.30

Validating Component Integration with C-TILCO, a Case Study

P. Bellini P. Nesi D. Rogai [1]

Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
Firenze, Italy
Via S. Marta 3, 50139 Firenze, Italy,
tel.: +39-055-4796523, fax.:+39-055-4796363

Abstract

Temporal logics are typically used to specify and verify properties and thus requirements, to describe the system and prove that such a formalization meets the expected behavior. In this paper, C-TILCO temporal logic is considered. C-TILCO is an extension of TILCO temporal logic which provides compositional and communication primitives. TILCO specifications of system behavior can be directly used as implementations since they can be directly executed in real-time by using the TILCO executor. On the other hand, the validation phase remains of high relevance in the system deployment. The validation phase can be applied to both the single components and their integration in order to validate the entire solution. The validation requires dedicated tools to easily work out a considerable amount of proofs in reasonable time. To this end, in this article, a case study about specification of a communicating system is presented together with some important property proofs taken from the validation phase.

Keywords: formal specification language, first order logic, temporal interval logic, real-time systems, temporal operators, theorem provers, validation, components integration, communicating system.

1 Introduction

The specification of real-time systems implies the adoption of a specific formal model for the definition of temporal constraints among events and actions [6],

[1] This work has been partially supported by the Italian Ministry of University and Research within the COFIN 2001 project "Quack: a platform for the quality of new generation integrated embedded systems"

[1], [7]. These formal methods are typically used for describing properties of invariance, precedence amongst events, periodicity, liveness and safety conditions, etc. For this purpose, several temporal logics have been used [1], or timed state machine, Petri Nets, etc. When the system under specification is not trivial its specification needs to be performed by decomposing the problems in smaller segments or components devoted to solve specific identified sub-problems [10], with the aim of obtain the whole system for composition.

The adoption of compositional models for the systems specification has to be supported by formal methods for the verification of components and composed systems [9]. To this end, several approached have been used mainly on the formal description of the component interface and behavior and supporting the compositional verification dividing the behaviour in external and internal [10], [2]. In most cases, the external description of the component is only an abstract description of the internal specification. Thus the validation based on only the external description may results to be partial with respect to an exhaustive validation based on the full specification. The verification is performed by using model checking approaches on the operational description of the system, in others the verification is performed by validating the formal composition and thus the compositional behavior.

The approach proposed in this paper is based on TILCO (Temporal Logic with Compositional Operators) and its compositional version called C-TILCO. Please note that C in TILCO acronym is referred to the composition of temporal constraints. TILCO presents a uniform model for time from past to future and unique operators for stating facts and events along the time axis [8], together with extended temporal operators (TILCO-X) [5] and process communication support (C-TILCO) [4]. The process communication of C-TILCO allows to specify a complex system by its decomposition in several process and to model inter-process communication between them. TILCO language can be directly executed, such executability consists in using the specification as an implementation of the real-time system, thus allowing (in each time instant) the *on-line* generation of system outputs on the basis of current inputs (including those concerning communication) and internal state. In this sense, TILCO-Executor, presented in [3], is capable of executing a fragment of TILCO specifications.

This paper presents a case study in which C-TILCO has been used for the specification and validation. C-TILCO permits the description of the (i) internal properties of each process involved in the architecture and (ii) the external properties suitable for a correct interaction of the components.

This paper is organized as follows, in Section 2, a short overview of C-TILCO and TILCO-X, as presented in [4], is reported. In Section 3, a case study specified using C-TILCO is presented; Section 4 performs some validation step for the specified system. Conclusions are drawn in Section 5.

2 C-TILCO Overview

A system specification in C-TILCO is a hierarchy of communicating process components whose specifications are written in TILCO. TILCO is a logic language which can be used to specify temporal constraints in either a qualitative or a quantitative way [5]; the meaning of a TILCO formula is given with respect to current time. Time is discrete and linear and the temporal domain is the set of integers $\mathbb{Z}$. The minimum time interval corresponds to one instant, the current time instant is represented by 0 and positive (negative) numbers represent future (past) time instants. The basic entity in TILCO is temporal interval, the boundaries of which can be either included or excluded by using the usual notation with squared, ("[", "]") or round ("(", ")") brackets, respectively.
The basic TILCO *temporal operators* are:

- "**@**", universal quantification ($\forall$) over a temporal interval;
- "**?**", existential quantification ($\exists$) over a temporal interval;
- "**until**", to express that either a predicate will always be true in the future, or it will be true until another predicate will become true;
- "**since**", to express that either a predicate has always been true in the past, or it has been true since another predicate has become true.

TILCO has been extend to provide some more expressive operators creating the TILCO-X language [5]. The dynamic intervals allow to define an interval using boundaries which are dependent on TILCO expression. For example, $A@(0, +B]$ asserts that A is *true* from the next time instant to the the instant when B occurs for the first time from now; such instant is included in the expression. Similarly a boundary like $-B$ refers to the last time B occurred in the past.

In C-TILCO many instances of the same process component specification can be arranged in the global architecture. Processes can have some parameters and every instance has distinct values. The communication between processes is based on a CSP like 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.

The basic element for C-TILCO is a *process component* which is represented by two views:

(i) the *external view* that describes the input/output behavior of the process;
(ii) 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 C-TILCO 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 parameter; a set of external TILCO *formulæ* that describe the external process behavior by means of the messages exchanged and constraints on the external variables. C-TILCO is *internally* characterized by: a set of C-TILCO *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). 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 access process components. Since many instances of the same process can be present in the system, its specification is valid for all of them. By means of *colon* operator applied to process components, process and local variables it can be easily distinguished in the specification.

Since in TILCO the time axis is infinite in both directions there is not a time instant that can be regarded as the *start* time instant of the execution process. In the system specification, it is natural to consider a reference time instant in which the process starts its work: before that time all the signals are 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.

Communication primitives

C-TILCO provides synchronous ports, the basic operators on these ports are: *Send* (!!) and *Receive* (??):

`<outPort>` !! `<expr>` [`<whileExpr>`] ; ; `<thenExpr>` sends through output port `<outPort>` the value obtained by evaluating expression `<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 to specify that a process has not to send a message on a port or that the process has not to ask for

a message. These conditions cannot be specified by using $\neg(inP \;!!\; v\; [P]\,;;W)$ which has a different meaning.

3 Using C-TILCO to specify a communication protocol

The following case study is presented to show how C-TILCO can aid in the formal verification of a component-based architecture.

The system under specification is a communication system, based on a well know protocol. The communication system is composed of several nodes which are connected in a ring structure (see Fig 1).

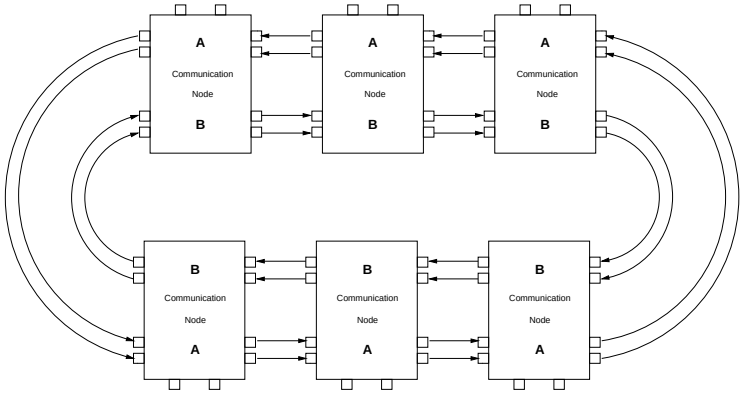

Fig. 1. The communication system

This communication basically aims to be robust against a single node failure and to distribute the communication priority in a uniform manner on the nodes. Two concentric rings are provided: the main ring where information is passed along the elements, and the backup ring which recovers a main ring failure connection and allows to perform the communication until the system is restored. The main ring is signed with **A** letter and the backup with **B**. Input and output port for each ring is required. The communication system is the result of the proper connection of the nodes. A fixed information (called *token*) is received and retransmitted by every node to the adjacent. If a node needs to transmit a data, it waits for the token then transmits the data to the adjacent keeping the token and, when the transmitted data returns back to the sender, it releases the token. A node recognizes a fault communication after a time-out and redirects the communication in the backup ring which works in the opposite direction of the main ring.

A part from the normal communication mode in Fig. 2 a possible scenario of the data flow is represented: a broken connection which has been recovered by using the backup ring.

The system is realized in terms of communication nodes; over each one a higher level communication interface is typically connected. These nodes perform only simple data communication and protocol management. The token is considered as boolean and is transmitted over a dedicated channel

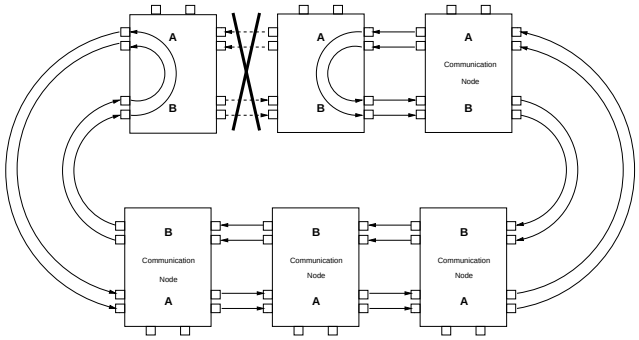

Fig. 2. The backup complete the cycle communication when a channel is broken

and the message is treated as structured type that includes the origin and the destination node ID.
The data communication structure is as follows:

```
struct DATAPACK {
    int srcID;
    int dstID;
    char data[MAX_LENGHT];
}
```

The communication node provides two ports for each direction of the main ring and for the backup. Two additional ports are present to communicate with the higher layer.

3.1 System requirements

The node basically performs the following operations:

- it waits for the token;
- when it has received the token, it can transmit data on the ring without releasing the token, otherwise it must retransmit the token along the node chain;
- if the adjacent node does not reply it has to redirect the same information (token or data) on the back up ring which works in the opposite direction;
- when the transmitted data comes back from the ring the node releases the token to allow the other to transmit their own data.
- any data received on the backup channel has to be retransmitted without any check, but just redirecting it on the main ring if the adjacent node does not replay.

3.2 Communication Node

The node must ensure the communication and a particular attention should be given to the token passing. When a wrong behavior is observed on a node the basic token transmission has to be granted to keep alive the communication. For this reason, the node system is decomposed in sub-systems. The decomposition has been performed by exploiting the capability of C-TILCO in decomposing complex system and validating them separately and compounded.

The decomposed communication node is shown in Fig. 3 and presents three sub-systems:

- **Communication Manager**: it grabs the token and performs the communication protocol on the main ring;
- **Token Repeater**: it repeats the token to the next node when the token reaches the communication node; it is used by the backup ring;
- **Data Repeater**: analogue of the Token Repeater, while it handles data.

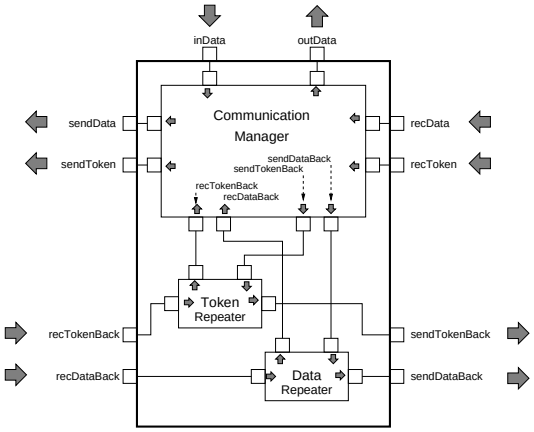

Fig. 3. The decomposition of the communication node

The main component is the Communication Manager whose specification is comprised of several parts to better understand their meaning.

The following TILCO-X specification expresses the basic token passing inside the communication manager.

$$
\begin{aligned}
&:readyForToken \Rightarrow\!\!> \\
&\qquad :recToken \,??\, [\neg :readyForToken \wedge \neg :transmitAnything]\,;; \\
&\qquad (\neg :readyForToken \wedge \\
&\qquad (:dataBufferEmpty \Rightarrow (:transmitToken \wedge \neg \exists any.\, :transmitData(any))\,@\,[0, +\, :readyForToken)) \wedge \\
&\qquad (:getDataBuffer(d) \Rightarrow (:transmitData(d) \wedge \neg :transmitToken)\,@\,[0, +\, :readyForToken)))
\end{aligned}
$$

A "ready" predicate, which is initialized when the node process starts, puts the system in a wait status for receiving. While the node is waiting for the token it cannot transmit anything (either token or data).

When a token comes two different choices are available: to re-transmit the token or to transmit the data buffered from the *inData* port.

About the data transmission the requirements specify that a new data is received from the higher layer input port and is stored in a specific buffer until the token is grabbed by the node. An asynchronous communication in this direction is used to avoid any unnecessary delay time in the ring communication: in this way the token is not grabbed without having a data to transmit. With a non-empty buffer and the grabbed token the transmission can start as it is specified in the following formulas.

A simple rule, which provides a token (or data) redirection on the backup ring, when the main fails, has to be considered. The *sendingToken* (or *sendingData*) predicate asserts that a transmission attempt is on (until the node receive the *ACK* signal from the adjacent after a successful transmission). Thus the time-out condition can be evaluated as:

$$\neg : reset \,@\, (-((: sendingToken \vee : sendingData) \,@\, (-10, 0]), 0) \iff : brokenChannel$$

The transmission of a token which occurs after a received token is specified by the following formulæ

$$\begin{aligned}
: transmitToken \vee \exists any. : transmitData(any) &\Longrightarrow : transmitAnything \\
up(: transmitToken \wedge \neg : brokenChannel) &\Rightarrow\!\!> \\
&\quad \neg : readyForToken_B \,@\, [0, + : brokenChannel] \wedge \\
&\quad : sendToken \;!!\; [sendingToken \wedge \neg : readyForToken_A] \,;; \\
&\quad \neg : sendingToken \,@\, [0, +(: transmitToken \vee : repeatToken)) \wedge \\
&\quad : readyForToken_A \wedge \neg : transmitAnything \\
up(: transmitToken \wedge : brokenChannel) &\Rightarrow\!\!> \\
&\quad : sendTokenBack \;!!\; [\neg : readyForToken_B] \,;; \\
&\quad : readyForToken_B \wedge \neg : transmitAnything \\
: readyForToken_A \vee : readyForToken_B &\iff : readyForToken
\end{aligned}$$

The following expressions specify the behavior after the activation of a *transmitData(d)*. Thus the node has to transmit the data d to the adjacent (it is very similar to the token transmission which has been previously described).

$$\begin{aligned}
up(: transmitData(d) \wedge \neg : brokenChannel) &\Rightarrow\!\!> \\
&\quad \neg : readyForToken_B \,@\, [0, + : brokenChannel] \wedge \\
&\quad : sendData \;!!\; d \; [sendingData \wedge \neg : readyForToken_A] \,;; \\
&\quad \neg : sendingData \,@\, [0, +\exists next.(: transmitData(next) \vee : repeatData(next))) \wedge \\
&\quad : readyForToken_A \wedge \neg : transmitAnything \\
up(: transmitData(d) \wedge : brokenChannel) &\Rightarrow\!\!> \\
&\quad : sendDataBack \;!!\; d \; [\neg : readyForToken_B] \,;; \\
&\quad : readyForToken_B \wedge \neg : transmitAnything
\end{aligned}$$

In these formulas, a failed attempt of communication on a broken channel is recovered using the backup ring, if both channels are broken the node cannot communicate anymore. Two different predicates can determine the ready state after a successful transmission ($readyForToken_A$, $readyForToken_B$) on one of the available channels. It has to be noticed that a broken channel will freeze the port on the send state waiting forever for the remote synchronization.

The management of incoming data is specified with a similar structure. In this case, the received data d has to be examined to decide which operation has to take place. The data may

- be addressed to the node: this data has to be sent to the higher level of the communication (the *outData* port) and repeated along the node ring;
- be originated from the node: this data is returned back from the ring the token it has been grabbed has to be released;
- has no relation with the node: it has to be repeated by sending it again along the ring.

The following expressions specify the above aspects where *arrivedData(d)* is a predicate which asserts that a data, with value d, has been received from the *recData* port. On the basis of $d.srcID$ and $d.dstID$ the appropriate predicate *repeatData* or *repeatToken* is asserted.

$$
\begin{aligned}
&:readyForData \Rightarrow\!\!\!> \\
&\quad :recData \,??\, [\neg :readyForData \wedge \neg :repeatAnything]\,;; \\
&\quad \lambda d.\neg :readyForData \wedge :arrivedData(d) \\
&:arrivedData(d) \Longrightarrow (\neg\exists any.\, :arrivedData(any))\ @\ (0, +\, :readyForData] \\
&:arrivedData(d) \wedge d.srcID \neq :nodeID \Rightarrow (:repeatData(d) \wedge \neg :repeatToken)\ @\ [0, +\, :readyForData) \\
&:arrivedData(d) \wedge d.srcID = :nodeID \Rightarrow (:repeatToken \wedge \neg\exists any.\, :repeatData(any))\ @\ [0, +\, :readyForData) \\
&:arrivedData(d) \wedge d.dstID = :nodeID \Longrightarrow :outData\ !!\ d\ [true]\,;; true \\
&:resetBuffer \iff \exists d.(:arrivedData(d) \wedge d.srcID = :nodeID)
\end{aligned}
$$

The previous expressions specify the different predicates which are activated on the basis of the received data. The *repeatToken* and *repeatData* predicates work in a similar manner of the previously presented *transmitToken* and *transmitData* predicates. These "repeat" predicates have been kept separated because at the end of the transmission different predicates have to be activated. In fact, after a *repeatToken* the system asserts *readyForData*,while *transmitToken* leads to the activation of the *readyForToken* predicate.

$$
\begin{aligned}
&:repeatToken \vee \exists any.\, :repeatData(any) \Longrightarrow :repeatAnything \\
&\quad up(:repeatData(d) \wedge \\
&\quad \neg :brokenChannel) \Rightarrow\!\!\!> \\
&\qquad \neg :readyForData_B\ @\ [0, +\, :brokenChannel] \wedge \\
&\qquad sendData\ !!\ d\ [:sendingData \wedge \neg :readyForData_A]\,;; \\
&\qquad :sendingData\ @\ [0, +\exists d.(transmitData(d) \vee repeatData(d))) \wedge \\
&\qquad :readyForData_A \wedge \neg repeatAnything \\
&\quad up(:repeatData(d) \wedge :brokenChannel) \Rightarrow\!\!\!> \\
&\qquad sendDataBack\ !!\ d\ [\neg :readyForData_B]\,;; \\
&\qquad :readyForData_B \wedge \neg repeatAnything \\
&\quad up(:repeatToken \wedge \neg :brokenChannel) \Rightarrow\!\!\!> \\
&\qquad \neg :readyForData_B\ @\ [0, +\, :brokenChannel] \wedge \\
&\qquad sendToken\ !!\ [:sendingToken \wedge \neg :readyForData_A]\,;; \\
&\qquad :sendingToken\ @\ [0, +(:transmitToken| :repeatToken)) \wedge \\
&\qquad :readyForData_A \wedge \neg repeatAnything \\
&\quad up(:repeatToken \wedge :brokenChannel) \Rightarrow\!\!\!> \\
&\qquad sendTokenBack\ !!\ [\neg :readyForData_A]\,;; \\
&\qquad :readyForData_A \wedge \neg repeatAnything \\
&:readyForData_A \vee :readyForData_B \iff :readyForData
\end{aligned}
$$

In the following the data buffer and its relation with the *inData* port have been expressed. The *getDataBuffer(d)* predicate is asserted only when a data with valude d is stored in the buffer. $setDataBuffer(d)$ and $resetBuffer$ are provided to set and reset the value in the buffer, respectively. The *dataBufferEmpty* predicate denies the *getBuffer(d)* for every d: no value is stored in the

buffer.

$$
\begin{aligned}
:up(DataBufferEmpty) &\implies \\
&:inData \,??\, [:dataBufferEmpty]\,;; \\
&setDataBuffer(d) \\
:setDataBuffer(d) &\vee \\
\mathbf{since}(:setDataBuffer(d), \neg :resetBuffer) &\iff getDataBuffer(d) \\
:getDataBuffer(d) &\iff \neg dataBufferEmpty
\end{aligned}
$$

Similar expressions are needed to perform the simple replication (without main and back attempt) of data and token which are received from *recDataBack* and *recTokenBack*.

The process initialization puts the system in a waiting status all the receiving ports. Therefore in order to complete the specification, specific predicates have been introduced to assert that not any data or token is sent until the apposite predicate is activated. It follows the initialization expression.

$$
\begin{aligned}
:process\_start \implies & \\
& :readyForToken \wedge :readyForData \wedge \\
& \neg :transmitAnything \wedge \neg :repeatAnything \wedge \\
& :readyForTokenBack \wedge :readyForDataBack \wedge \\
& \neg :sendingToken \,@\, (-\infty, +(:transmitToken \vee :repeatToken)) \wedge \\
& \neg :sendingData \,@\, (-\infty, +\exists d.(:transmitData(d) \vee :repeatData(d)))
\end{aligned}
$$

The other sub-components have not been described in this paper; the specification of these parts usually reuse formulæ from the communication manager and introduces different features in a less complex behavior to help component reuse.

4 Validating the specification

In order to prove properties at level of single process and for the whole system the inference rules defined for C-TILCO and for TILCO-X could be used. The validation reported here is only a small part of the whole validation.

To prove properties for a single process the following two theorems has to be considered:

$$
\frac{\vdash_t p \,!!\, v\, [W_s]\,;; P_s}{\vdash_t \mathbf{until_0}\, P_s\, W_s} \qquad \frac{\vdash_t p \,??\, [W_r]\,;; P_r}{\vdash_t \exists v.\, \mathbf{until_0}\, P_r(v)\, W_r}
$$

These two theorems allow to substitute a *Send/Receive* operator with a weak until operator in the premises of a goal.

In the theorems used to prove properties for connected processes, the *RWait* operator plays an important role. It summarizes the communication status saying if a message has been received in the past and it has not been

acknowledged. The two main theorems are the following:

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

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

That means in the premises of the left-side theorem: 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 on the left-side: 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 theorem on the right-side covers the opposite case: in the absence of pending message, the Send is done after the Receive or within the delay.

Other theorems have been proved: one concerns 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.

At level of a single process the Communication Node safeness properties about token/data transmission can be proved. The $readyForToken$ predicate cannot be asserted toghether with the transmission attempt of token/data ($transmitToken$ and $transmitData$). This can be stated as:

$$:process\_start \Longrightarrow (:readyForToken \Rightarrow (\neg :transmitToken \wedge \neg \exists any. :transmitData(any)))@[0, +\infty)$$

it can be transformed in:

$$:process\_start \Longrightarrow :readyForToken \wedge \neg :transmitAnything$$

$$:readyForToken \wedge \neg :transmitAnything \Longrightarrow \neg :transmitAnything\,@\,[+ :readyForToken]$$

which can be proved using the specification.

Moreover, from this result a safeness property can be derived: that token and data cannot be transmitted simultaneously ($\neg B\,@\,[0, +\infty)$). This can be stated as:

$$:process\_start \Longrightarrow \neg(:transmitToken \wedge \exists any. :transmitData(any))@(0, +\infty)$$

In order to prove that this critical condition cannot be reached, an initial induction-like strategy has been adopted to branch the main goal:

$$:process\_start \Longrightarrow :readyForToken$$

$$:readyForToken \Longrightarrow (\neg BAD)\,@\,(0, + :readyForToken]$$

where $BAD =: transmitToken \wedge \exists any. : transmitData(any)$ asserts the bad condition. The first part is trivially derived from the specification. The second

implication needs a further step to separate the singular point at the end of the dynamic interval. This can be written as follows:

$$:readyForToken \implies (\neg BAD)\,@\,(0,+:readyForToken) \wedge$$
$$:readyForToken \implies (\neg BAD)\,@\,[+:readyForToken]$$

The second sub-goal is directly solved by the safeness condition $:readyForToken \Rightarrow \neg :transmitAnything$. While the first subgoal can be proved by looking at the specification that rules the system when it waits for the token and after it is arrived. From the expression written at page 7 the following proof status can be achieved:

$$\frac{\begin{array}{ll} & \vdash_{t+1} :recToken\,??\,[\neg :readyForToken \wedge \neg :transmitAnything]\,;; \\ & \quad (\neg :readyForToken \wedge \\ & \quad (:dataBufferEmpty \Rightarrow \\ & \quad (:transmitToken \wedge \neg\exists any.\, :transmitData(any))\,@\,[0,+:readyForToken)) \wedge \\ & \quad (:getDataBuffer(d) \Rightarrow \\ \vdash_t :readyForToken & \quad (:transmitData(d) \wedge \neg :transmitToken)\,@\,[0,+:readyForToken))) \end{array}}{\vdash_t (\neg BAD)\,@\,(0,+:readyForToken)}$$

By using the communication theorem allowing the substitution of a Send/Receive operator with an $\mathbf{until_0}$ operator the goal can be transformed in this way:

$$\frac{\begin{array}{ll} & \vdash_{t+1} \mathbf{until_0}(\neg :readyForToken \wedge \\ & \quad (:dataBufferEmpty \Rightarrow \\ & \quad (:transmitToken \wedge \neg\exists any.\, :transmitData(any))\,@\,[0,+:readyForToken)) \wedge \\ & \quad (:getDataBuffer(d) \Rightarrow \\ & \quad (:transmitData(d) \wedge \neg :transmitToken)\,@\,[0,+:readyForToken))) \\ \vdash_t :readyForToken & \quad (\neg :readyForToken \wedge \neg :transmitAnything) \end{array}}{\vdash_t (\neg BAD)\,@\,(0,+:readyForToken)}$$

Using the rules for $\mathbf{until_0}$ two subgoals are generated:

$$\frac{\vdash_t :readyForToken \quad \vdash_{t+1} (\neg :readyForToken \wedge \neg :transmitAnything)\,@\,[0,+\infty)}{\vdash_t (\neg BAD)\,@\,(0,+:readyForToken)}$$

$$\frac{\begin{array}{ll} \vdash_{t+1+x} & (\neg :readyForToken \wedge \\ & \quad (:dataBufferEmpty \Rightarrow \\ & \quad (:transmitToken \wedge \neg\exists any.\, :transmitData(any))\,@\,[0,+:readyForToken)) \wedge \\ & \quad (:getDataBuffer(d) \Rightarrow \\ & \quad (:transmitData(d) \wedge \neg :transmitToken)\,@\,[0,+:readyForToken))) \\ \vdash_t :readyForToken \vdash_{t+1} & \quad (\neg :readyForToken \wedge \neg :transmitAnything)\,@\,[0,x) \end{array}}{\vdash_t (\neg BAD)\,@\,(0,+:readyForToken)}$$

The first goal is proved considering that $\neg :transmitAnything \Rightarrow \neg BAD$ holds. The second goal is proved using the specification and considering the two possible cases: that there it is a pending data in the buffer or not.

The specification must be validated against the integration of the component. The property which grants the token passing, ensures a balanced communication priority for all the nodes of the ring. The token passing is quick and, on the basis of a small delay of port communication, is performed inside a single time sample. An integration property asserts that if the communication channel is broken a token is passed along the backup ring.

Considering two adjacent nodes (n_1, n_2) it has been supposed that n_1 is attempting to transmit the token ($n_1.sendingToken$) and n_2 is waiting for a token. The waiting status of n_2 can be expressed as: $n_1.sendToken\,\overline{!!}\ @\,[-n_2.readyForToken, 0)$ Moreover it must be asserted that $\neg n_1.brokenChannel \wedge n_2.brokenChannel$; $n_2.bufferEmpty$ is true, meaning that on n_2 no message has to be sent.

The synchronization between the connected ports $n_1.sendToken$ and $n_2.recToken$ activates $n_2.transmitToken$; the broken channel condition enables the trasmission of the token on the $n_2.sendTokenBack$ port. The connected port $n_1.recTokenBack$, which was waiting for a synchronization can propagate the token in the backup ring.

It should be noted that in the specification there have been omitted all the $port\,\overline{!!}$ and $port\,\overline{??}$ operators. They are needed to complete the specification and to use the communication theorems previously described.

5 Conclusions and Future Work

The verification and validation is very important for the system which are build on the basis of components. C-TILCO allows the specification of the whole system in sub-components and the primitives to control communication among them. After a proper formalization of the component-based architecture integration tests can be performed by means of properties proofs. This validation requires dedicated tools to easily work out a considerable amount of proofs. To this end, an implementation of TILCO temporal logic (including TILCO-X and C-TILCO features) in the PVS theorem prover is in progress.

In addition the core specification of component can be easily turned into the implementation of the behavior since TILCO specification can be directly executed.

References

[1] P. Bellini, R. Mattolini and P. Nesi, *Temporal logics for real-time system specification*, ACM Computing Surveys **31** (2000).

[2] P. Bellini, M. A. Bruno, P. Nesi, *Verification of External Specifications of Reactive Systems*, IEEE Trans. on Systems Man and Cybernetics - Part A, **30-6**(2000), pp. 692–709.

[3] P. Bellini, A. Giotti and P. Nesi, *Execution of tilco temporal logic specifications*, Proc. of the 8th IEEE Intl. Conference on Engineering of Complex Computer Systems, Greenbelt, (Maryland, USA) (2002).

[4] P. Bellini and P. Nesi, *Communicating TILCO: a model for real-time system specification*, in: *Proc of the 7th "IEEE International Conference on Engeneering of Complex Computer Systems",ICECCS'01.*

[5] P. Bellini and P. Nesi, *TILCO-X: an extension of TILCO temporal logic*, in: *Proc. of the 7th "IEEE International Conference on Engeneering of Complex Computer Systems",ICECCS'01.*

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

[7] A. Coen-Porisini, C. Ghezzi and R. Kemmerer, *Specification of real-time systems using ASTRAL*, IEEE Trans. on Soft. Eng., 23 (1997) 572-598

[8] R. Mattolini and P. Nesi, *An interval logic for real-time system specification*, IEEE Trans. on Soft. Eng., March-April (2001).

[9] G. Leavens and M. Sitaraman. *Foundations of component-based systems.* Cambridge University Press, (2000).

[10] L. Mariani, *A fault taxonomy for component-based software*, proc. of International Workshop on Test and Analysis of Components Based Systems, TACOS2003, (M. Pezze, Ed.), Warszawa, April, 2003.