Integrated Development Environment

for Real-Time Systems:

Temporal Logic and C++

Introduction

Often we have got an idea. In mind the power of abstraction makes the project simple and easy to understand in details. If someone asks for description, it will be natural defining, rule by rule, every feature or requirement of this imaginary object. A logic mind does not meet any problem in formally showing what it want to.

Sadly, during the project realization, some steps do not appear so easy, no more. It is mandatory taking compromises and being satisfied. In the end, the idea, such a high-level concept, becomes a prototype that only just can fit some base-requirements, which were skipped during the analysis. Why such a pure idea cannot find a satisfying realization ever?

About software development, the same problems grow up: the products are acceptable, but not exactly our aim so far. In some cases, in applications where system-fault are not allowed like real-time system, the classical approach of development is not enough. It is necessary to safely engage description and realization.

The project, illustrated in this article, is an important result about communication between description and realization: more closely between temporal logic TILCO-X, useful for describing the behaviour of systems, and C++ language, needed to execute functions for system manipulation.

This document will take an overview of a real-time controller final product of this new approach. It is organized in several section: first of all it is necessary to give an idea about how the controller works, then it is suitable to explain which develop system is provided to set-up every part of the controller. In the end a simple use case is discussed, form the behaviour specification to the I/O function customisation.

Control model

The real-time controller join two opposite program approaches: operational and denotational.

The operational approach, the most used, allows the programmer to force the sequence of the operations for the system and to control the time when they are executed. In the denotational way, who programs, can show to the machine the rules that it must respect, without explanation about how it can satisfy those rules.

The control model we show tries to get maximum benefit from both approaches. More precisely, our aim is to contain every relation rule about system events in a formal language, enjoining all the advantages of a strict description, and to handle the real-world interaction by operational programming, using powerful instrument to access to data-structures, drivers and system environment.

Furthermore, this controller contain:

- a “logic brain”, really flexible about event relation changes;
- a “real-time heart”, that synchronizes every control action and samples the system at a fixed time-resolution;
- several “procedural senses and limbs”, which can easily interface to every device and make the controller itself universal, that is to say effective with every kind of application.

In **Figure 1** is illustrated a intuitive view of the controller structure.

The controller is modular, inside is built by many element of the same kind. These blocks are the executable form of the behaviour specification written in TILCO-X or parts of it. They are contained in a real-time structure, where a “sequencer” periodically evaluates the conditions, deduces the action to apply and executes them. These evaluations and actions are performed by a set of C++ procedures; they are the interface between logic and procedural environment and make the controller reactive.

The modular structure allows the programmer to reuse several sub-parts of the controller for more applications, as example the C++ interface used to control a device, or the logic specification of a periodic events. In this way it is realizable a library of logic and procedural parts.

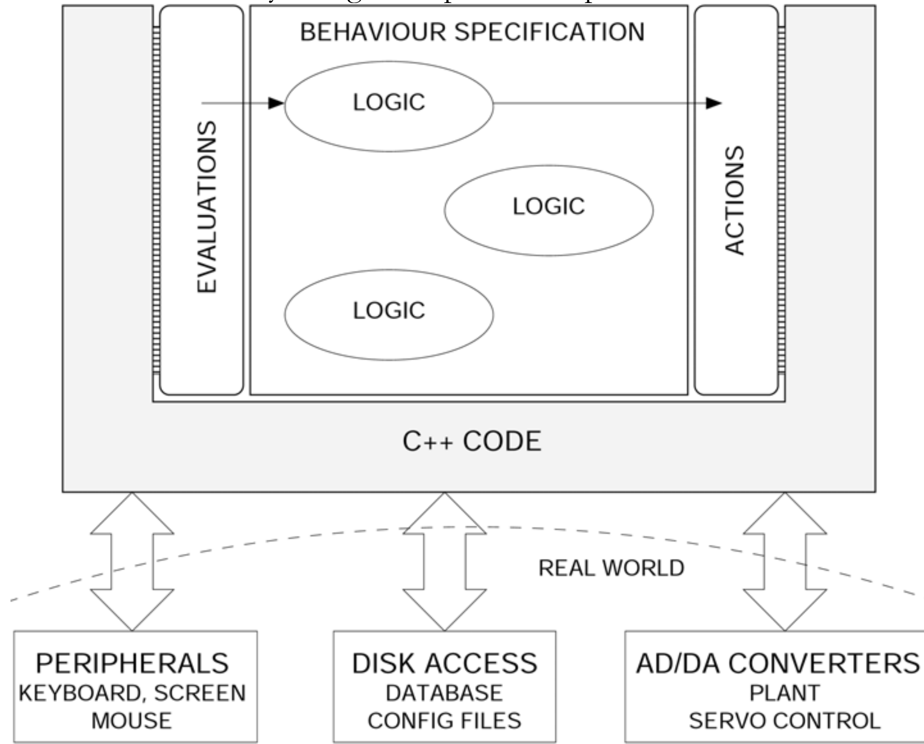

Figure 1 - Control model

The controller is discrete-time; the step sequence that gives reactive properties is based on three step, as illustrated in **Figure 2.**

1. **Evaluation** - at the sampling instant the input interface carries out the evaluations: user-defined functions that check condition in the system.
2. **Logic deduction** - during the sample period the deductive part takes the input result events and, based on the behaviour description, generates other output events.
3. **Action** – at the next sampling instant the output bound change the system executing actions related to the output events: as the input evaluations, they are user-defined to perform the suitable action. The reactive changes to the system are performed by the controller only after the whole sample period is elapsed.

The input-output delay is maximized to give enough time to the deduction step, but is kept constant.

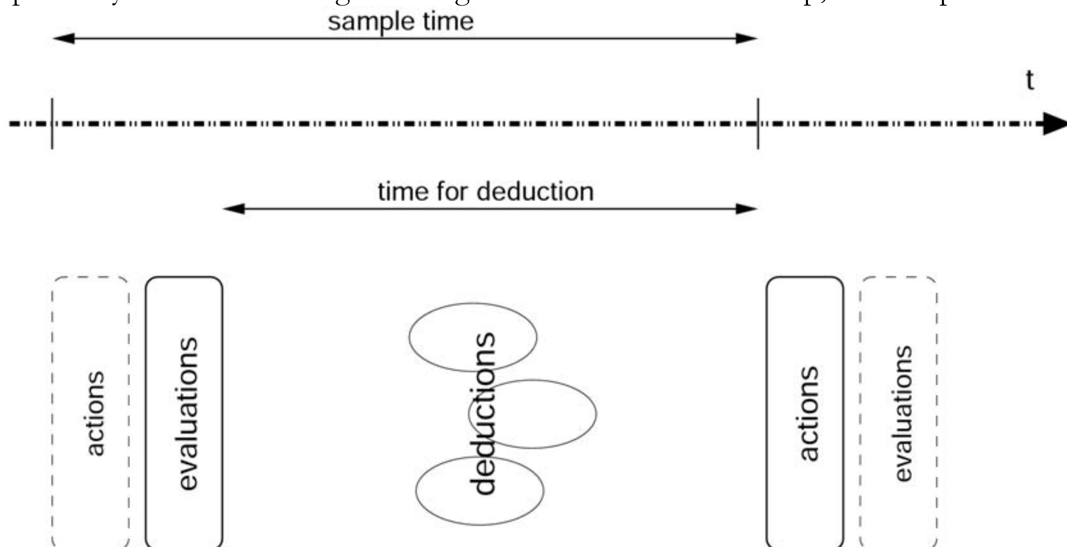

Figure 2 - Control sequence

Object Oriented Implementation

Every basic element of the control model becomes a C++ class or a group of them.

The logic engine has been created as **TIN** (Temporal Inference Network) class, which loads from a file the behaviour specification (or a part of it) that it must control. For each I/O literal in the specification, this object keeps a references to an **IODev** class, which should be linked with an instantiated I/O device.

A generic I/O device is modelled by an **Input** class object for what is evaluated by the controller or by an **Output** object if it is linked to an output literal in the specification.

All the Input and the Output references are collected separately in two arrays called **InBox** and **OutBox**, while TIN object references are stored in a **TINBox** object.

The **Rtexec** class contains all these containers to run the basic steps previously described.

1. Accessing from the InBox, it executes the `scan` method of each Input device: this perform the evaluation step.
2. From the TINBox array it controls the deduction process ordering to each TIN object to read data from the Input devices and to start the inference algorithm to determine the output events.
3. When the sample period is elapsed, scrolling the OutBox container, it performs all the reactive actions that are necessary.

In **Figure 3** is illustrated an object diagram of the controller .

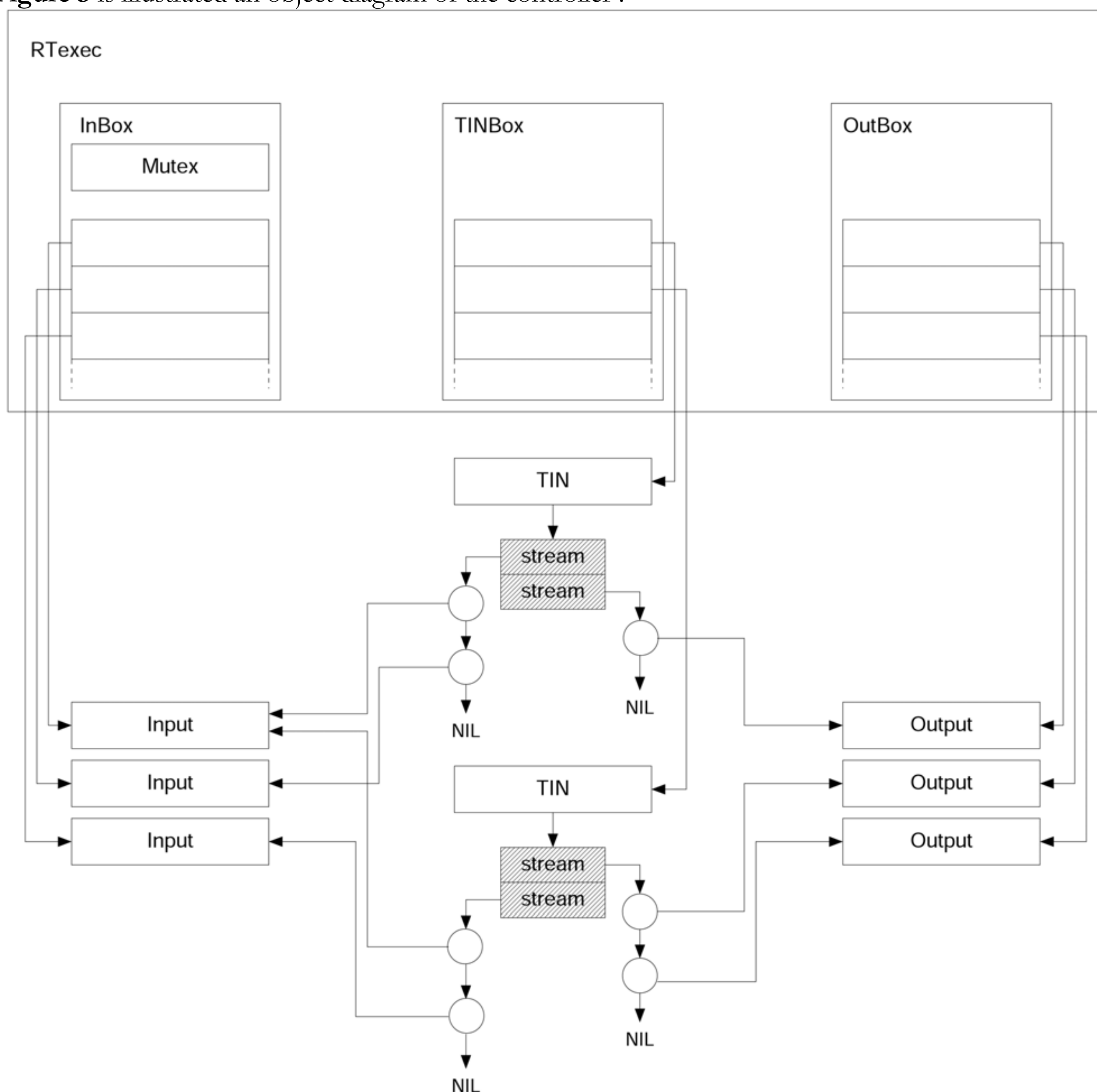

Figure 3 - Object Diagram

Process Control

The standard I/O models can be extended to obtain more complex structure, suitable to those realities which consider evaluation and action.

The process control needs thread and semaphore models, both of these use this new interface concept, illustrated in **Figure 4**.

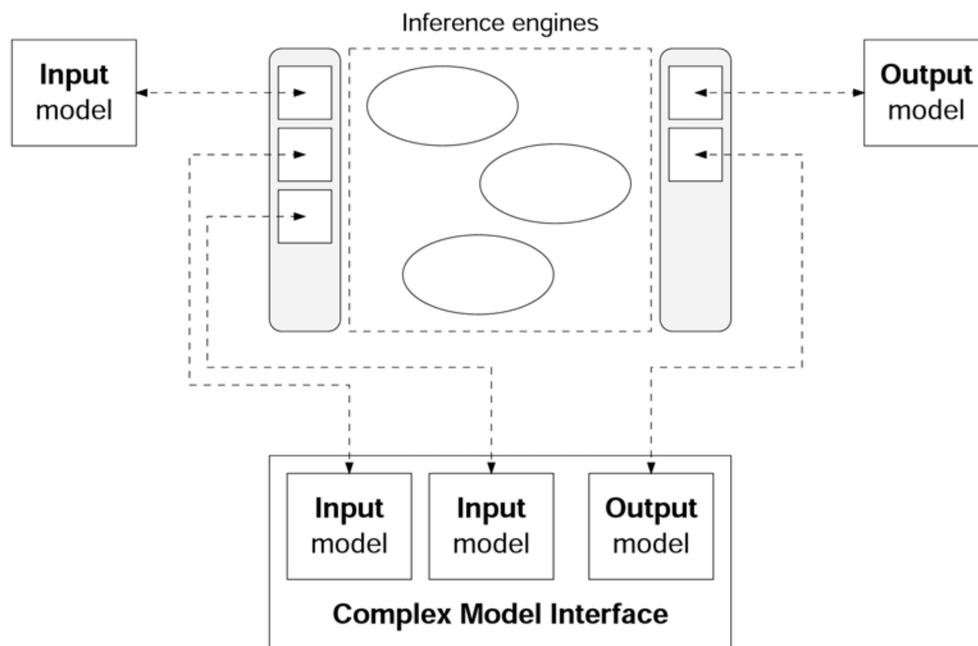

Figure 4 - I/O Model Extension

The idea is to model the thread and the semaphore within an Input device for each state condition that has to be evaluated and an Output device for every action that has to be performed on it. These included device are linked to the logic as the standard one.

Therefore the **TINThread** class is the thread model for the controller and it includes:

- a **StartOutput** object– the device that starts the thread,
- a **KillOutput** object – which terminates the thread,
- a **RunningInput** object – which evaluates if the thread is running,
- a **CommandInput** object – which is necessary to trace if the last execution succeeded.

The semaphore model is the **TINSemaphore** class and it provides:

- a **SignalOutput** object– the device that release a thread which was waiting,
- a **WaitingInput** object – which knows if a thread is waiting or not for the release signal.

Everyone of this device is inherited from the standard device classes Input and Output.

The class that combine this two entities is **TINThreadWithSemaphores** and is necessary to handle synchronization problem with TILCO-X.

From the "logic side" there are function-literals correlated to every action/evaluation of these objects; this allows to write specifications that are readable and semantically controllable.

The TILCO-X compiler syntax must include new types for the parameters: thread and semaphore.

The function-literals (or parametric literals) that use these new variables are illustrated in **Table 1**.

	Using thread only	**Using thread and semaphore**
Output	*start(<thread>)*	*signal(<thread>,<semaphore>)*
	kill(<thread>)	
Input	*running(<thread>)*	*waiting(<thread>,<semaphore>)*
	succeeded(<thread>)	

Table 1 - Process control function-literals

Architecture

This project is grown-up from a starting point that is TILCO-X; around this important language took place a bigger project, which includes other tools for the logic manipulation: TOTS.

This is really useful because, dealing with all behaviour aspects inside the logic, it is necessary to confirm exhaustively that the specification really fit with the problem. Before insert a specification inside a real-time controller every kind of checking has to be made.

Briefly, TOTS includes:

- instruments for property proof of the specification, *Isabelle* and *Giselle*;
- a compiler, *tilcox2tin*, that translate the specification in a Temporal Inference Network, which is a representation of the same rules but in a more executable format, editable by *TinEdit*;
- instruments for simulation tests, made with simple file I/O, suitable for the history checking, which are tinx, for the execution, and SgnEdit for a user-friendly handle of the signals;
- a develop system that allows, after a valid specification was written, to include it as controller engine fixing up the I/O function; this raise TILCO-X to face directly to the real-world.

Every part of TOTS and the steps in building a logical controller are illustrated in **Figure 5**.

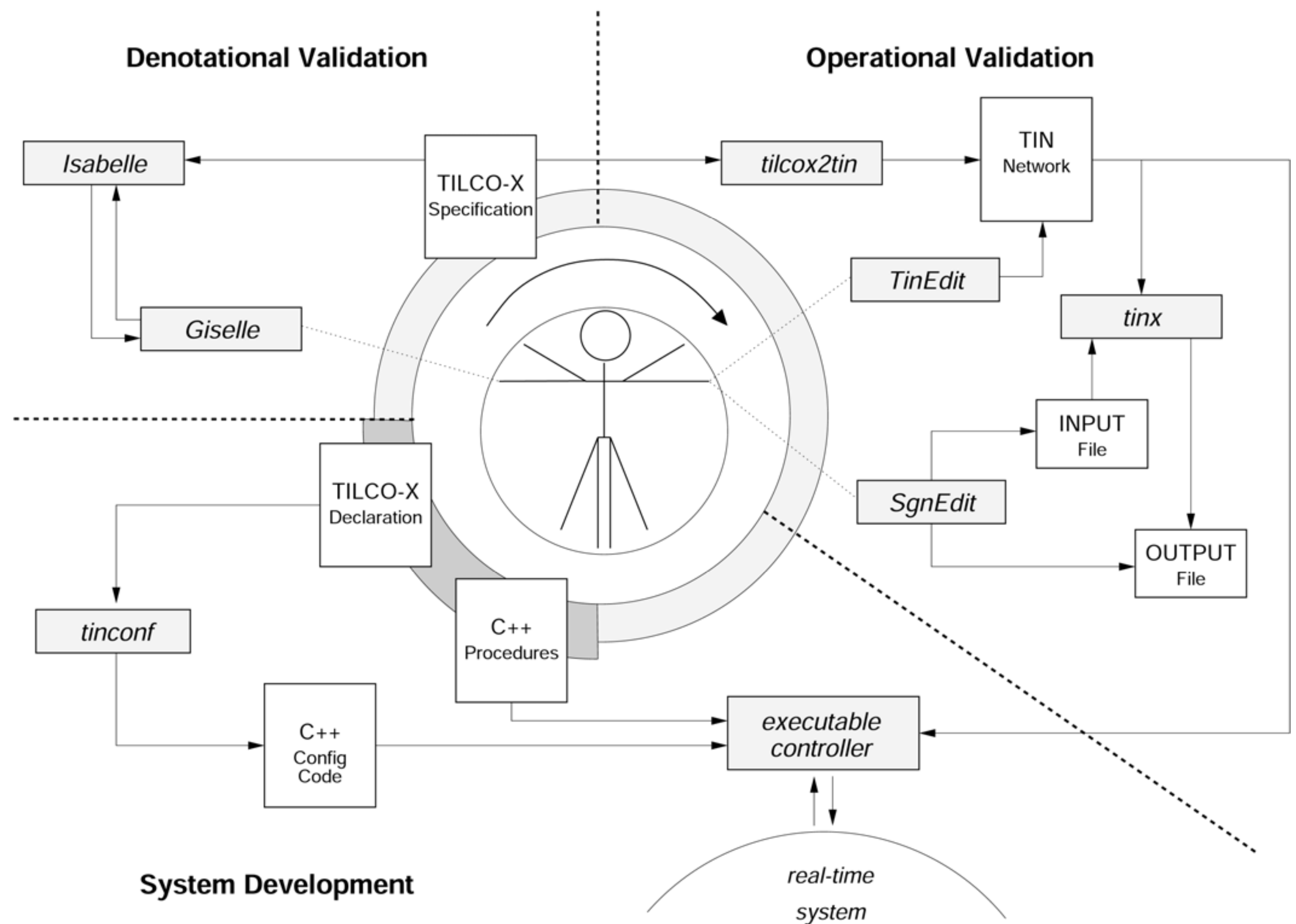

Figure 5 - TOTS

The steps of describing the behaviour and defining the actions and the evaluation of the controller are separated. By the way, it is important that the specification and the procedures are conceptually joined. In fact is possible to write a rule like $hightemp@[-5,0] \Leftrightarrow start(cooler)$ and is intuitive what it rules: if a read temperature is too high during some time instants a cooler task must start. It would be nice that specifications was written thinking about what they are designed to control. This make TILCO-X a really useful language to control real-time systems.

Development system

This new method to define real-time system would not be really advantageous without an usable development system; a tool which aids the user to build the system completely, starting from the idea.

The attention has to be focused on the description step, this is the most risky part of the development, where every small bug generates unpredictable behaviours, which are hard to find in a debug section. A part from the denotational validation of the rules, the link step between rules and interfaces to the external environment should be strongly aided.

A semantic control is suitable to obtain a good specification linked to the external world correctly; if the action is written in the associated literal, as example starting a thread is *start(<thread name>)*, the semantic control can see if *thread name* is effectively the name of a system thread. The programmer has to list every object that plays in the controller, as threads, semaphores, inputs and outputs; this information allows the development system to make the semantic control on the rules of the specification.

To create the controller in an executable form the following step has to be considered:

1. list of all the entities of the controller in a declarative file ("`.bth`" extension), these names could be used as variable in the behaviour specification;
2. to explain the relationships between events in one or more specification files ("`.bth`" extension), every requirements that express a synchronization rule must appear in this section;
3. to compile the specification files with *tilcox2tin* including the declaration part, this ensure the semantic control checks the consistency of the rules;
4. to run the code generator *tinconf* based on the declaration file, it creates C++ source files which automatically sets up the controller creating and linking the interfaces to the literals in the rule;
5. to customize the source code to perform the right control action, filling some empty procedures as in the event-driven programming style;
6. to link the C++ source files with the controller kerner, which is a class library created for this aim, obtaining an executable file, which will load logic files ("`.tin`" extension) at run-time.

In **Figure 6** these steps are reviewed.

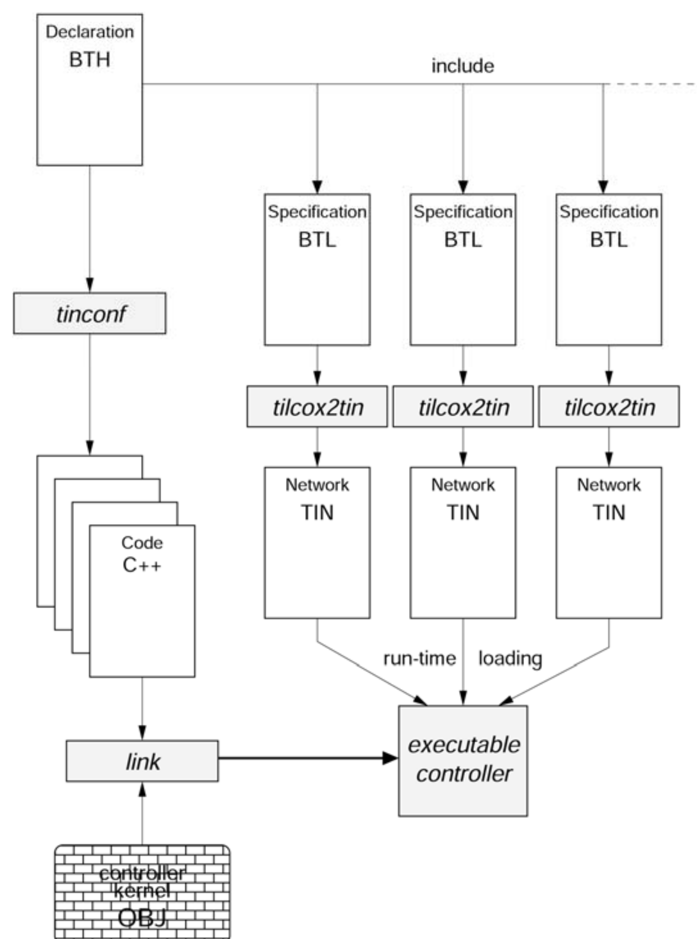

Figure 6 - Development system

Control Example

The example we deal with consider same aspects that are basilar real-time possible requirements: periodicity, time-out, mutual exclusion. The problem requirements schema is illustrated in **Figure 7**.

The key elements of the behaviour are:

- **Refresh** – the thread who refresh the data-structures in memory must run periodically to maintain true values in the apposite data;
- **Retry** – another thread, who accesses to a communication channel, must finish his procedure in a fixed time interval, if not it must be killed and restarted as soon as possible.
- **Mutex** – the application thread, who prints data results on the screen, reads from the same memory area accessed by the refresh thread, the mutual exclusion between these two activities must be performed.

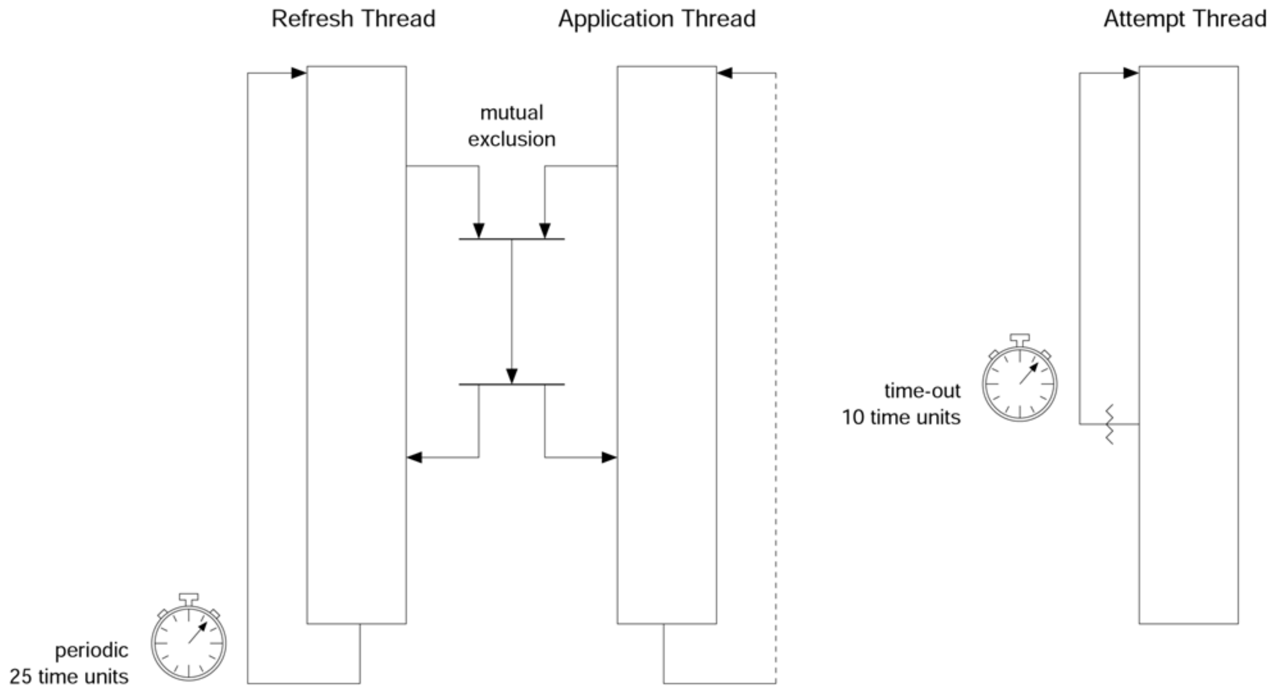

Figure 7 - Control example

The refresh specification (**Specification 1**) express that the start signal to the refresh thread must be given after a fixed number of sample (25) from the previous start signal if everything went well, that is the thread started and it terminated with success:

$$start(ref) \Leftrightarrow \neg start(ref)@[-24,-1] \wedge succ$$
$$succ \Leftrightarrow (\partial \neg running(ref) \wedge running(ref)) \wedge succeeded(ref)$$

Specification 1 - Refresh

where $start(ref)$ is the start signal to the refresh thread and $succ$ evaluates if the previous execution was ok; the literal $running(ref)$ is true when the refresh thread is running and the sub equation in parenthesis is nothing more than a *up* expression, that is it evaluates when the signal raise up.

The specification about killing and restarting the connection attempt thread (**Specification 2**) needs the *since* operator. It times the execution of the thread from the instant it starts and send the kill request if it does not finish in 10 instant and then after it is really terminated send again a start signal:

$$wait_{birth} \Leftrightarrow \mathbf{since}(start(attempt), \neg running(attempt))$$
$$wait_{death} \Leftrightarrow \mathbf{since}(kill(attempt), running(attempt))$$
$$kill_{req} \Leftrightarrow \mathbf{since}(\partial wait_{birth} \wedge \neg wait_{birth}, running(attempt))@[-10,0]$$
$$kill(attempt) \Leftrightarrow \partial \neg kill_{req} \wedge kill_{req}$$
$$start(attempt) \Leftrightarrow (\partial \neg start_{req} \wedge start_{req}) \vee (\partial wait_{death} \wedge \neg wait_{death})$$

Specification 2 – Retry

where the *wait* literals are true when the start signal is send but the thread is not running yet; the $start_{req}$ is an external input that activates the connection attempt, $kill_{req}$ instead is generated automatically and on its raise step is send the kill signal. It has to be notice that every request signal should be sent on a instantaneous literal to ensure that this signal was not sent several times without answer.

The mutual exclusion protocol is performed using two semaphores, but controlled by the specification. These semaphore can tell the controller if a thread is waiting on them and the release signal can be sent only by the behaviour specification.

The interested specification (**Specification 3**) must trace which thread is trying to access to the critical section and send the release signal to allow the thread in. They are needed two semaphores because we have to know when a thread is going out form the section.

$$
\begin{aligned}
signal(ref,enter) &\Leftrightarrow \neg\mathbf{since}(\partial signal(ref,enter), waiting(ref,enter)) \wedge waiting(ref,enter) \\
&\quad \wedge \neg inside(app) \\
signal(app,enter) &\Leftrightarrow \neg\mathbf{since}(\partial signal(app,enter), waiting(app,enter)) \wedge waiting(app,enter) \\
&\quad \wedge \neg inside(ref) \wedge \neg waiting(ref,enter) \\
signal(ref,exit) &\Leftrightarrow \neg\mathbf{since}(\partial signal(ref,exit), waiting(ref,exit)) \wedge waiting(ref,exit) \\
signal(app,exit) &\Leftrightarrow \neg\mathbf{since}(\partial signal(app,exit), waiting(app,exit)) \wedge waiting(app,exit) \\
inside(ref) &\Leftrightarrow \mathbf{since}(signal(ref,enter), \neg signal(ref,exit)) \\
inside(app) &\Leftrightarrow \mathbf{since}(signal(app,enter), \neg signal(app,exit))
\end{aligned}
$$

Specification 3 – Mutex

where the literals with two parameters are related to the semaphore. The literals $inside(ref)$ and $inside(app)$ are parametric but they are determined by the rules and they are true when the related process is working in the critical section. The sub expressions $\neg\mathbf{since}(\ldots)$ were included to avoid sending multiple signal when the thread do not restart immediately.

These parts of the behaviour specification are written in three different files: `refresh.btl`, `retry.btl`, `mutex.btl`. The variables included must be listed in the declaration file `test1.bth`, the following elements are considered:

- **thread ref** – the refresh thread which is responsible of the data-structures refresh;
- **semaphore enter, exit** – the semaphore used in the mutual exclusion for the thread which request to enter or exit;
- **thread app** – the application thread, which is concurrent with the refresh thread;
- **input start_req** – the start signal for the communication attempt;
- **logic refresh(50)** – the logic that is responsible about the periodicity of refresh thread;
- **logic mutex(10)** – the logic that handle the exclusive access to the critical section;
- **logic retry(20)** – the logic that control the time-out condition of the communication attempt.

This declaration must be included in the specification file, in this way all the variables included in the rules are really connected with external devices. This file consider the information about which logic file has to be loaded too; surely this information must be skipped by the compiler and for this it is located inside pre-processor instructions.

Another file must be included to complete the definition of the parametric literals used in the rules; in `thrsem.bth` are defined the new types, thread and semaphore, there are specified the literal prototypes that use thread or semaphore variables.

The specification files must be compiled with *tilcox2tin*, which checks if all the variables used in the rule were previously declared; and they produce three logic file for the real-time execution: `refresh.tin`, `retry.tin`, `mutex.tin`.

The declarations must be processed by *tinconf*, this program creates four C++ files:

- `test1.h` – base header;
- `test1.cpp` – source file that create and link the I/O objects to the logic;

- test1_custom.h – custom header, where is possible to modify the definition of the base classes dedicated to the I/O interface;
- test1_custom.cpp – custom source file, where has to be inserted the evaluation or action procedures, filling empty methods of the I/O classes.

The method toRun must contain the run code of the thread, inside it is possible to insert the waiting commands for the semaphores; for example, the resource usage simulation of the refresh thread is illustrated in **Code 1** as the customisation of the method eval of an Input device.

```
io_val Input_start_req::eval()
{
//key press Input

if(kbhit())
if(getch()=='1')
return(IO_TRUE);
//values read from the logic
return(IO_FALSE);
}



bool Thread_ref::toRun()
{
//Something to refresh
unsigned long to_wait;

printf("%ld\n",getTickCount());
printf("Waiting for IN\n");

//access request to the critical section
waitOnSem(enter);

//granted...
printf("Let's go!!!\n");

//resource usage simulation
to_wait=(unsigned long)(rand()/100);
printf("Refreshing this... %ld\n",to_wait);
sleep(to_wait);

//resource usage simulation
to_wait=(unsigned long)(rand()/100);
printf("Refreshing that... %ld\n",to_wait);
sleep(to_wait);

printf("Waiting for OUT\n");
//notifying exit from the critical section
waitOnSem(exit);
printf("Bye bye!!!\n\n");

//succeeded flag
return(true);
}
```

Code 1 - Customisation

After this editing step, the controller is ready to be formed by linking all the code files with the library. In **Figures 8, 9** are illustrated some snapshots from the controller run-time execution.

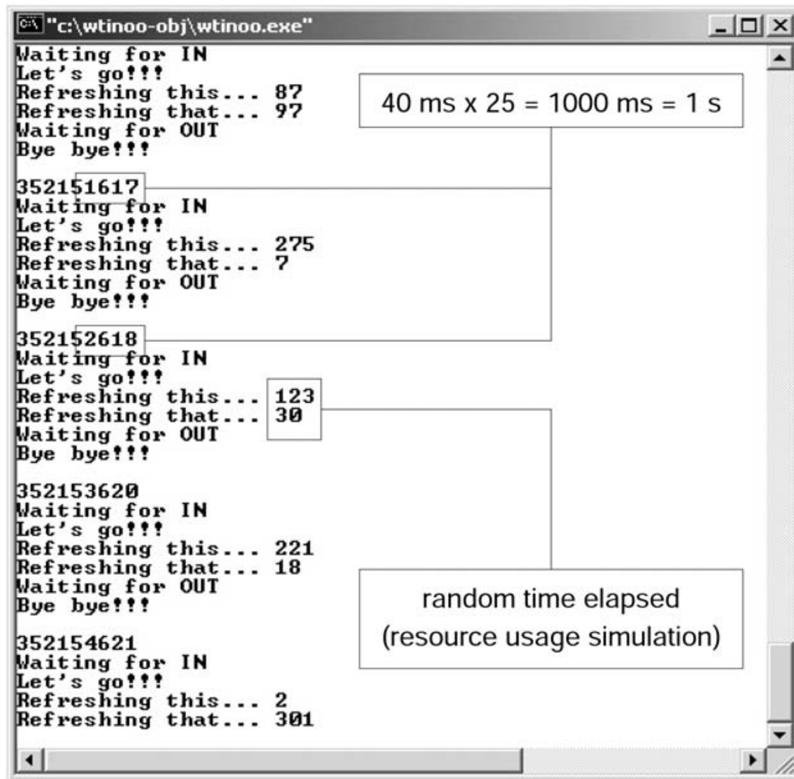

Figure 8 - Periodicity

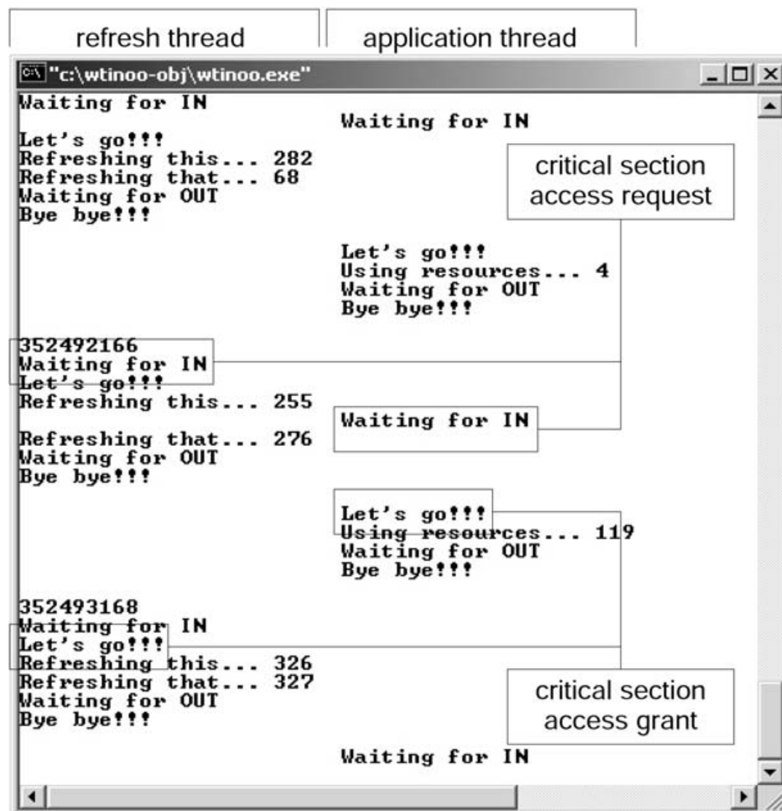

Figure 9 - Mutual exclusion

The aim of this project is to ensure a rigorous approach to the process control task especially for the synchronization objectives; taking confidence with TILCO-X as formal language, should be much easier to develop real-time controller. All the relationships between events are described (not implemented) in the specification. It is possible to proof that what is written in rules satisfies all the safeness requirements; this reduces a lot the reliability problem in developing real-time system.