Communicating TILCO: a Model for Real-Time System Specification

Pierfrancesco Bellini **Paolo Nesi**
Dipartimento Sistemi e Informatica, University of Florence
Via S.Marta, 3 50139, Firenze - Italy
+39-0554796523, nesi@dsi.unifi.it, http://www.dsi.unifi.it/~nesi

ABSTRACT

Formal techniques for the specification of real-time systems must be capable of describing a set of relationships expressing the temporal constraints among events and actions: properties of invariance, precedence, periodicity, liveness and safety conditions, etc. This paper describes CTILCO, an extension of TILCO (Temporal Interval Logic with Compositional Operators). CTILCO introduces the Communication among components specified in TILCO and allows the adoption of decomposition/composition mechanisms. TILCO has been expressly designed for the specification of real-time systems. CTILCO is based on time intervals and can concisely express temporal constraints with time bounds, such as those needed to specify real-time systems. It can be used to verify the completeness and consistency of specifications, as well as to validate system behavior against its requirements and general properties. CTILCO has been formalized by using the theorem prover Isabelle/HOL. CTILCO specifications satisfying certain properties are executable. CTILCO is defined in terms of theorems and allows the system specification and the formal proof of properties including composition/decomposition with communications. An example of system specification and validation has been also included.

Keywords: formal specification language, first order logic, temporal interval logic, verification and validation, real-time systems.

1 INTRODUCTION

Applications of avionics, robotics, process control, patient monitoring, etc., frequently must meet temporal constraints in order to avoid critical or degenerative conditions. These applications are typically modeled as real-time systems by using suitable specification techniques. For their specification, a set of relationships expressing temporal constraints among events must be used – e.g., [1], [2] – for example: properties of invariance, precedence among events, periodicity, liveness and safety conditions, etc. The specification correctness in meeting the temporal constraints has to be demonstrated by using verification and validation techniques.

For these reasons, formal specification techniques are presently considered the best tools for the specification of real-time systems (see [1] for a survey). Most of the formal methods allow the verification and validation of the specification with respect to system requirements and/or to real stimuli by using classical and symbolic model-checking techniques. These approaches allow verifying the most critical aspects and use-cases in limited time. To guarantee the absolute reliability of the specifications is still an open problem since the costs of exhaustive verification and validation with model-checking techniques are often unmanageable. For these cases, a solution is to demonstrate the satisfactory of specific system properties and behavior by using theorem prover approaches [3],[4].

Composition/decomposition techniques are mechanisms used to cope with the general system complexity. Most of software development methodologies address the structural composition/decomposition of the systems. A composite object is defined in terms of its subobject/components and their relationships. Object-based and object-oriented approaches include and formalize composition/decomposition concepts. Different communication mechanisms among components, as shared variables, synchronous or asynchronous communications are chosen. Components can be separately developed, tested and then combined to model the whole system. Problems arise when the combination of components produces unexpected and, thus, difficulty controllable and verifiable behavior due to the presence of communication among components. To this end, verification and validation criteria for compositional methods are used – e.g., [5], [7]. These must address the verification and validation of component-composition and their relationships with the requirements of the composite object.

For complex and large systems, the compositional approaches are typically accompanied by the availability of a layering support. The verification of consistency between the composite object and its components at each level of the structural hierarchy guarantees the satisfaction of the abstract specification and therefore of the system requirements – for example, [6], [7], [8],

[9], [10], [11].

For the specification of real-time systems temporal logics have been profitably used (see [5] for a very large survey on temporal logics), and they can be used also for the validation of the system under specification. In particular, the temporal logic TILCO (Temporal Interval Logic with Compositional Operators) has been defined with a special emphasis on its expressiveness and conciseness [3]. TILCO has been designed for the specification of real-time systems, it extends FOL with a set of temporal operators and can be regarded as a generalization of the classical temporal logics operators *eventually* and *henceforth* to time intervals [5]. TILCO has a metric for time, the time is discrete and no explicit temporal quantification is allowed. TILCO allows the definition of expressions stating the ordering relationships among events, delays, time-outs, periodicity, liveness and safety conditions, etc. These features are mandatory to specify the behavior of real-time systems.

In this paper, CTILCO (Communicating TILCO, Temporal Interval Logic with Compositional Operators) is presented. CTILCO has been defined since TILCO does not provide facilities for the specification of complex/wide systems. To this end, CTILCO permits the decomposition of the system in a hierarchy of communicating processes. Processes communicate using message-passing primitives on synchronous ports. The communication between processes is based on typed synchronous input/output ports connected through channels. The connection is 1:1. Each output port is connected to at most one input port and vice versa. In the following, the mechanisms to model CTILCO processes are introduced. In the next sections, the formalization of communication between processes in TILCO and the mechanisms used for reasoning about communicating processes are presented. The CTILCO theory in Isabelle is available on http://www.dsi.unifi.it/~pbellini/tilco/ together with several other details about TILCO family of logic.

This paper is organized as follows. Section 2 briefly presents TILCO temporal logic. Section 3 presents a CTILCO overview. Section 4 shows the communication model used in CTILCO: low-level and the high-level communication constructs with their semantics expressed in TILCO. Section 5 briefly highlights the validation methods usable in CTILCO specifications. Section 6 provides an example of specification to show the composition/decomposition capabilities of CTILCO. Conclusions are drawn in Section 7.

2 TILCO OVERVIEW

In TILCO, the same formalism used for system specification is employed to describe high-level properties that should be satisfied by the system itself. These properties must be proven on the basis of the specification in the system validation phase. Consequently, a formalization of TILCO has been implemented in the theorem prover Isabelle/HOL [4], [3]. Using this formalization, a set of fundamental theorems has been proven and a set of tactics has been built to support the semi-automatic demonstration of TILCO specifications' properties. Causal TILCO specifications are also executable by using an inferential engine and algorithm.

TILCO's *temporal operators* have been added to FOL by leaving the evaluation time implicit. The meaning of a TILCO formula is given with respect to the current time such as in many other logical languages — e.g., MTL, TRIO, see [12], [13], [5], [3]. TILCO has been compared with MTL, TRIO, ITL, IL, PTL, EIL, RTIL, CTL, RTL, and other temporal logics in [5].

This approach has been demonstrated to be the best solution for writing simple and reusable specifications predicates. Time is discrete and linear, and the temporal domain is $\mathcal{Z}$, the set of integers. The current time instant is represented by 0, whereas positive (negative) numbers represent future (past) time instants. In TILCO, the basic temporal entity is the time interval. Intervals can be quantitatively expressed by using the notation with round, "(", ")", or squared, "[", "]", brackets for excluding and including interval boundaries, respectively. Time instants are considered as special cases represented as closed intervals which are composed of a single point (e.g., $[a, a]$). When infinite intervals are used, the extremes are open, then symbols $+\infty$ and $-\infty$ are used as interval boundaries.

The basic TILCO *temporal operators* are:

- "$A@i$" is true if formula A is true in every instant in interval i, with respect to the current time instant;
- "$A?i$" is true if formula A is true in at least one instant in the interval i, with respect to the current time instant;
- "**until** A B" is true if either predicate B will always be true in the future, or it will be true until predicate A will become true;
- "**since** A B", is true if either predicate B has always been true in the past, or it has been true since predicate A has to become true.

$A@i$ is true if formula A is true in every time instant in interval i, with respect to the current time instant. Therefore, if t is the current time instant, $A@i$ represents a constraint on A considering the interval i with respect to the evaluation time instant t, that is $(A@i)^{(t)} \equiv \forall x \in i.A^{(x+t)}$. The notation used with $^{(t)}$ is to put in evidence the evaluation time instant. In the following it is obviously omitted such as in several other implicit time temporal logics. This approach is called implicit time and is used in RTL, TRIO and in many other temporal logics [5]. In particular, $A@[t_1, t_2)$ evaluated in t means:

$$\forall x \in [t_1, t_2).A^{(x+t)}.$$

Obviously t_1 and t_2 can be either positive or negative, thus the interval can be in the past and/or in the future, respectively. If the lower bound of an interval is greater than the upper bound, the interval is null. In the temporal domain, operators "@" and "?" correspond to FOL quantifiers $\forall$ and $\exists$, respectively; hence, they are related by a duality relationship analogous to that between $\forall$ and $\exists$. "@" and "?" operators are used to express delays, time-outs and any other sort of temporal constraint that requires a specific quantitative bound. Concerning the other *temporal operators*, **until** A B (evaluated in t) is true if B will always be true in the future with respect to t, or if B will be true in the interval $(t, x+t)$ with $x > 0$ and A will be true in $x + t$. This definition of **until** does not require the occurrence of A in the future, so the **until** operator corresponds to the *weak until* operator defined in PTL [14]. The operators **until** and **since** can be effectively used to express ordering relationships among events without specifying any numeric constraint.

until A B operator does not consider the evaluation time instant as an instant where A could happen, then operator $\mathbf{until_0}$ has been introduced. It is defined as:

$$\mathbf{until_0}\ A\ B \equiv A \vee (B \wedge \mathbf{until}\ A\ B)$$

and also a "strong" **until** is sometimes needed. For this reason the operator **until'** has been defined as:

$$\mathbf{until'}\ A\ B \equiv A?(0, +\infty) \wedge \mathbf{until}\ A\ B$$

For completeness, the $\mathbf{until'_0}$ has been defined as:

$$\mathbf{until'_0}\ A\ B \equiv A?[0, +\infty) \wedge \mathbf{until_0}\ A\ B$$

In a similar manner, $\mathbf{since_0}$, **since'** and $\mathbf{since'_0}$ operators have been also defined.

In a TILCO specification, predicates and functions with typed parameters can also be defined. Predicates return a value of type **bool**. The body of each predicate must be specified by means of a TILCO formula, where the only non-quantified (free) variables are the predicate parameters. Predicates are an instrument to make formulæwriting simpler; hence, more complex temporal expressions and formulæ can be hidden in predicates. For example, the two predicates:

$$\begin{array}{lcl} \mathbf{rule}(A : \mathrm{bool}) & \stackrel{\mathrm{def}}{=} & A@(-\infty, +\infty) \\ \mathbf{up}(A : \mathrm{bool}) & \stackrel{\mathrm{def}}{=} & A \wedge \neg A@[-1, -1] \end{array}$$

where: **rule** expresses that a predicate A is always true and **up** means that A from false becomes true. Predicates with parameters are often used in specifications to have shorter and easily readable formulæ.

$A@[0,t)$	**A is true from now for t time instants**
$A?(0,+\infty)$	**A will be sometimes true in the future**
$A@[t_1,t_2]$	**A is true in $[t_1,t_2]$**
$A?[t_1,t_2)$	**A is true in an instant of $[t_1,t_2)$**
$\neg(A@(-\infty,+\infty))$	**A is not always true**
$A@[t_1,t_1],(t_2,t_3]$	**A is true in t_1, and in $(t_2,t_3]$**
$A@[t_1,t_1];(t_2,t_3]$	**A is true in t_1, or in $(t_2,t_3]$**
$A@[t,t] \wedge \neg A@(0,t)$	**t is the next time instant when A will be true**
$A?[0,t_1]@[0,+\infty)$	**A will become true within t_1 for each time instant in the future (response)**
$(A \Rightarrow B)?[0,t]$	**if A is true within t, then also B will be true at the same time**
$(A \Rightarrow B?i)@j$	**A leads to an assertion of B in i for each time instant of j**

Table 1: Examples of TILCO formulæ.

In Tab. 1, in order to provide a clearer view of TILCO expressiveness, some examples of formulæ are reported with an explanation of their meaning, where t stands for a positive integer number.

3 CTILCO OVERVIEW

In CTILCO, a system specification is a hierarchy of communicating processes, the specifications of which are written in TILCO. Many instances of the same process can be present in the specification. Processes can have some parameters and every instance has distinct values. Several temporal logics do not support the communication. They can be used to define communication protocols. The research presented in this paper can be used by other research groups to implement communication mechanisms. Examples can be recovered in [15], [16], [17]. The novelty of our approach consists in the fact that CTILCO is supported by a Theory in Isabelle. Therefore, it can be profitably used for theorem-based proof of the specification. In most of the mentioned approaches the verification is based on Model or History checking.

Communication between processes is based on typed synchronous input/output ports connected by channels. CTILCO presents a communication model quite similar to that of CSP, while asynchronous models (such as that of CCS) can be build by using a intermediate buffer [1]. The connection is 1:1, each output port is connected to at most one input port and viceversa. In the following, the way in which processes are modeled in CTILCO is introduced. In the next sections, the formalization of communication between processes in TILCO and the methods used to reason about communicating processes are presented.

In the following, a *process* represents a class according to an object-based formalism.

In CTILCO a process is represented by two views:

1. the *external view* that describes the input/output behavior of the process;

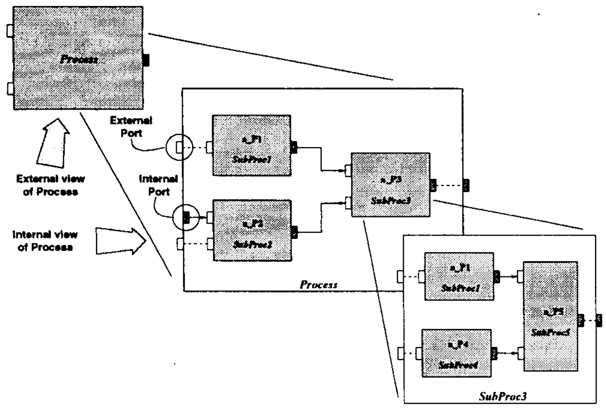

Figure 1: External and internal representation of a CTILCO process

2. the *internal view* that describes the process decomposition into subprocesses or a low-level formalization of the process behavior if it cannot be furtherly decomposed.

A CTILCO process is *externally* characterized by:

- a set of external *input* ports used to acquire information from the outside;
- a set of external *output* ports used to produce information to the outside;
- a set of external *variables* used to give some general information about the process state or to simplify the external behavior specification;
- a set of external *parameters* used to permit general process specification to make easy process reuse, since different process instances may have different parameters;
- a set of external TILCO *formulæ* that describe the external process behavior by means of message exchanging and constraints on the external variables,

CTILCO is *internally* characterized by:

- a set of CTILCO *subprocesses*;
- a set of internal *input* ports, used to get information from subprocesses;
- a set of internal *output* ports used to send information to subprocesses;
- a set of internal *variables*;
- a set of internal TILCO *formulæ*, which describe the internal behavior of the process.

The ports of subprocesses can be directly connected to the containing process ports (of the same type, input to input and output to output) or can be connected through channels to the complementary internal ports (output to input and input to output). In Fig. 1, a decomposition is exemplified. The use of internal ports permits the realization of *partial* decompositions, when the process behavior is only partially specified by subprocesses and, thus, some interactions with the subprocesses are stated by means of the TILCO formulæ of the internal specification.

In TILCO formulæ, the *dot* notation is used to provide access to process components. For example, if p is a process with a variable v, then $p.v$ is used to refer to the variable of p. Whether process p has a subprocess s with a variable v, then $p.s.v$ is used to provide access to the subprocess variable.

Since many instances of the same process can be present in the system, its specification is valid for all of them. For example, if the internal specification of a process with a variable *ivar* includes the following formula:

$$:ivar = 1 \Rightarrow (:ivar = 0)@[20, 20]$$

It means that if *ivar* is equal to 1, then after 20 time instants *ivar* will be equal to 0. This will be true in each process independently. By means of *colon* operator, process and local variables it can be easily distinguished.

Being in TILCO the time axis infinite in both directions, there is not any time instant that can be regarded as the *start* time instant of the execution process. In the specification of a system, it is natural to think about a reference time instant in which the process starts its work, and before that the signals were stable. For this reason, a Boolean variable *process_start* has been introduced to each process. This variable is true only in one time instant for each process. It should be noted that each process has its own start instant and a formula of the internal specification is used to define the start time instant of its subprocesses. Typically when a process starts all its subprocesses start as well.

4 CTILCO COMMUNICATION MODEL

The communication between two processes is structured in two layers: the *low-level* communication model for transmission of typed messages and of acknowledgements (ACKs); and the *high-level* communication model that uses the low-level to realize a synchronous communication protocol. The two layers solution has been adopted in order to keep separate the level in which the simple unidirectional actions of sending, receiving, asking, etc., are used and the high level used in the effective specifications. This solution allows reusing the specification predicates in a simpler manner.

Low-level communication

Properties assumed for the low-level are:

- **no data creation**: an arrived message (or ACK) has been surely sent;
- **no data loss**: a sent message (or ACK) will be received;
- **constant delay**: a sent message (or ACK) will be received after a constant delay greater or equal than zero.

The *no data creation* assumption is fundamental (without this assumption the communication has no sense). The *no data loss* and *constant delay* assumptions have been introduced in order to have a deterministic behavior. From these assumptions, the *no reorder* property can be derived (messages arrive in the same order as they are sent).

In this layer, the following temporal predicates have been defined and, thus, can be used by the higher-level:

<outPort>.send(<expr>)
: is true when output port <outPort> sends the value obtained BY evaluating expression <expr>.

<outPort>.receiveAck
: is true when an ACK has been received by output port <outPort>.

<inPort>.receive(<expr>)
: is true when a message has been received by input port <inPort> with the value indicated by <expr>.

<inPort>.sendAck
: is true when input port <inPort> sends an acknowledgement.

There is also a *connection* predicate between ports:

$$outP \xrightarrow{d} inP$$

that asserts that output port *outP* is connected to input port *inP* and messages (and ACKs) sent are delayed of *d* time instants. Please note that *connections* are static assertions, design-fixed.

The rules to manage low-level communication are reported in the following.

message transmission:

$$(outP \xrightarrow{d} inP) \Rightarrow$$
$$\text{rule}(outP.\text{send}(k) \iff inP.\text{receive}(k)@[d, d])$$

This rule states: if port *outP* is connected to port *inP* then in every time instant, *outP* sends a message if and only if *inP* receives the same message after *d* time instants. From this rule, we have that the message sent is received after *d* time instants (no data loss) and that the message received has been sent *d* time instants ago (no creation).

ack transmission:

$$(outP \xrightarrow{d} inP) \Rightarrow$$
$$\text{rule}(inP.\text{sendAck} \iff outP.\text{receiveAck}\ @[d, d])$$

This rule is similar to the previous except for dealing with the ACKs and having an opposite direction (from input port to output port).

send one value:

$$\text{rule}(outP.\text{send}(k) \wedge outP.\text{send}(v) \Rightarrow k = v)$$

This rule states: if at the same time instant two values are sent on the same port these values have to be equal.

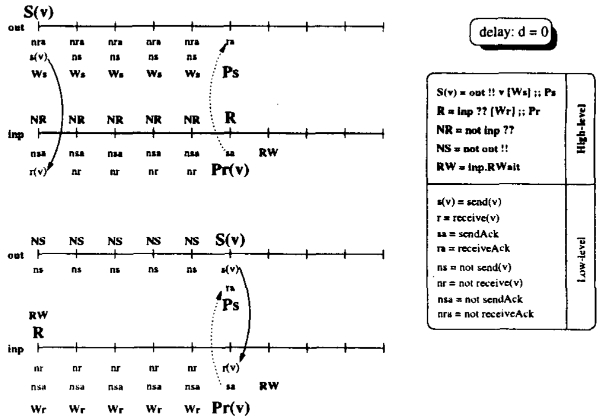

Figure 2: Examples of synchronous communications with no delay.

receive one value:

$$\text{rule}(inP.\text{receive}(k) \wedge inP.\text{receive}(v) \Rightarrow k = v)$$

This rule states: if at the same time instant two values are received on the same port these values have to be equal.

High-level Communication

The high-level layer introduces synchronous ports, the basic operators on these ports are: *Send* (!!) and *Receive* (??). They are quite easy to remember due to their similarity with CSP:

<outPort> !! <expr> [<whileExpr>];;<thenExpr> sends through output port <outPort> the value obtained by evaluating expression <const expr>. When the communication ends TILCO expression <thenExpr> is asserted. During the waiting the temporal expression <whileExpr> is asserted.

<inPort>?? [<whileExpr>];;<thenExpr> waits for a message (if not already arrived) from input port <inPort>. When the message arrives TILCO expression <thenExpr> is evaluated as a function of the value received. During the waiting the expression <whileExpr> is asserted.

Operators: $outP\,\overline{!!}$ and $inP\,\overline{??}$ have been introduced in order to specify that a process must not send a message on a port nor it must ask for a message. These conditions cannot be specified by using $\neg(inP\ !!\ v\ [P]\,;;W)$ which has a different meaning.

In TILCO, high-level synchronous operators are defined by using the low-level predicates as reported in the following. In Fig. 2, the only two cases of synchronous communication are reported: (i) the emitting process sends a message, and after the receiving process asserts that it wants to receive a message; (ii) the receiving process waits for a message and later the emitting process sends the message.

- operator **Send** emits the message and waits for an ACK. While waiting, wait formula W_s is asserted and no other messages are sent. When the ACK arrives, the formula P_s associated with the end of communication is asserted. In TILCO, the behavior of Send operator has been specified with the following axioms:

$$\begin{array}{l} \text{rule}((outP \,!!\, v\, [W_s]\,;;\, P_s) \Longrightarrow outP.\text{send}(v)\wedge \\ \quad \textbf{until}_0(outP.\text{receiveAck} \wedge P_s) \\ \quad\quad (\neg outP.\text{receiveAck} \wedge W_s)\wedge \\ \quad (outP.\text{receiveAck} \vee \\ \quad\; (\neg outP.\text{receiveAck} \wedge \\ \quad\quad \textbf{until}(outP.\text{receiveAck}) \\ \quad\quad\quad (\neg outP.\text{receiveAck} \wedge outP\,\overline{!!})))) \end{array}$$

$$\text{rule}(outP\,\overline{!!} \Longrightarrow \neg\exists k.outP.\text{send}(k))$$

the $\textbf{until}_0$ formula is used to state that P_s is true when the ACK is received and W_s is true until this time instant. The other part of the formula states that during the waiting for the ACK no message is sent.

- operator **Receive** has two possible situations. If a message, different than an acknowledged, was received in the past, then the ACK must be sent and the "end of communication" formula, P_r, is asserted with the value received. In the other case, a new message has to be waited asserting wait formula W_r. When a message is received (if any), the "end of communication" formula, P_r, is evaluated with the value received. In TILCO, the behavior of Receive has been specified with the following axioms:

$$\begin{array}{l} \text{rule}((inP\, ??\, [W_r]\,;;\, P_r) \wedge inP.\text{RValue}\ v \Longrightarrow \\ \quad inP.\text{sendAck} \wedge P_r(v)) \\ \text{rule}((inP\, ??\, [W_r]\,;;\, P_r) \wedge inP.\text{RWait} \Longrightarrow \\ \quad \textbf{until}_0(\exists k.inP.\text{receive}(k) \wedge inP.\text{sendAck} \wedge P_r(k)) \\ \quad\quad (\neg\exists k.inP.\text{receive}(k) \wedge W_r)\wedge \\ \quad (\exists k.inP.\text{receive}(k)\vee \\ \quad\; (\neg\exists k.inP.\text{receive}(k) \wedge \neg\, inP.\text{sendAck} \wedge \\ \quad\quad \textbf{until}(\exists k.inP.\text{receive}(k)) \\ \quad\quad\quad (\neg\exists k.inP.\text{receive}(k) \wedge inP\,\overline{??})))) \end{array}$$

$$\text{rule}(inP\,\overline{??} \Longrightarrow \neg\, inP.\text{sendAck})$$

$$\text{rule}(inP\, ??\, [W_r]\,;;\, P_r \wedge inP\,\overline{??} \Longrightarrow \bot)$$

where next formula indicates that there exists a pending v message:

$$\begin{array}{l} inP.\text{RValue}\ v\ = \\ \quad \textbf{since}'(inP.\text{receive}(v) \wedge \neg inP.\text{sendAck}) \\ \quad\quad (\neg inP.\text{sendAck}) \end{array}$$

and formula

$$inP.\text{RWait}\ = \neg\exists v.\ inP.\text{RValue}\ v$$

states the absence of a pending message to be elaborated (the current instant is not considered).

Fig.3 shows the more complex case in which there is a delay in transmission. In this case, two situations are possible. The first, when the distance from the Send and the subsequent Receive is greater than the delay, thus the message is received prior to the Receive action. The second and opposite case, when the Send action is performed after the Receive or before it having a distance lower than the delay.

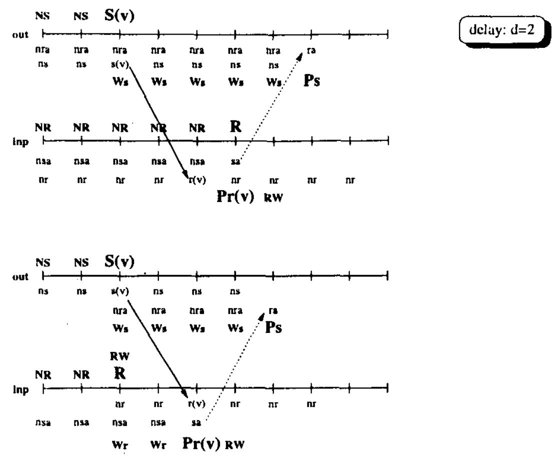

Figure 3: Examples of synchronous communications with delay.

CTILCO Communication Theorems

During the definition of CTILCO Communication Theorems many properties have been proved about the communication operators. This has been performed in order to validate the definitions of operators and to aid the construction of proofs involving these operators. The proofs were made by using a formalization of TILCO and CTILCO in Isabelle/HOL [4], please see http://www.dsi.unifi.it/~pbellini/tilco/ for details.

Theorems proved can be divided in two groups:

- theorems used to prove internal properties of a process. They substitute operators Send and Receive with their semantics;
- theorems used to prove properties involving connected processes.

In the first group, there are the theorems that can be used to eliminate a Send from the assumptions of a goal.

$$\dfrac{\dfrac{\begin{array}{c}[\vdash_t p.\text{send}(v)] \\ \vdots \\ \vdash_t p \,!!\, v\,[W_s]\,;;\, P_s \quad \vdash_t p.\text{receiveAck}?[0,+\infty)\end{array}}{\vdash_t \textbf{until}'_0\, P_s\, W_s}}{}$$

$$\dfrac{\vdash_t p \,!!\, v\,[W_s]\,;;\, P_s}{\vdash_t \textbf{until}_0\, P_s\, W_s}$$

The first theorem states that: if the process wants to send a message at time t and the message is sent receiving the ACK, then a time instant exists in which P_s is true. And, until that time instant, predicate W_s is

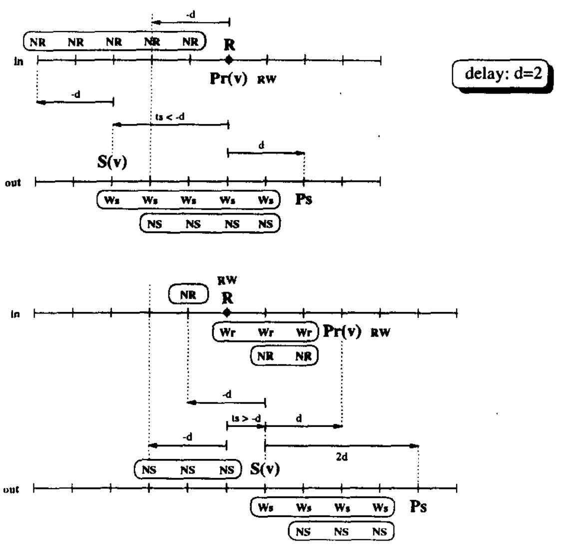

Figure 4: Theorems for synchronous communication

true. This theorem is used to substitute the Send with a strong until in the assumptions of the goal within the backward proofs of Isabelle.

The second theorem is similar to the previous without the assumption that *if a message is sent an ACK will be received*. In this weaker condition, the same condition with the weak-until has been derived.

For the Receive, similar theorems have been proved:

$$\frac{\vdash_t p\,??\,[W_r]\,;;\,P_r \quad \vdash_t \exists k.\ p.\text{receive}(k)\ ?\ [0,+\infty)}{\vdash_t \exists v.\ \textbf{until}'_0\, P_r(v)\ W_r}$$

$$\frac{\vdash_t p\,??\,[W_r]\,;;\,P_r}{\vdash_t \exists v.\ \textbf{until}_0\, P_r(v)\ W_r}$$

The first theorem of Receive states that, if a message will be received the operator Receive may be substituted with a strong until. The other theorem substitutes the Receive operator with a weak until without making any assumption about the message arrival.

In Fig. 4, the visual descriptions of the next two proved theorems are reported. The assumptions of the theorems are depicted upon the time axis while consequences are below. In theorems used to prove properties for connected processes, the *RWait* operator plays an important role. It summarizes the communication status:

$$\frac{\begin{array}{l} \mathcal{I} \models out \xrightarrow{d} in \\ \vdash_t in\,??\,[W_r]\,;;\,P_r \\ \vdash_{t+t_s} out\ !!\ v\ [W_s]\,;;\,P_s \\ \vdash_t in\,\overline{??}\,@[t_s-d,0) \\ t_s < -d \end{array}}{\begin{array}{l} \vdash_t P_r(v) \\ \vdash_{t+d} P_s \\ \vdash_t W_s@[t_s,d) \\ \vdash_t out\,\overline{!!}\,@(t_s,d) \\ \vdash_{t+1} in.\text{RWait} \end{array}}$$

This means in the premises: if two ports are connected with a delay d, a Receive is asserted at time t, and a Send is asserted t_s instants before the Receive. In the implication: the message is received at time t, P_s is true after d time instants, the wait formula of Send is true from the Send time instant to the end of communication time instant, and at $t+1$ *RWait* is true stating that no message is pending.

The following theorem covers the opposite case: in the absence of pending message, the Send is done after the Receive or within the delay.

$$\frac{\begin{array}{l} \vdash_t in.\text{RWait} \\ \mathcal{I} \models out \xrightarrow{d} in \\ \vdash_t in\,??\,[W_r]\,;;\,P_r \\ \vdash_{t+t_s} out\ !!\ v\ [W_s]\,;;\,P_s \\ \vdash_t in\,\overline{??}\,@[t_s-d,0) \\ \vdash_t out\,\overline{!!}\,@[-d,t_s) \\ -d \le t_s \end{array}}{\begin{array}{l} \vdash_{t+t_s+d} P_r(v) \\ \vdash_{t+t_s+2d} P_s \\ \vdash_t W_r@[0,t_s+d) \\ \vdash_{t+t_s} W_s@[0,2d) \\ \vdash_t in\,\overline{??}\,@(0,t_s+d) \\ \vdash_{t+t_s} in\,\overline{!!}\,@(0,2d) \\ \vdash_{t+t_s+d+1} in.\text{RWait} \end{array}}$$

Other theorems have been proved: some of them deal with the *RWait* operator that permits deducing that: if *RWait* is true for an input port and the connected emitting process is not sending, then *RWait* will remain true.

5 CTILCO SPECIFICATION VALIDATION

In order to validate a CTILCO specification, properties have to be proved by using the Isabelle/HOL theorem prover [4] with the formalization of TILCO and CTILCO. In that environment, theorems reported in the previous sections and many others make easier the properties proof either manually or automatically. In this environment, it should be noted that properties can be proved for the entire system as well as for single processes with generic parameters.

Proved properties are typically those of safeness (nothing bad will never happen) or liveness (something good will happen). Other properties that can be demonstrated are those aimed at validating the composition/decomposition of components. The evidence for the external properties of process is validated by means of its internal specification (decomposition), or vice versa (composition), depending on the approach used for building the system (bottom-up or top-down).

Since TILCO specifications can be executed by using a causal inferential engine, even a CTILCO specification can be executed. Obviously, not all the specifications can be executed, quantifications have to be done on finite domains, the specifications have to be deterministic and no generic parameters have to be present. However, the specification can be time incomplete, meaning that the system behavior can be partially specified for all the time instants.

6 AN EXAMPLE

In this section, an example to highlight the composition and reuse capabilities of CTILCO is presented together with some validations.

The system under specification is an abstraction of a train system that connects a set of stations. Every train passes from a fixed set of stations with a cyclic path. A train needs a bounded time duration to go from a station to the next. The train has to ask for permission to enter in a station. Once the permission is granted the train remains in the station for a constant time duration and then it leaves the station for the next one. Every station may have only one train inside at the same time. As an example, we consider the system shown in Fig.5.

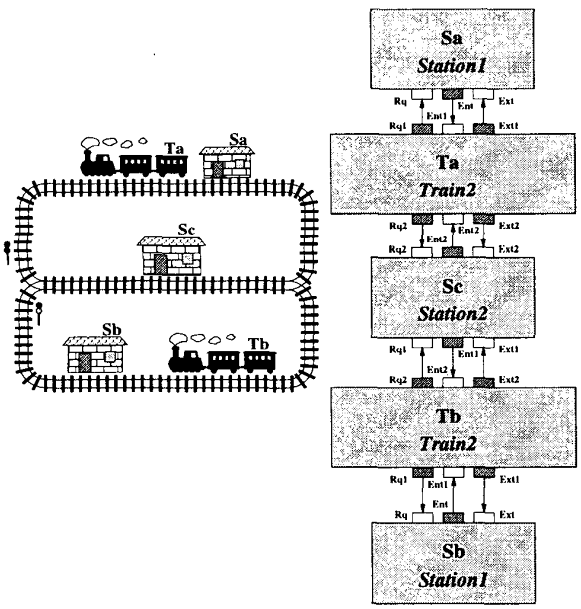

Figure 5: The railway system and its decomposition.

The system is decomposed with three types of processes:

- process *Station1* (**Sa** and **Sb**) manages the access of only one train.
- process *Station2* (**Sc**) manages the access of two trains.
- process *Train2* (**Ta** and **Tb**) models a train that reaches two stations.

Please note that the specification at system level consists only of the definition of process relationships and of a global start predicate.

In order to manage the access to a station, three ports are needed, one for the request to enter the station (Rq), another to give access to the station when the station is free (Ent), and the last one to notify to the station that the train has left the station (Ext).

Due to the limited space of the article the full specification of the system cannot be reported. In the following, many details are omitted. However, we think that the parts we chose to report highlight and explain the main aspects of CTILCO.

Process *Station1*

Process *Station*1 has three ports (Rq, Ent, Ext) to communicate with the train and three Boolean internal variables:

- *hasTrain* stating that the station has a train inside.
- *waitRq* that is true when the process has to wait for a request coming from the train.
- *waitExt* that is true when the process has to wait for the notification of exit of the train.

When the process starts, it has to wait for a request and before the starting the station has no train inside and no communication has been issued:

$$:process\_start \implies \; :waitRq \wedge (\neg :hasTrain)@(-\infty, 0)$$
$$:process\_start \implies \; (:Rq\,\overline{??} \wedge :Ent\,\overline{!!} \wedge :Ext\,\overline{??})@(-\infty, 0)$$

The general behavior is specified with the following formula:

$$\begin{array}{ll} :waitRq \implies & \\ & :Rq\,??\; [\neg :hasTrain \wedge :Ent\,\overline{!!} \wedge :Ext\,\overline{??}]\,;; \\ & :Ent\;!!\; enter\; [\neg :hasTrain \wedge :Rq\,\overline{??} \wedge :Ext\,\overline{??}]\,;; \\ & :Ext\,??\; [:hasTrain \wedge :Rq\,\overline{??} \wedge :Ent\,\overline{!!}]\,;; \\ & :waitRq \end{array}$$

This formula states that if the process must wait for a request a Receive is performed on port *Rq*. And, when a request is received the matching grant is immediately sent. During the waiting for the Receive on *Rq* port and the Send on *Ent* port, the train is not in the station ($\neg$: *hasTrain*). When the grant is received, the process waits for the exit notification. In the meanwhile, the train is in the station. When the notification is received, the *waitRq* variable is newly asserted in order to begin the waiting for a new request. It should be noted that, during the waiting for a certain port, the waiting predicate states that the process is not sending/receiving on the other ports. This is taken for granted in the following.

Process *Station2*

Process *Station2* has six ports (Rq1, Ent1, Ext1, Rq2, Ent2, Ext2) to communicate with the two trains and two Boolean variables: *hasTrain1* and *hasTrain2*.

These state that the station hosts train 1 or 2 inside, respectively.

A general requirement of *Station2* consists in allowing only one train to be inside the station at the same time instant:

$$(\neg(:\mathit{hasTrain1} \wedge :\mathit{hasTrain2}))@(-\infty, +\infty)$$

For the internal specification of process *Station2*, the following Boolean variables have been used:

- *free* states that the station is free;
- *waitRq1* and *waitRq2* – when one of these is true, the process must wait for an access request by train 1 or 2, respectively;
- *req1* and *req2* indicate that an access request has been received for train 1 or 2, respectively. It is kept true until the train has access to the station;
- *sendEnt1* and *sendEnt2* – when one of these is true, the process must send the enter notification to train 1 or 2 and wait for the exit notification.

According to the system specification, the following shortcuts were used:

$$\begin{aligned} A \Rightarrow\!\rhd B &\equiv A \Rightarrow B@[1,1] \\ \mathrm{inv}(A) &\equiv A \Leftrightarrow A@[-1,-1] \end{aligned}$$

The *free* process variable is defined as:

$$:\mathit{free} \Longleftrightarrow \neg :\mathit{hasTrain1} \wedge \neg :\mathit{hasTrain2}$$

When the process starts, it must wait for the requests, until a received request, *req1*/*req2*, is false and when the received request, *req1*/*req2*, becomes true:

$$\begin{aligned} &:\mathit{process\_start} && \Rightarrow :\mathit{waitRq1} \wedge :\mathit{waitRq2} \wedge :\mathit{free}@(-\infty, 0] \\ &:\mathit{process\_start} && \Rightarrow \\ & && (:\mathit{Rq1}\,\overline{??} \wedge :\mathit{Ent1}\,\overline{!!} \wedge :\mathit{Ext1}\,\overline{??})@(-\infty, 0) \\ &:\mathit{process\_start} && \Rightarrow \\ & && (:\mathit{Rq2}\,\overline{??} \wedge :\mathit{Ent2}\,\overline{!!} \wedge :\mathit{Ext2}\,\overline{??})@(-\infty, 0) \\ &:\mathit{waitRq1} && \Rightarrow \\ & && :\mathit{Rq1}\,??\,[\neg :\mathit{req1} \wedge :\mathit{Ent1}\,\overline{!!} \wedge :\mathit{Ext1}\,\overline{??}];; \\ & && :\mathit{req1} \wedge \neg :\mathit{hasTrain1} \wedge :\mathit{Ent1}\,\overline{!!} \wedge :\mathit{Ext1}\,\overline{??} \\ &:\mathit{waitRq2} && \Rightarrow \\ & && :\mathit{Rq2}\,??\,[\neg :\mathit{req2} \wedge :\mathit{Ent2}\,\overline{!!} \wedge :\mathit{Ext2}\,\overline{??}];; \\ & && :\mathit{req2} \wedge \neg :\mathit{hasTrain2} \wedge :\mathit{Ent2}\,\overline{!!} \wedge :\mathit{Ext2}\,\overline{??} \end{aligned}$$

When no request is received *hasTrain1* and/or *hasTrain2* remains stable (holding the same value):

$$\begin{aligned} \neg :\mathit{req1} &\Rightarrow\!\rhd \mathrm{inv}(:\mathit{hasTrain1}) \\ \neg :\mathit{req2} &\Rightarrow\!\rhd \mathrm{inv}(:\mathit{hasTrain2}) \end{aligned}$$

When the station is free and a request is received for a train but not for the other the enter notification is sent:

$$\begin{aligned} :\mathit{free} \wedge :\mathit{req1} \wedge \neg :\mathit{req2} &\Rightarrow\!\rhd :\mathit{sendEnt1} \\ :\mathit{free} \wedge :\mathit{req2} \wedge \neg :\mathit{req1} &\Rightarrow\!\rhd :\mathit{sendEnt2} \end{aligned}$$

When two requests are contemporaneously received, train 1 has the precedence:

$$:\mathit{free} \wedge :\mathit{req1} \wedge :\mathit{req2} \Rightarrow\!\rhd :\mathit{sendEnt1} \wedge :\mathit{req2} \wedge \neg :\mathit{hasTrain2}$$

When the station is not free and a request is received, the request is maintained active:

$$\begin{aligned} &\neg :\mathit{free} \wedge :\mathit{req1} \Rightarrow\!\rhd \\ &\quad :\mathit{req1} \wedge \neg :\mathit{hasTrain1} \wedge :\mathit{Rq1}\,\overline{??} \wedge :\mathit{Ent1}\,\overline{!!} \wedge :\mathit{Ext1}\,\overline{??} \\ &\neg :\mathit{free} \wedge :\mathit{req2} \Rightarrow\!\rhd \\ &\quad :\mathit{req2} \wedge \neg :\mathit{hasTrain2} \wedge :\mathit{Rq2}\,\overline{??} \wedge :\mathit{Ent2}\,\overline{!!} \wedge :\mathit{Ext2}\,\overline{??} \end{aligned}$$

When *sentEnt1*/(*sentEnt2*) is true, the enter notification is sent, and the exit notification is waited. In the meanwhile, *hasTrain1*/(*hasTrain2*) is true and no requests must be received. When the exit notification is received, *hasTrain1*/(*hasTrain2*) becomes false and, at the next instant, the process begins to wait again for a new request (to have the possibility to serve a pending request):

$$\begin{aligned} &:\mathit{sendEnt1} \Rightarrow \\ &\quad :\mathit{Ent1}\,!!\,\mathit{enter}\,[:\mathit{hasTrain1} \wedge \neg :\mathit{req1} \wedge :\mathit{Rq1}\,\overline{??} \wedge :\mathit{Ext1}\,\overline{??}];; \\ &\quad :\mathit{Ext1}\,??\,[:\mathit{hasTrain1} \wedge \neg :\mathit{req1} \wedge :\mathit{Rq1}\,\overline{??} \wedge :\mathit{Ent1}\,\overline{!!}];; \\ &\quad \neg :\mathit{hasTrain1} \wedge \neg :\mathit{req1} \wedge :\mathit{waitRq1}@[1,1] \end{aligned}$$

$$\begin{aligned} &:\mathit{sendEnt2} \Rightarrow \\ &\quad :\mathit{Ent2}\,!!\,\mathit{enter}\,[:\mathit{hasTrain2} \wedge \neg :\mathit{req2} \wedge :\mathit{Rq2}\,\overline{??} \wedge :\mathit{Ext2}\,\overline{??}];; \\ &\quad :\mathit{Ext2}\,??\,[:\mathit{hasTrain2} \wedge \neg :\mathit{req2} \wedge :\mathit{Rq2}\,\overline{??} \wedge :\mathit{Ent2}\,\overline{!!}];; \\ &\quad \neg :\mathit{hasTrain2} \wedge \neg :\mathit{req2} \wedge :\mathit{waitRq2}@[1,1] \end{aligned}$$

Process *Train2*

Process Train2 manages the access to two stations. It is decomposed by using two kinds of processes connected as depicted in Fig. 6. Processes of type *TrainAtStation* manage the access to a station. Processes of type *MinMaxDelay* are used to model the time spent by the train to reach the next station. A deterministic delay has to be fixed depending on the railway path length. Since the delay to pass on the path depends on the train velocity, its value is associated with the train aspects. Please see in the following.

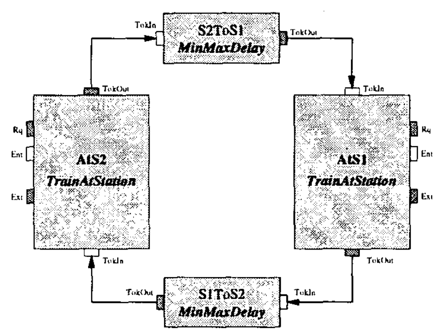

Figure 6: Train2 decomposition.

Ports *TokIn* and *TokOut* are used to sequentially activate the processes. When a message is received from port *TokIn*, the process is activated. And, when the process has finished, a message is sent via the $TokOut$ port. It is a sort of token passing mechanism.

Reusing the above-mentioned processes, more complex configurations can be defined and validated by using complex and general properties. For example, to demonstrate that the train will reach the station within a given time duration.

Process *TrainAtStation*
Process TrainAtStation manages the access to the station, the presence in the station and finally the departure from the station. It can be decomposed in three processes as shown in Fig. 7. Process $EnterStation$ manages the request of access to the station and the wait for enter notification. Process $MinMaxDelay$ (already presented for the upper level) is reused to model the time spent by the train in the station. Process $ExitStation$ states the exit from the station.

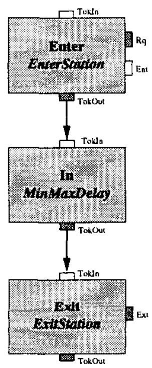

Figure 7: TrainAtStation decomposition.

The specifications of processes $EnterStation$ and $ExitStation$ are rather simple. For example, process $EnterStation$ must wait for the token, then it sends the access request, waits the enter notification, sends the token to the next and waits for the token again.

Process EnterStation
This process must wait for the token, then it sends the access request, waits for the enter notification, sends the token and waits for the token again.

$$
\begin{aligned}
&:process\_start \Rightarrow\\
&\quad :waitTok \wedge (:TokIn\,\overline{??} \wedge :Ent\,\overline{??} \wedge :TokOut\,\overline{!!})\,@(-\infty,0)\\[1ex]
&:waitTok \Rightarrow\\
&\quad :TokIn\,??\,[\neg :waiting \wedge :Rq\,\overline{!!} \wedge :Ent\,\overline{??} \wedge :TokOut\,\overline{!!}]\,;;\\
&\quad :Rq\,!!\,request\,[:waiting \wedge :TokIn\,\overline{??} \wedge :Ent\,\overline{??} \wedge :TokOut\,\overline{!!}]\,;;\\
&\quad :Ent\,??\,[:waiting \wedge :TokIn\,\overline{??} \wedge :Rq\,\overline{!!} \wedge :TokOut\,\overline{!!}]\,;;\\
&\quad :TokOut\,!!\,token\,[\neg :waiting \wedge :TokIn\,\overline{??} \wedge :Rq\,\overline{!!} \wedge :Ent\,\overline{??}]\,;;\\
&\quad :waitTok
\end{aligned}
$$

Process ExitStation
This process must wait for the token, then it sends the exit notification, sends the token and waits for the token again.

$$
\begin{aligned}
&:process\_start \Rightarrow\\
&\quad :waitTok \wedge (:TokIn\,\overline{??} \wedge :Ext\,\overline{!!} \wedge :TokOut\,\overline{!!})\,@(-\infty,0)\\[1ex]
&:waitTok \Rightarrow\\
&\quad :TokIn\,??\,[\neg :waiting \wedge :Ext\,\overline{!!} \wedge :TokOut\,\overline{!!}]\,;;\\
&\quad :Ext\,!!\,exit\,[:waiting \wedge :TokIn\,\overline{??} \wedge :TokOut\,\overline{!!}]\,;;\\
&\quad :TokOut\,!!\,token\,[\neg :waiting \wedge :TokIn\,\overline{??} \wedge :Ext\,\overline{!!}]\,;;\\
&\quad :waitTok.
\end{aligned}
$$

Process MinMaxDelay
This process has to wait for the token and to send the token to the next process after a delay between $MinDelay$ and $MaxDelay$. These values can be imposed according to Train velocity and path features.

$$
\begin{aligned}
&:process\_start \Rightarrow\\
&\quad :waitTok \wedge (:TokIn\,\overline{??} \wedge :TokOut\,\overline{!!})\,@(-\infty,0)\\[1ex]
&:waitTok \Rightarrow\\
&\quad :TokIn\,??\,[\neg :waiting \wedge :TokOut\,\overline{!!}]\,;;\\
&\quad (\neg :sendTok\,@[0,:MinDelay) \wedge\\
&\quad :sendTok\,?[:MinDelay,:MaxDelay] \wedge\\
&\quad \mathbf{until}_0 :sendTok\,(\neg :waiting \wedge :TokIn\,\overline{??} \wedge :TokOut\,\overline{!!}))\\[1ex]
&:sendTok \Rightarrow\\
&\qquad :TokOut\,!!\,token\,[\neg :waiting \wedge :TokIn\,\overline{??}]\,;;\\
&\qquad :waitTok
\end{aligned}
$$

Validation
Using the proven rules reported in the previous sections several properties have been proved.

The specification has been formally validated by using Isabelle theorem prover. In addition, the whole systems, as well as each single process, have been tested with the TILCO executor. In this case, several typical histories for inputs and outputs have been generated by using a signal editor, and formally verified.

For example, concerning process $Station2$, the external mutual exclusion requirement has been derived from the internal specification, this must be considered as a decomposition verification and is also a safeness property proof.

For example, for the train **Ta**, the following liveness property has been proved:

$$
\begin{aligned}
&\mathrm{up}(:Ta.inStation1) \Longrightarrow\\
&\qquad \mathrm{up}(:Ta.inStation1)\,?\,[min_{\mathbf{Ta}}, max_{\mathbf{Ta}}]
\end{aligned}
$$

That is, the distance between two successive time instants in which the train enters the first station is bounded. In the best case, the minimum time needed to across the path is:

$$
\begin{aligned}
min_{\mathbf{Ta}} &= Ta.timeInS1 + Ta.minS1ToS2 +\\
&\quad Ta.timeInS2 + Ta.minS2ToS1
\end{aligned}
$$

In the worst case, we have:

$$max_{\mathbf{Ta}} = Ta.timeInS1 + Ta.maxS1ToS2 + Tb.timeInS2 + Ta.timeInS2 + Ta.maxS2ToS1$$

Where: *timeInS1*, *timeInS2*, *maxS1ToS2*, *maxS2ToS1*, *minS1ToS2* and *minS2ToS1* are generic parameters of process *Train*2. These express the time spent in each station and the maximum/minimum time to pass from a station to the next. *inStation1* is a Boolean variable indicating that the train is in the first station of its path.

7 CONCLUSIONS

In this paper, CTILCO (Communicating TILCO) extension of the TILCO (Temporal Logic with compositional Operators) temporal logic has been presented. CTILCO is well suited for system composition/decomposition. It permits reusing component specifications within the same system or for the development of other systems. We think CTILCO language used for the specification is expressive, simple and concise with a limited "time to learn" since it has inherited conciseness from TILCO [5].

CTILCO has been formalized within Isabelle/HOL theorem prover [4]. Properties for the whole system as well as for a single process can be proved. This logical framework permits also the validation of system decomposition in terms of processes. The possibility to execute the specification is an important feature since well-known conditions can be quickly tested.

CTILCO has been successfully used for the formal specification of critical complex real-time systems. A prototype of the visual specification tool for CTILCO has been developed. The tool is called TOTS. It is based on the available theorem prover, the executor of TILCO specifications and the signal editor (please see http://www.dsi.unifi.it/~pbellini/tilco/).

ACKNOWLEDGEMENTS

The authors would like to thank all the members of CTILCO and TILCO projects. This work has been partially supported by the MURST Ex60% and COFIN.

REFERENCES

[1] G. Bucci, M. Campanai, and P. Nesi, "Tools for specifying real-time systems," *Journal of Real-Time Systems*, vol. 8, pp. 117–172, March 1995.

[2] A. D. Stoyenko, "The evolution and state-of-the-art of real-time languages," *Journal of Systems and Software*, pp. 61–84, April 1992.

[3] R. Mattolini and P. Nesi, "An interval logic for real-time system specification," *IEEE Transactions on Software Engineering, March-April*, 2001.

[4] L. C. Paulson, *Isabelle: A Generic Theorem Prover.* Springer Verlag LCNS 828, 1994.

[5] P. Bellini, R. Mattolini, and P. Nesi, "Temporal logics for real-time system specification," *ACM Comp. Surv.*, vol. 32, N.1, pp.12-42, March 2000.

[6] M. Felder and A. Morzenti, "Validating real-time systems by history-checking TRIO specifications," *ACM Trans. on Soft. Eng. And Method.*, vol. 3, pp. 308–339, Oct. 1994.

[7] A. Morzenti and P. SanPietro, "Object-oriented logical specification of time-critical systems," *ACM Trans. on Soft. Eng. and Method.*, vol. 3, pp. 56–98, Jan. 1994.

[8] P. Bellini, M. Bruno, and P. Nesi, "Verification of external specifications of reactive systems," *IEEE Trans. on System Man and Cybern.*, in press, 2000-2001.

[9] P. Bellini, M. Bruno, and P. Nesi, "Verification criteria for a compositional model for reactive systems," *Proc. of the IEEE Intern. Conf. on Complex Computer Systems*, Sept. 11-15 2000.

[10] R. Koymans, *Specifying Message Passing and Time-Critical Systems with Temporal Logic.* No. 651, LNCS, Springer-Verlag, 1992.

[11] A. Coen-Porosini, C. Ghezzi, and R. A. Kemmerer, "Specification of realtime systems using ASTRAL," *IEEE Trans. on Soft. Eng.*, vol. 23, pp. 572–589, Sept. 1997.

[12] C. Ghezzi, D. Mandrioli, and A. Morzenti, "TRIO, a logic language for executable specifications of real-time systems," *Journal of Systems and Software*, vol. 12, pp. 107–123, May 1990.

[13] M. Felder, D. Mandrioli, and A. Morzenti, "Proving properties of real-time systems through logical specifications and Petri net models," tech. rep., Politecnico di Milano, 91-072, Milano, Italy, 1991.

[14] M. Ben-Ari, *Mathematical Logic for Computer Science.* New York: Prentice Hall, 1993.

[15] Z. Chaochen, "Specifying communicating systems with temporal logic," in *Proc. of the Intern. Conf. on Temporal Logic in Spec.*, (B. Banieqbal, H. Barringer, and P. Pnueli, eds.), (Altrincham, UK), pp. 304–323, LNCS 398, 8-10 April 1987.

[16] R. Koymans, "Specifying message passing systems requires extending temporal logic," in *Proc. of Temporal Logic Specification Instanted Kingdom, (B. Banieqbal, H. Barringer, A. Pnueli, eds)*, pp. 213–223, LNCS 398, April 1987.

[17] G. M. Reed and A. W. Roscoe, "A timed model for communicating sequential processes," in *Proc. of ICALP'86, LNCS N. 226*, pp. 314–323, 1986.