TCOM : a Temporal Component Oriented Methodology for the industrial Automation Engineer

G.Vidal-Naquet[1] and H.G.Mendelbaum[2,3]

[1]Ecole Supérieure d'Electricité
rue Joliot Curie, Plateau de Moulon, 91192 Gif-sur-Yvette Cedex, France
[2]Jerusalem College of Technology - POB 16031 - Jerusalem 91160, Israel
[3]Univ. Paris V, Institut Universitaire de Technologie, 143-av. de Versailles, Paris 75016, France

Abstract

We present in this paper a methodology for the industrial Automation Engineer to help him design, build and run computer-based systems containing, physical and software parts. This methodology enables him to define a system through its requirements and its specifications, and it allows him to validate and then to run it (for instance on a PLC, a Programmable Logical Controller). This methodology is based on the description of the computer based systems in terms of hierarchical temporal-components, which are described as synchronous subsystems validated through the use of temporal logic. The concept of temporal-component is an extension of the concept of objects, it is made of four parts : **(i)** *a declarative part giving the links with other components, the signals and the variables,* **(ii)** *a set of local operations (hard/soft),* **(iii)** *a requirements part (named 'Goals') and* **(iv)** *a working specification part (named 'Behavior' or 'Controller'), these two last parts are written using an "engineering dialect" of the Temporal logic. The validation process consists in proving that the Behavior specification of the whole embedded-system satisfies its main goal using all the subcomponents' Goals (for this, one can use an automatic prover such as the Stanford's STEP prover). These Temporal logic requirements and specifications can easily be translated in Ladder Diagrams and run on a PLC (Programmable Logical Controller) using for instance the international standard IEC 1131-3. Our "engineering dialect" of the Temporal logic is built so that the translation rules from the TCOM notation to provable Temporal Logic and also to executable Ladder Diagrams (PLC) are simple. A bottle filler system example is presented.*

Keywords : *Components methodology, Temporal Logic, CBS, Automation, PLC.*

Introduction

In the process of building a computer-based systems, the ultimate goal of the engineer should be the satisfaction of the client's demands and expectations.

All engineering methods include at some level the expression of what the client expects from the system. At some point in the process, there must be a formal description of what is expected from the product (i.e. the specification), and also a validation step, meaning an argumentation that what the system does is conform to the specifications.

One of the main difficulties is how to go from an informal specification (usually described in a natural language) to a formal specification. Only from this formal specification a validation diagnosis can be deduced.

For embedded systems this problem is particularly important, but also particularly hard, since real time constraints, distribution and safety are crucial issues. In these cases, temporal logic has shown to be a powerful tool for specification and for validation [1]. However, temporal logic specifications are hard to make, they have been used for large systems (such as avionics, train design etc.) and could be used even more by the industrial automation engineers.

The aim of this paper is three-folds :

- First, we want to introduce a notation that is relatively "clear and natural" for the automation engineer to express the expected properties (requirements and specification) of the system he wants to design.
- Second, this notation must be close enough to classical notations of temporal logic [1][6] (such as PTL[1][2], TLA [3], Metatem [4]) so that it can be used relatively easily by temporal logic provers such as Stanford's STEP [5].

- Third, we want that this use of Temporal Logic can be integrated in today's development methods based on the notions of objects, components, and reuse[7].

I The TCOM approach

A computer-based system can be viewed as a large active component composed of various other components described in a hierarchical form.

Each component is made of

- other active sub-components,
- a logical controller
- physical or software passive parts.

The active part of a component is its logical controller which reacts to conditions coming from internal or external components or physical/software parts, and orders the elementary actions of its sub-components or physical/software parts.

Each component must satisfy a functionality which has to be specified, and validated with respect to its specification.

I.1 Development process

In order to build a new component, an engineer can use a set of validated existing components and physical/software parts. He will write a control software to control these components. Associated to the controller, there is a set of temporal formulas which express the behavior of the controller. This new component has itself a goal defined also with temporal formulas. It is possible to use

- either a bottom up approach: from the sub-components, the new component's behavior-controller is conceived to obtain the new goal,
- or a top down approach : from the goal of the new component to build, existing components are chosen, and the new controller is designed.

In either case, the validation will consist in showing

(B) and (SCG) implies (G)

where *B* is the formula expressing the behavior of the controller, *G* is the formula expressing the desired property (Goals) of the new component, *SCG* is the conjunction of all the Goals of its sub-components.

If we want such a proof to be carried out, goals and behaviors must be expressed in the same logical notation (here it will be in Temporal Logic).

The conditions can be based on internal boolean flags or external valued fugitive signals exchanged between components, or exchanged between some components and the outside world. Actions are started, and possibly stopped by the logical controller through the use of fugitive signals or permanent flags, they can also induce the emission of other signals or the setting of other flags.

We consider systems (such as industrial PLCs) where the fugitive signals and the persistent flags are taken into account in a periodic way using clock cycles, and at each cycle the elementary actions of the logical controller are completed before a new clock cycle arrives. This allows us to follow the synchronous model [8], and has the consequence that a fugitive typed signal, emitted during a cycle of the system's clock, is present and detected only during the same cycle. In addition we have also typed variables and boolean flags whose values are persistent during several clock cycles until they are changed or reset.

I.2 Components specification :

When using this method, the engineer will first specify each embedded component (software and/or hardware) using four types of specification :

a) Signals, variables and sub-components declarations (*D*). In these declarations, the interface of the component with other components is also given.
b) Description of the actions to perform in the component (hardware or software) they can be described by function activated by the behavior Controller when some conditions are fulfilled.
c) The goal of the component *(G)* meaning the temporal properties that the component must satisfy,
d) The component behavior *(B),* which describes the ordering of the operations, this describes the logical controller properties.

The TCOM specification can also serve for simulation and prototyping , in addition to the validation.

Therefore this method can be used :

- by the client to express his requirements,
- by the application engineer to specify the system,
- and also by the certification engineer to validate the system.

Let us describe intuitively our paradigm and the syntax we are led to use.

Each component is described in the following manner :

```
COMPONENT    X
{ DECLARATIONS : ...
  OPERATIONS :
  GOALS : ...
  BEHAVIOR : ...
}
```

Let us describe each paragraph of the component specification

Declarations (*D*)

Each Component contains a set of variables (they can be simple boolean flags or typed values). The exchange between components, or between components and the outside world, may be done through a set of valued fugitive signals or persistent variables.

The Declarations are described using sentences of the form :

*DECLARATION*S:
IN list of typed fugitive signals or persistent variables which can be received from other components
OUT list of typed fugitive signals or persistent variables which can be sent to other components
LOCAL list of internally used typed fugitive signals and persistent variables
INCLUDES list of sub-components

Operations *(O)* :

To each Operation is associated a flag (true or false) indicating that the operation has to be performed or not. The Behavior controller orders the Operations by setting or resetting these flags, the operations can be done through hardware, physical parts or software (function calls).

The Operations are declared using sentences of the form:

OPERATIONS : list of the operation-flags and their corresponding software/hardware/physical functions.

Goals*(G)*

The goals express the logical properties that the component has to fulfill.

The goals are described using sentences of the form :

*GOAL : When (*Condition*) Then (*Property*)* ;

this corresponds in PTL notation to "Condition ==> Property "

(Condition true implies that Property is also true)

Behavior *(B)*

The Behavior expresses the logical work of the component controller .

The behavioral propositions are described using sentences of the form :

*BEHAVIOR : When (*Condition*) Then (*Reaction*);*

this corresponds in PTL notation to "Condition ==> Action"

(Condition true implies that Action becomes true)

For the conditions, the properties and the actions, we adopted a dialect of Temporal logic which is sufficiently close to the automation engineer vernacular language:

For instance <> is replaced by "*later*", [] is replaced by "*always*", () is replaced by "*next*", (-) is replaced by "*before*", "*A Until C"* means that action *A* is performed until the first clock-cycle when *C* becomes true (which is the usual interpretation of the English word "Until"), etc.

I.3 Composition of components

The declaration part of a component specifies internal names of signals that are exchanged between the component and other components (the "outside world" being also considered as a component). One simplified architecture could be that all the signals and variables of the system, are global and known by all the components.

In an encapsulated architecture, each component would have private local signals and variables that it can exchange with other components. Input signals of a component should be identified with output signals of other components. For the purpose of reusability and maintainability, we should need to describe the correspondence of the different signals emitted and received by the different components. This could be done using an approach similar to the ROOM (Real-Time OO model [11] is is based on capsules communicating with other capsules through ports. The ports communicate with each other using signal-based connectors. These capsules and ports are modeled as UML classes augmented by stereotypes to denote input and outputs methods. It can also be done using ideas of configuration languages [9][10][11]which describes the links between components, so that in case of an architectural change, it is only necessary to change this link description. So, with this option the links between the components are described in a loose manner allowing to change them easily without changing the internal components specifications.

II Semantic of the TCOM notation

TCOM is based on the Temporal Logic as defined in [1-3], so TCOM describes the evolution of the system using the reactive model to describe the controller behavior.

In this paper, for brevity's sake, we present the version of TCOM based on classical Propositional Temporal Logic [1] [2]. Formulas are constructed from

a) A set of propositional variables,
b) boolean connectors,

c) Temporal operators.

The temporal operators are designated with clear names so that an engineer can express easily different behaviors. They can be translated directly into PTL.

II.1 Propositional variables are typed according to their semantics:

a) Presence of a fugitive signal
If I is a fugitive signal, Pres_I is the propositional variable which has the value True when I is present at the logical instant in the synchronous model, meaning it is detected only during the present cycle of the synchronous clock

b) Emission of a fugitive signal
If O is a fugitive signal, Em_O is the proposition which has the value True when O is emitted by the component's controller, during a cycle of the synchronous clock. This emission implies that this signal is present, and can be detected in a condition only during the same cycle of the synchronous clock . This is compatible with the synchronous model [8] according to the axiom :
$[]Em\_S \rightarrow Pres\_S$

c) Permanent variables : are typed variables which have values which remain during several cycles. Boolean permanent variables are called Flags.

d) Setting of Flags
If F is a flag , the propositional variable Set_F has the value True when the flag is set to the value True.
If F is a flag, the propositional variable Reset_F has the value True when the flag is set to the value False

e) Value of Flags
For each flag F, the propositional variable Val_F has the value True (or False) when the flag has the value True (or False)
There is an implicit axioms :
When Set_F then Val_F Until Reset_F
When Reset_F then not Val_F Until Set_F
This expresses the fact that the Flag retains its value until it is changed by Set or Reset.

f) Properties
A property P is associated with the propositional variable p, p has the value True when the property is satisfied. Formulas will define when a property P is true, i.e;. when p has value True

II.2 Conditions

An elementary condition is a formula of the type
Pres_S or Val_F
An elementary condition is a condition,
a boolean combination of conditions is a condition.
A condition can be a boolean combination of presence of signals, or conditions on the values of variables and signals, or a temporal logic formula, which does not involve future temporal operators.

II.3 Actions

When some action-signals or flags are turned on, it will perform corresponding Operations (on hardware or on physical parts or software function calls)
An elementary action is a formula of the type
Em_S or Set_F or Reset_F
An elementary action is an action,
A boolean combination of actions is an action,

NOTE : To simplify the engineer's notation, in our Temporal dialect, instead of writing explicitly Val_F or Set_F etc., when the context has no unimbiguity, we shall simply write 'F' in the Conditions to indicate Val_F, or 'F' for Actions to indicate Set_F, and '!F' for Reset_F. We shall use analogous conventions for signals (to avoid writing each time Em_S , Pres_S etc.).

II.4 Behavior Controller

The reactions of the controller is described by a set of sentences of the form :

*When (*Condition*) Then (*Reaction*)*

At each cycle of the synchronous clock, the controller is activated, all conditions are evaluated, and for each true condition the corresponding reaction is done, right away.
This correspond to the notion of guard.
A reaction can be simple of combined. For example, a simple reaction can consist in the emission of a signal,
A combined reaction can be of the form

- A reaction and another reaction
- A reaction and a reaction starting at next instant
- a reaction is performed done until a condition is fulfilled

We do not introduce conditional reaction or looped reaction. We want to stick to what can be described simply by a PLC (programmable Logical controller).

II. 5 Goals

Goals will be expressed as temporal formulas, of the form *When (*Condition *and* Properties*) then (*Properties*)*

The Properties can contain both past and future temporal operators.

III Example : the bottle filler system

The following example is a simplification of the Production Cell (see Fig. 1).

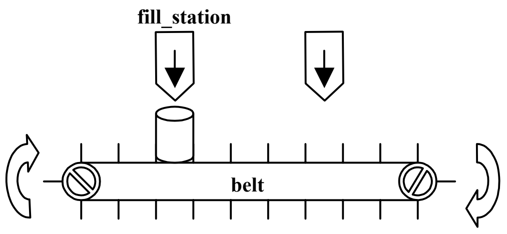

Figure 1 : the Bottle filling system

III.1 Informal Specification

It is composed of a conveyor belt, with a start-location, an end-location, and two special locations (under a Filling station and under a Sealing station) where the bottle must arrive, in order to be filled and sealed.
The bottles enter through the start-location on the belt, the goal is that when the bottles reach the end-location , they are filled and sealed.
The belt moves the bottles to each station. When a bottle reaches the "Fill_station" it is filled, and when it reaches "Seal_station" it is sealed.

Each time the belt is given an order "doMove" it moves to the next station and stops automatically. When a bottle arrives to a station, a sensor detects it and emits a signal corresponding to the Station: "S_BatF" when the bottle arrives at the Filling station and "S_BatS" at the Sealing station. When a bottle is put on the first slot of the belt, a signal "S_EnterB " is emitted. When a bottle passes to the last step of the belt, a signal " S_ ExitB " is emitted.

III.2 Methodological issues :

1) Our approach relies heavily on the classical notion of reuse. For example, to build a main component named "Bottle_filling_system", we shall reuse existing components such as a machine that fills a bottle (the "Fill_station"). One of the properties of the "Fill_station" will be that when it receives a 'S_Fill' signal from the "Bottle_filling_system" controller, it will fill the bottle and will send back a 'S_Full" Signal to it, with the semantic that the bottle have been filled (the property : 'P_BottleFilled' is satisfied, and when a bottle is filled, it stays filled) , and this does not have to be proven again (the action has not to be redefined, it is internal to the existing "Fill_station" component).
2) In our approach, the engineer does not describe the execution program, but the specification of the program, a compiler will translate directly this specification in an executable code.

III.3 Writing the components

Here are the text of the components specifications :

```
COMPONENT   Fill_station
{ DECLARATIONS : ...
          IN        S_Fill :boolean signal;
          OUT       P_Full: flag ;
                    S_Full: boolean signal;
  OPERATIONS :   doFilling : FillingFunction( );
  GOALS :  when (S_Fill) then ( (later(S_Full));
           when (S_Full) then ( always(P_Full));
  BEHAVIOR :
when (S_Fill) then (doFilling Until S_Full);
when (S_Full) then ( always(P_Full));    }
```

```
COMPONENT   Seal_station
{ DECLARATIONS:
          IN        S_Seal :boolean signal;     OUT
          P_Sealed: flag ;
                    S_ Sealed: boolean signal;
OPERATIONS : doSealing : SealingFunction( );
 GOALS: when (S_Seal) then( later( S_Sealed) );
       when (S_Sealed) then (always P_Sealed);
BEHAVIOR: .
when (S_ Seal) then (doSealing Until  S_ Sealed );
when (S_ Sealed) then( always(P_ Sealed));    }
```

The goals of the filling and sealing stations stipulate that a signal is emitted at the end of the operation(S_Full , S_Sealed), so that other components' controllers can start other operations. The goals stipulate also the conditions on the working property flags (P_Full , P_Sealed), so that they can be tested in other components' goals.

```
COMPONENT   Belt
{ DECLARATIONS: ...
    IN  S_EnterB, S_BleaveF, S_LeaveS,
          S_Full, S_Sealed :boolean signal;
   OUT  S_BatF, S_BatS, S_ExitB :boolean signal;
   LOCAL :oneStep: integer=10;
OPERATIONS : doMove : MoveFunction(oneStep);
GOALS: when (S_EnterB) then (later (S_BatF));
          when (S_BleaveF) then (later (S_BatS))
         when (S_BLeaveS) then (later (S_ExitB))
```

BEHAVIOR:
when (S_EnterB) then (doMove *Until* S_BatF);
when (S_ BleaveF) *then* (doMove *Until* S_ BatS);
when (S_ BleaveS) *then* (doMove *Until* S_ ExitB); }

The goal of the belt expresses the fact that when a bottle enters, later it reaches the filler, when a bottle leaves the filler, later it reaches the sealer.

COMPONENT MainSystem
{ *DECLARATIONS :*
IN S_BatF, S_BatS,
S_EnterB, S_ExitB ,
S_Full, S_Sealed *:boolean signal;*
OUT S_BleaveF, S_LeaveS,
S_Fill, S_Seal *:boolean signal;*
GOALS: *when* (S_EnterB) *then*
(*later* (S_ExitB and P_Full and P_Sealed));
BEHAVIOR:
when(S_BatF) *then*(S_Fill);
when(S_Full) *then* (S_BleaveF);
when(S_BatS) *then* (S_Seal);
when(S_Sealed) *then* (S_BLeaveS);
}

The Main System behavior expresses the fact that when a bottle reaches the filler, the signal S_fill is emitted to the filler, when the signal S_full that tells the bottle is full arrives, the bottles leaves the filler...etc.

IV Translation into PTL and Validation

The translation is straightforward :
the key words *When, then, later, next, always* etc... are replaced by the classical PTL notation : ==> , <> , (), [] etc... The name of the signals and flags are unchanged.

The formula given to the Stanford's STEP linear PTL validity checker [5], is :

the conjunction of the goals of the components and of the behavior *implies* the goal of the main-system. Here is the formula given to the prover :

```
SPEC
variable S_Fill, S_Full, P_Full, S_Seal,
S_Sealed, P_Sealed, S_EnterB, S_BatF,
S_BleaveF, S_BatS, S_BleaveS, S_ExitB :
bool Flexible
macro Filler_Goal:  bool where Filler_Goal
=
((S_Fill ==> <>S_Full)/\(S_Full
==>[]P_Full) )
macro Sealer_Goal:  bool where Sealer_Goal
=
(       ( S_Seal    ==> <> S_Sealed )
    /\ (S_Sealed  ==> [] P_Sealed )    )
macro Belt_Goal:  bool where Belt_Goal  =
(     (S_EnterB  ==> <>S_BatF )
   /\ (S_BleaveF ==> <>S_BatS )
   /\ ( S_BleaveS ==> <>S_ExitB )    )
macro System_Behavior:  bool
where System_Behavior  =
 (      (S_BatF    ==>   S_Fill)
   /\  (S_Full    ==>   S_BleaveF)
   /\  (S_BatS    ==>   S_Seal )
   /\  (S_Sealed  ==>   S_BleaveS )    )
macro System_Goal:  bool
where System_Goal  =
( S_EnterB  ==> <>(S_ExitB /\ P_Full /\
P_Sealed) )
PROPERTY:
(      Filler_Goal /\ Sealer_Goal /\
Belt_Goal
   /\ System_Behavior) ==> System_Goal
```

****

the Stanford's STEP linear PTL validity checker [5] answer was :

```
Checking
Filler_Goal /\ Sealer_Goal /\ Belt_Goal /\
System_Behavior ==> System_Goal
Building transition relation...done
PTL Valid
```

Note: This validation process is made possible only by the fact that the components form a synchronous system, i.e. that all the properties of the various components are tested simultaneously at each cycle of the clock.

V PLC Execution

Here we use the Simlev PLC editor and executer [13] which allows to simulate Allen-Bradley, Texas-Instruments and IEC1131-3 PLCs : (see Fig. 2)

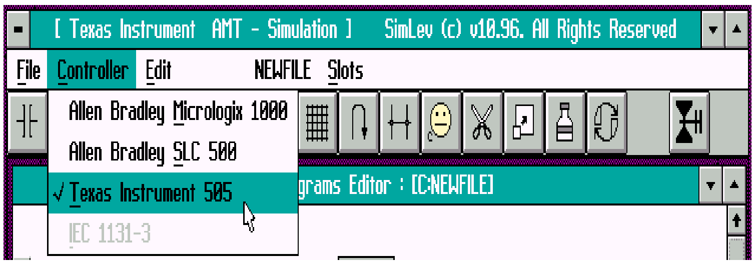
[Texas Instrument AMT - Simulation] SimLev (c) v10.96. All Rights Reserved
File
Controller
Edit
NEWFILE
Slots
Allen Bradley Micrologix 1000
Allen Bradley SLC 500
√ Texas Instrument 505
IEC 1131-3
grams Editor : [C:NEWFILE]

Figure 2: Simlev PLC Editor and Executer

For TCOM, we could translate strictly each component-behavior into a separate PLC, this would imply that all the PLCs can communicate together (for instance through an interconnecting common Bus which allows to transfer all the signals to all the PLCs). To simplify let us translate all the components behavior into one PLC.

V.1 Components' Ladder Diagrams [13]

Here is the translation into a PLC Ladder Diagram (see Fig. 3) :

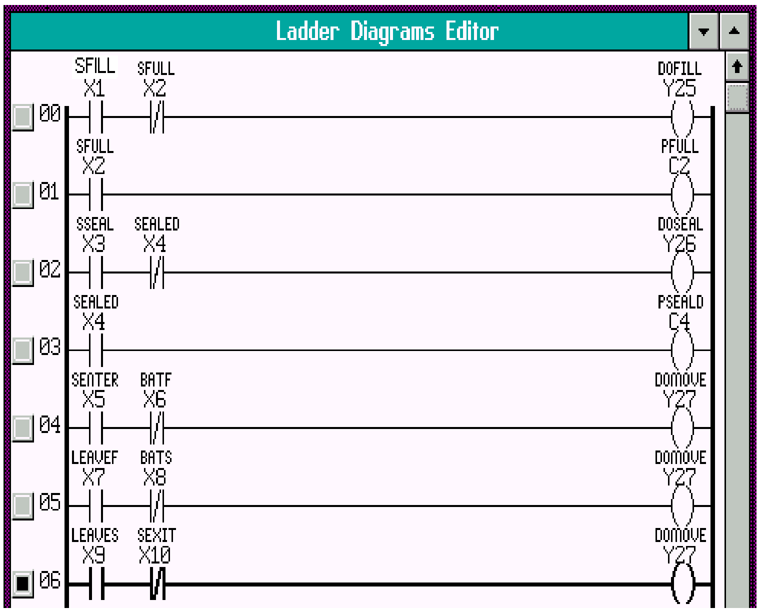

Figure 3: sub-components Ladder Diagram

This means that this PLC will scan at each cycle :

The Fill_station's behavior:

line 00, if the action-signal S_Fill is ON (produced by the System's PLC), and if the current bottle is not yet full (signal S_Full OFF), it will perform the local Operation doFilling. When the signal S_Full becomes ON, this will disable the line 00, and the doFilling will become false.

line 01, When the signal S_Full becomes ON, it will set the persistent flag P_Full to True, so that the Bottle System's PLC can test it and order the further Operations.

The Seal_station's behavior is described in the same way in the lines 02-03.

The Belt's behavior :

In the lines 04-06 we can see that this PLC will scan each line at each cycle and will perform the output-action *doMove.* For instance, for the line 04, *doMove* will be performed when the signal *S_EnterB* is ON (produced by the Physical System) only if the signal *S_BatF* is OFF. When the signal *S_BatF* becomes ON, this will disable the line , and the *doMove* will become false. The other lines work in the same way.

Here is the translation of the main-System's behavior into a PLC Ladder Diagram (see Fig. 4) :

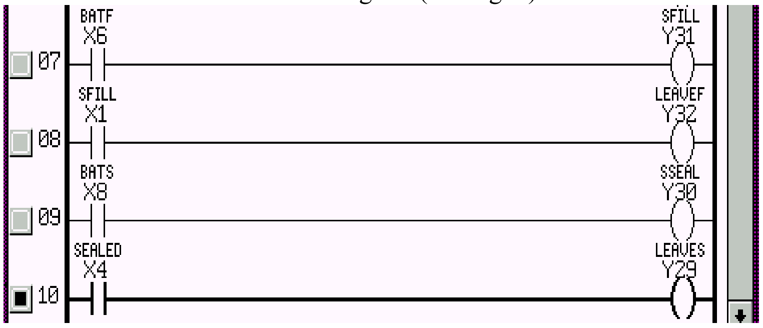

Figure 4: the Main-System Ladder Diafram

Here also the execution is simple : the outputs signals S_Fill, S_BleaveF etc. are produced when the input signals S_BatF, S_Full etc. appear.

V.2 TCOM to PLC translation Rules:

Roughly, the main translation rules are :

1- each *when*-line of the behavior-controller is translated in a Ladder Line.

2- each behavior *condition* is translated by a set of Ladder input flags or signals using the AND/OR syntax of the Ladders (i.e. --|S_input1|-----|/F_input2|--).

3- each behavior *reaction* is translated by a set of Ladder outputs flags or signals (i.e. ---(S_output3)---).

4- for each behavior-controller line, which contains an "*until* signal/flag" constraint for the reaction, we need to test that this "signal/flag" if OFF in the inputs of the Ladder-line.

5- if there is time constraints in the behavior-controller *conditions*, we have to introduce Timer-boxes in the inputs of the corresponding Ladder lines.

Conclusion

The field of Computer Based Systems is growing and needs to describe the integration of some parts in hardware and others in software. The component-oriented approach facilitates the description of these systems in several parts; furthermore hardware parts as well as software parts can be modeled by objects in the same manner.

Building a system hierarchically using components enables to build it by refinements. By defining gradually each component it permits the requirements, specification, and validation of each part, and then the whole system as a super-component. This approach can be helpfull not only for specific Real-Time systems, but also for general distributed applications, since nowadays almost every modern application contains synchronization and temporal constraints, and there will be no need for two distinct methodologies to develop real time and non real time parts.

Our present proposal of an "engineering dialect" of the temporal logic has the advantage, on one hand, to be easy to understand by the industrial automation engineer, and on the other hand, to be easy to translate in a PTL format used by automatic provers such as the Stanford SteP, for validation.

Our proposal allows also to execute these specification (after validation), for instance using a simple translation in PLC Ladder Diagrams.

This proposal of Temporal Components can be viewed as an extension of the UML active object model or the Java Thread objects. Like these models it is hierarchical, it contains local variables (attributes) and Operations (methods). But, in addition, it contains a Description of on-line signals interface with other components, a Goal description giving the Temporal properties that the component has to fulfill, and a Behaviour Controller which reacts to the signals and the local conditions and performs the corresponding Operations.

Acnowledgements : We want to thank our colleagues Dr R.B. Yehezkael and Dr R. Gallant for their helpfull comments, and our Engineering students Menachem Yosub, David Rosenberg who worked with us on this subject to build up the translators for their final Engineering Project at JCT.

References

[1] Z. Manna, A. Pnuelli "The temporal logic of reactive and concurrent systems", Springer Verlag, New-York, 1992

[2] R. Alur and T.A. Henzinger. "Logics and models of real time: a survey", J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, Real Time: Theory in Practice, LNCS vol. 600, pages 74--106. Springer-Verlag, 1992

[3] L.Lamport "Introduction to TLA : the Temporal Logic of Actions", Digital Research report SRC 1994-01, Palo-Alto, dec. 1994

[4] M.Fisher, S. Kono, and M. Orgun (eds) Journal of Symbolic Computation, Special Issue on Executable Temporal Logics, 22(5), Academic Press, Nov/Dec. 1996
M.Fisher "A survey of Concurrent Metatem, the language and its applications" , M.Fisher@mmu.ac.uk

[5] Z. Manna and the STeP group," STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems", 8th Intern. Conf. on Computer-Aided Verification, LNCS, vol. 1102, Springer-Verlag, pp. 415-418,
July 1996 - http://www-step.stanford.edu/

[6] F. Moller,G.Birtwistle, editors, "Logics for concurrency : structure versus automata" LNCS vol. 1043, Springer-Verlag, 1996

[7] F. Boulanger, G. Vidal-Naquet "An object Execution model for reactive modules with C++ implementation", ECOOP'96, Linz, July 1996 , Max Mühlhäuser editor, dpunkt.verlag (1997)443-449

[8] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," Science of Computer Programming, Vol. 19, No. 2, November 1992

[9] J. Magee, J. Kramer, and M.S. Sloman "Constructing Distributed Systems in Conic",
IEEE Transactions on Software Engineering, Vol 15 No. 6, pp 663 - 675, 1989.

[10] J. Magee, N. Dulay, and J. Kramer, "Structuring Parallel and Distributed Programs"
Procs. of the International Workshop on Configurable Distributed Systems, London, 1992.

[11] H.G. Mendelbaum, R.B. Yehezkel , Y. Wiseman, I.L. Gordin, "Experiments in separating Computational Algorithm from Program Distribution", PARA2000 Workshop, Univ. Bergen, Norway, june 2000, Procs. in LNCS of Springer Verlag 1947, 278-269 (2001)

[12] "PLC software standard : IEC 1131-3", International Electrical Comitee Publ., Geneva, 1988
R.W.Lewis "Programming Industrial Control systems using IEC 1131-3", Institution of Electrical Engineers Publ., London, 1995

[13] H.G. Mendelbaum "SimLev User Manual ; A graphical PLC simulator", Eshed Robotec Inc., Tel Aviv, 1996

[14] Selic B., Rumbaugh, J. : Using UML for Modeling Complex Real-Time Systems.
ObjectTime Ltd/Rational Software Corp. White Paper, (March 1998)