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

TRANSITION Process_Msg(R_id:Receiver_ID)  P_M_Dur
  ENTRY
/*Packet is not full and either (a) the present message has been produced after the last message that
has been processed from that channel, or (b) the value of the current message is Closed and the value
of the previously processed message for that channel was not Closed*/
      LIST_LEN(Packet) < Maximum
  &  (   Receiver[R_id].End(New_Info) > Previous(R_id)
      |      (Receiver[R_id].Msg[Data_Part]=Closed &
past(Receiver[R_id].Msg[Data_Part],Previous(R_id))≠Closed))
  EXIT
      Packet = Packet' CONCAT LIST(Receiver[R_id].Msg)
  &  Previous(R_id) BECOMES Now

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

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

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

END Top_Level

```
LEVEL Second_Level

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

EXPORT
   Output

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

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

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

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

TRANSITION Deliver  Del_Dur
   ENTRY
     LIST_LEN(Packet) = N/L
   |    (    LIST_LEN(Packet) > 0
     &   Receiver[R_id].End(New_Info) ≤ Previous(R_id)
     &   (    Receiver[R_id].Msg[Data_Part] ≠ Closed
          |    past(Receiver[R_id].Msg[Data_Part],Previous(R_id)) = Closed))
   EXIT
        Output = Packet'
   &   Packet = EMPTY

IMPLEMENTATION
   Packet == Packet,
   Previous == Previous,
   Output == Output,
   H3 == H3,
   Maximum == N/L,
   Del_Tout == 0,
   Process_Msg == Process_Msg,
   Deliver == Deliver

END Second_Level

END Packet_Maker
```