Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 116 (2005) 241–252
www.elsevier.com/locate/entcs

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 to prove whether such 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. The validation phase can be applied to both the single components and their integration in order to validate the entire solution. 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 framework of the COFIN 2001 project "Quack: a platform for the quality of new generation integrated embedded systems"

doi:10.1016/j.entcs.2004.02.080

[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]. 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 decomposing it in several processes and it allows 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], can execute a fragment of TILCO specifications.

This paper presents a case study where 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.

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 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 becomes true;
- "**since**", to express that either a predicate has always been true in the past, or it has been true since another predicate became true.

TILCO has been extended 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* since the next time instant to the instant when B occurs for the first time since 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 its own distinct values. The communication among 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.

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 so as to make process reuse easier, since different process instances may have different parameter; a set of external TILCO *formulæ* describing the external process behavior through the messages exchanged and the 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 there can be many instances of the same process in the system, its specification is valid for all of them. By means of a *colon* operator applied to process components, process and local variables 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 when the process starts its work: before that time, all 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, its subprocesses start as well.

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. While waiting, the temporal expression `<whileExpr>` is asserted.

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

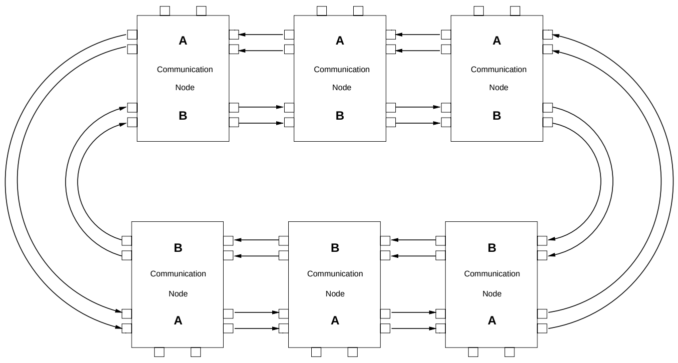

Fig. 1. The communication system

This communication basically aims at being robust against a single node failure and at distributeing the communication priority uniformly 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 marked with **A** letter and the backup with **B**. Input and output port for each ring are 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 it transmits the data to the adjacent while keeping the token and, when the transmitted message returns back to the sender, it releases the token. A node recognizes a fault communication after a time-out and it redirects the communication in the backup ring which works in the opposite direction of the main 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 it is transmitted over a dedicated channel. The message is treated as a 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];
}
```

3.1 System requirements

The node basically performs the following operations:

- it waits for the token;
- when the token is received, 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 so as to allow the other to transmit their own data.
- any data received on the backup channel has to be retransmitted without any check, 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 in order to keep communication alive. For this reason, the node system is decomposed in sub-systems. The decomposed communication node is shown in Fig. 2 and it 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**: like the Token Repeater, simply it handles data.

The main component is the Communication Manager the specification of which is made of several parts, so as to assure a better understanding of their meanings.

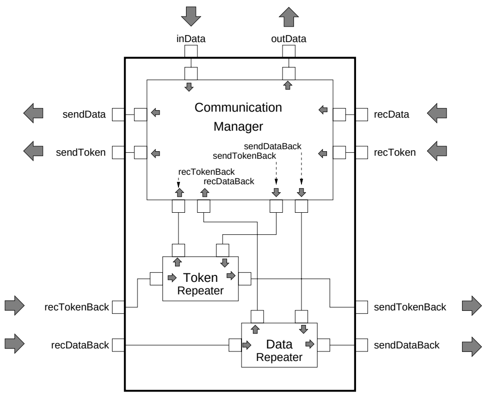

Fig. 2. The decomposition of the communication node

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

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

A “ready” predicate, initialized when the node process starts, puts the system in a wait status for receiving. While the node is waiting for the token, it can transmit nothing (neither token nor 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.

Concerning the data transmission the requirements specify that a new data is received from the higher layer input port and it 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 till nedded, which is to say when there is 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, providing a token (or a data) redirection on the backup ring whenever 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:

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

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

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

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

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

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. The process initialization puts in a waiting status all the system's receiving ports. Therefore in order to complete the specification, specific predicates have been introduced to assert that no data or token is sent until the apposite predicate is activated. The initialization expression is as follows:

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

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

4 Validating the specification

In order to prove properties at single process level 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.

The following two theorems have to be considered in order to prove properties for a single process :

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

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 was received in the past and it has not been acknowledged yet. These two main theorems are as follows:

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

Which 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 since 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 copes with the opposite case: when there is no pending message, the Send is done after the Receive or within the delay.

The Communication Node safeness properties about token/data transmission can be proved for single processes level. The $readyForToken$ predicate cannot be asserted together 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 into:

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

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

which can be proved using the specification.

Moreover, out of this result a safeness property can be derived: 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 demonstrate that such a critical condition cannot be met, 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:

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

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

The second sub-goal is directly solved by the safeness condition

$$:readyForToken \Rightarrow \neg :transmitAnything;$$

Whereas the first subgoal can be proved by looking at the specification ruling the system when it waits for the token and after it has arrived. From the expression written at page 7 the following proof status can be achieved:

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

The specification must be validated against the integration of the component. The property which grants the token passing, ensures a balanced communication priority for every node of the ring. The token passing is quick and, on the basis of a small delay of port communication, it 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) what has been supposed is 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.

5 Conclusions and Future Work

Verification and validation is very important for systems which are built 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 work out easily 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.

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.