IDE for real-time system: Temporal logic and C++

A Controller for crossroad signals: Traffic-Light Simulator

Introduction

The aim of this technical report is to describe the whole experience that has been collected during our first attempt in developing a "real" system with a new IDE, which integrates declarative and imperative programming. The IDE allows building a C++ project; in this project a new kind of source can be integrated: TILCO temporal logic files can be added and compiled into the project and they take care of time relationship between events.
This report explains each step of this development process and finally it analyzes how complete is the IDE and which procedures should be more quick and intuitive.

Crossroad traffic environment

The crossroad traffic devices considered in this simulation are shown in **Figure 1**:

- 2 roads
- 4 traffic-lights (2 direction x 2 roads)
- 1 disable switch
- 2 emergency requests (one for each road)
- 4 traffic panel (as the traffic-lights)

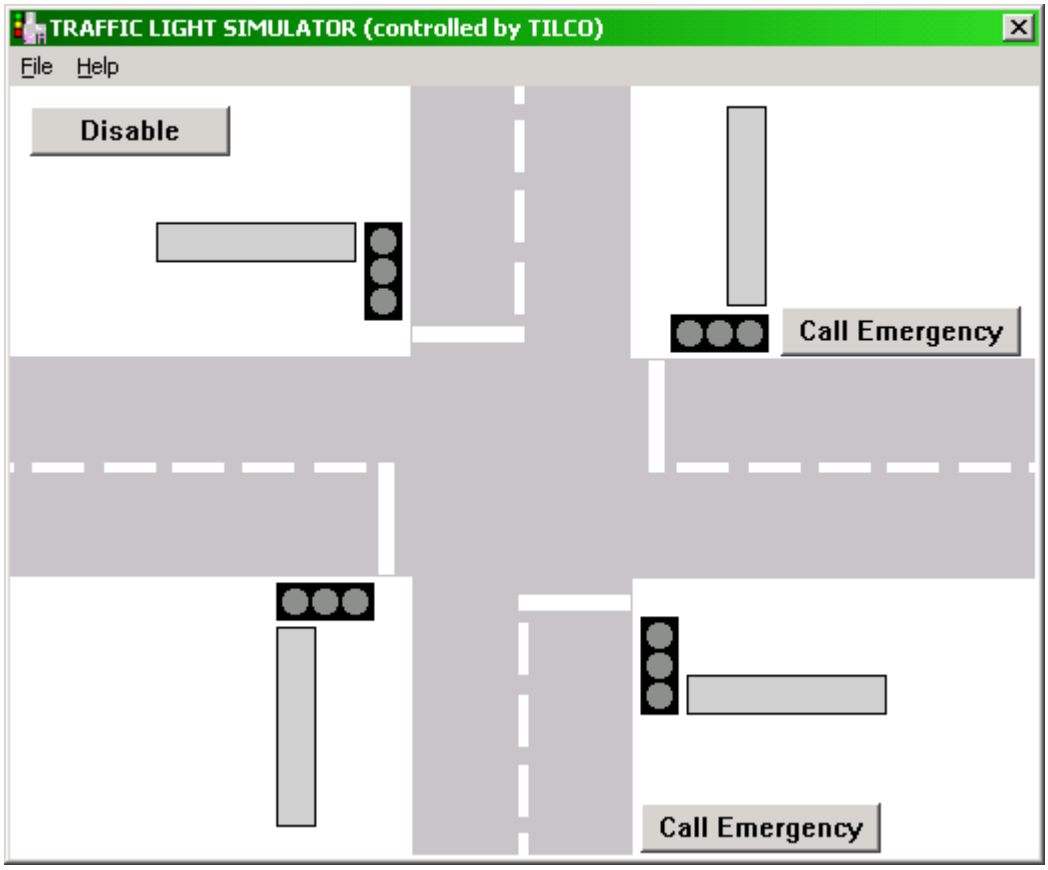

Figure 1: Crossroad signals

Specification of a traffic-light crossroad behavior

To specify the behavior of crossroad signals is not so natural as it likes. A crossroad traffic light system is typically realized with a simple state-machine. The state sequence of the car flow in the crossroad is the base of the traffic behavior. Some additional features have been added to make this workbench complex enough to measure meaningful evaluations on development system properties.

Informal description of a crossroad traffic light system

1 The car flow alternates between the two roads and the nominal cycle of a traffic light follow the sequence
   - green
   - yellow
   - red
   - red-yellow

2 If the system is disabled the lights start to blink on the yellow color

3 If a emergency condition is assumed on *road A*, the light sequence run as soon as possible on green color to *road A*, while respecting safety conditions

4 The traffic panel must show commercial spot in a road only when the car flow is stopped in the same road, otherwise it communicates information about traffic situation nearby the crossroad.

1st Step: Create a state sequence

First of all, during normal mode, the traffic environment follows a state sequence. In the analysis various traffic scenarios have been recognized and symbolic names, based on the car flow (go, warning, alert, stop), are assigned to the corresponding states. A complete cycle of traffic signals executes this sequence alternatively between two roads.
In the specification the state order and time constants for each state should be expressed. These parameters should be clearly visible in the rules and they should not appear more than once in the specification, some rule examples are illustrated in **Specification 1** and **Specification 2**.

$$go_A \Leftrightarrow \text{since}(up\_go_A, \neg up\_warning_A)$$
$$warning_A \Leftrightarrow \text{since}(up\_warning_A, \neg up\_alert_A)$$
$$alert_A \Leftrightarrow \text{since}(up\_alert_A, \neg up\_stop_A)$$
$$stop_A \Leftrightarrow \text{since}(up\_stop_A, \neg up\_go_B)$$
$$go_B \Leftrightarrow \text{since}(up\_go_B, \neg up\_warning_B)$$
…

"$state_{ROAD}$ **is true** since its activation signal up_state_{ROAD} becomes true, until activation signal of the next state in the sequence $up\_nextstate_{ROAD}$ is false"

Specification 1: state order

$$up\_go_A \Leftrightarrow stop_B\,@[-\text{STOP\_TIME}, 0)$$
$$up\_warning_A \Leftrightarrow go_A\,@[-\text{GO\_TIME\_A}, 0)$$
$$up\_alert_A \Leftrightarrow warning_A\,@[-\text{WARNING\_TIME}, 0)$$
$$up\_stop_A \Leftrightarrow alert_A\,@[-\text{ALERT\_TIME}, 0)$$
$$up\_go_B \Leftrightarrow stop_A\,@[-\text{STOP\_TIME}, 0)$$
…

"activation signal of next state $up\_nextstate_{ROAD}$ **is true** after $state_{ROAD}$ has been true for all the duration of its time constant"

Specification 2: state time constants

2nd Step: Interface state with boolean light signals

Traffic lights do not accept states like "go_A" (free way for cars in road A) as control signal, but, considering every state, it is possible to determine the traffic light configuration for each road like "green" or "red-yellow".
After mapping light configurations to flow states it is necessary to map these configurations to boolean signals for each single light element in the traffic light, so as to determine which lights are "on" or "off". For example, in the red-yellow configuration red light and yellow light are on instead of green that is turned off.
Some rules of the two phases state-mapping are illustrated in **Specification 3** and **Specification 4** and the normal sequence of the lights is shown in **Figure 2**.

$green_A \Leftrightarrow go_A$
$yellow_A \Leftrightarrow warning_A \vee alert_A$
$red_A \Leftrightarrow stop_A \vee go_B \vee warning_B$
$red\_yellow_A \Leftrightarrow alert_B \vee stop_B$
$green_B \Leftrightarrow go_B$
…

"configuration $green_A$ **is true** if and only if the go_A state is true"
"configuration $yellow_A$ **is true** if and only if $warning_A$ state or $alert_A$ state are true"

Specification 3: car flow to traffic light configuration

$green\_signal_A \Leftrightarrow green_A$
$yellow\_signal_A \Leftrightarrow yellow_A \vee red\_yellow_A$
$red\_signal_A \Leftrightarrow red_A \vee red\_yellow_A$
$green\_signal_B \Leftrightarrow green_B$
…

"light signal $green\_signal_A$ is true if and if only the $green_A$ configuration is true"

Specification 4: traffic light configuration to light boolean signal

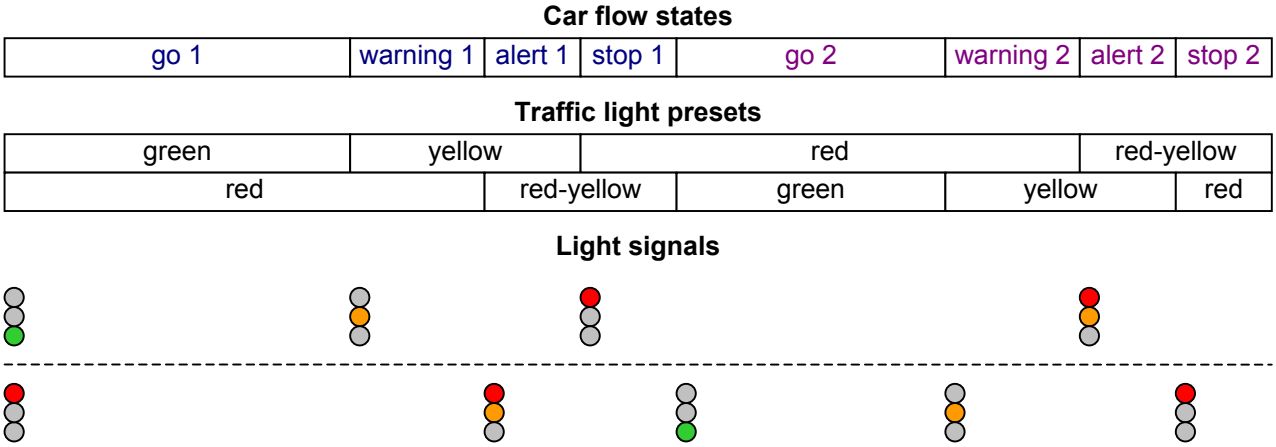

Figure 2: States to output signals mapping

3rd Step: Exceptional conditions

The development of a real-time system, which simply performs nominal sequence of the traffic signals, is a trivial result, but the safe handling of exceptional conditions, by just extending the behavior specification, is much more interesting.
In this step some exit departures from normal sequence are considered; for example the disable request should be satisfied only when both car flows are stopped: it is possible to switch on disable mode at the end of a "stop" state.
After an emergency request the aim is to make the wait period of a vehicle on the road interested by the request as short as possible, while granting all the safety conditions like a minimum time for the "go" state in the other road (as is written in **Specification 5**).

$$up\_warning_A \Leftrightarrow (go_A @[-\text{GO\_TIME\_A}, 0) \wedge \neg emergency\_req_A) \vee (go_A @[-\text{GO\_MINIMUM\_TIME}, 0) \wedge \neg emergency\_req_A \wedge emergency\_req_B)$$

"activation signal of $warning_A$ state **is true** after go_A has been true for its normal duration and no request has been issued, or go_A has been true for its minimum time constant and an emergency request is issued by road B without road A preemption"

Specification 5: emergency request satisfaction

4th Step: Thread panel control

Display panels have been included into this system exclusively to explain how easy is to perform thread control with this IDE. Panels show something active, which moves or scrolls. If the implementation of scrolling is demanded to a dedicated thread; the start and the kill signal has to be sent to such thread depending on the car flow, taking in account the informal specification. For instance, a kill request is sent to the thread that runs commercial spots when the car flow restarts on the road; this behavior is illustrated in **Specification 6**.

$$kill(commerce_A) \Leftrightarrow \text{since}(up\_alert_B \vee up\_disabled, running(commerce_A))$$

$$start(info_A) \Leftrightarrow \neg disabled \wedge \text{since}(\text{down}(kill(commerce_A)), kill(info_A))$$

"a kill request to the thread, which is responsible of showing commercial spots on road A, is sent since the $alert_B$ or $disable$ activation signal are true until it is satisfied"
"a start request is issued to the other thread, which shows traffic information, if disable mode is off and since the previous kill request has succeeded until this thread receives a stop request"

Specification 6: thread activation and deactivation

Complete Specification

The whole set of rules is presented in **Source 1** and **Source 2**.
Beside some TILCO symbols conversion to stardard ASCII codes, like $\wedge$ to &, some more explanations about to write a specification file are needed:

- ✓ **header files** – The specification starts with some preprocessor instructions with the aim to include some libraries, which contain definitions and rule macros. The `signal.bth` header contains operators "up" and "down" definitions; the `thrsem.bth` file is richer and includes all the instruments needed to handle thread and semaphore functions. The last include statement is expected by the development system and declare the I/O signals that interface C++ environment.
- ✓ **time constants** – This section improves the readability of the rules about the state sequence by defining a set of mnemonic constants, which represent states duration and make easier modifying the sequence timing.
- ✓ **state sequence and triggers** – the *switch* literal is introduced to drive the system in the disable mode, if it has been requested, after a *stop* state. The *disable_control* is a gate for the *disable_request* signal: while emergency situations are active it is ignored, otherwise it causes the *disable* state entering; when such state is exited the system restarts the sequence from a *stop* state.
- ✓ **thread management** – the specification source for thread control is slightly different from what has been presented above: *safe_start* and *safe_kill* functions have been defined in the thread control library (`thrsem.bth`), so to allow keeping true a thread activation signal until the activation occurs while granting that the actual activation request (*start* or *kill* function) is bounded within a single time instant.
- ✓ **simulation** – the specified system simulation can be accomplished by giving a description of the thread behavior. The *simulated_thread* rule fits for this purpose by expressing reasonable hypotheses about the *running* state history depending on *start* and *kill* functions. With this addiction the specification becomes testable in off-line mode.

Some validations of the specified behavior

The specification is the most important source of this project; it rules every time relationship in the real-time system. This aspect has migrated in this new type of source code because using an operational style is quite hard to ensure a correct solution or to debug it when it does not work. Two steps of validation are presented below.

Properties Proof: State sequence

This case study is a typical state machine example, its behavior is based on a set of rules that specify a state sequence. Since such a general problem has been faced, an open solution, like a template, could be presented for state-sequence-like behavior.
The interested rule set should be formally validated. The property about activation of one and just one of the states in the sequence in this example has been mathematically prooved for each time instant. Such property is presented in **Specification 7**; the state sequence template considered for this proof does not use any input signal; its evolution depend only by initial state. For this reason it has been proved if the condition is respected at time *t* is the same for *t+1* instant.

$just\_one\_state \Leftrightarrow$ $(\mathbf{go}_{\mathbf{A}} \wedge \neg warning_A \wedge \neg alert_A \wedge \cdots) \vee$ $(\neg go_A \wedge \mathbf{warning}_{\mathbf{A}} \wedge \neg alert_A \wedge \cdots) \vee \cdots$ $just\_one\_state \Rightarrow just\_one\_state@1$	"property checks if just one of the states in the normal sequence is active. "property is valid for each time instant."

Specification 7: state sequence property proof

History checking: Response to input signals

This validation step is more "practical", history checking produces output signal histories from input ones, which are given. This simulation of the real-time behavior is made off-line and in this case an "ad hoc" tool , like *SgnEdit*, is used. As it has been said above, the behavior is not completely defined, because it depend on input signals like *running* functions that watch thread reaction to the *start* or *kill* request. The specification has been completed for history checking test adding *simulated_thread* rules (macros defined in `thrsem.bth`).
The aim of this testing is to confirm the expected response which is driven from input signals like disable or emergency requests. The histories of this signal are coded in simple input files, which are processed by the executable *tinx*. This application taking into account the compiled specification produces the respective output histories. The visualization tool (a snapshot is illustrated in **Figure 3**) allows to modify and easily watch the I/O behaviour

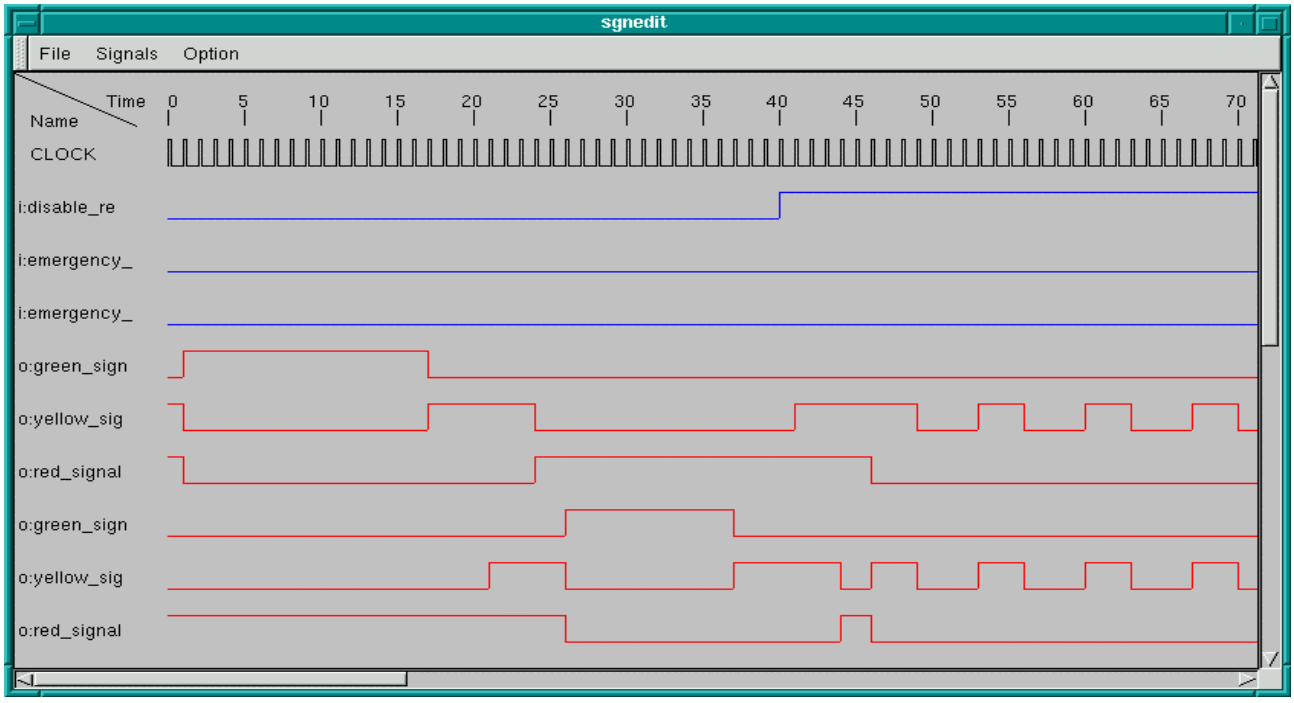

Figure 3: History checking snapshot (disable request satisfaction)

```
// Headers

#include "signal.bth"
#include "thrsem.bth"

#ifndef SIMULATION
#include "../src/traffic_light.bth"
#endif

// Constants

#define GO_TIME_1            16
#define GO_TIME_2            11
#define EMERGENCY_GO_TIME    5
#define WARNING_TIME         4
#define ALERT_TIME           3
#define STOP_TIME            2
#define BLINK_TIME           3
#define BLINK_RATE           6

// State sequence

aux    go_1, alert_1, warning_1, stop_1, up_go_1, up_warning_1, up_alert_1, up_stop_1, switch_1,
       go_2, alert_2, warning_2, stop_2, up_go_2, up_warning_2, up_alert_2, up_stop_2, switch_2;

init   ~go_1 @ 0, ~warning_1 @ 0, ~alert_1 @ 0, stop_1 @ 0,
       ~go_2 @ 0, ~warning_2 @ 0, ~alert_2 @ 0, ~stop_2 @ 0, switch_1 @ 1;

go_1 == since(up_go_1, ~up_warning_1);
warning_1 == since(up_warning_1, ~up_alert_1);
alert_1 == since(up_alert_1, ~up_stop_1);
stop_1 == since(up_stop_1, ~switch_1);

go_2 == since(up_go_2, ~up_warning_2);
warning_2 == since(up_warning_2, ~up_alert_2);
alert_2 == since(up_alert_2, ~up_stop_2);
stop_2 == since(up_stop_2, ~switch_2);

// State triggers

#ifdef SIMULATION
input emergency_request_1, emergency_request_2, disable_request;
#endif

aux disable_control, disabled, up_disabled, down_disabled;

init   ~disabled @ 0;

disabled == since(up_disabled, disable_control);
up_disabled == (switch_1 | switch_2) & disable_control;
down_disabled == down(disabled);
disable_control == disable_request & ~emergency_request_1 & ~emergency_request_2;

up_go_1 == switch_2 & ~disable_control;
up_warning_1 == (go_1 @ [-GO_TIME_1, 0) & ~emergency_request_1) |
              (go_1 @ [-EMERGENCY_GO_TIME, 0) & ~emergency_request_1 & emergency_request_2);
up_alert_1 == warning_1 @ [-WARNING_TIME, 0);
up_stop_1 == alert_1 @ [-ALERT_TIME, 0) | (down_disabled  & emergency_request_2);
switch_1 == stop_1 @ [-STOP_TIME, 0);

up_go_2 == switch_1 & ~disable_control;
up_warning_2 == (go_2 @ [-GO_TIME_2, 0) & ~emergency_request_2) |
              (go_2 @ [-EMERGENCY_GO_TIME, 0) & emergency_request_1);
up_alert_2 == warning_2 @ [-WARNING_TIME, 0);
up_stop_2 == alert_2 @ [-ALERT_TIME, 0) | (down_disabled & ~emergency_request_2);
switch_2 == stop_2 @ [-STOP_TIME, 0);
```

Source 1: `traffic.btl` – part 1

```
// Disabled state behaviour

aux    yellow_blink, blink_tick;

blink_tick == (disabled & ~blink_tick @ [-BLINK_RATE, 0)) | up_disabled;
yellow_blink == disabled & (blink_tick ? (-BLINK_TIME, 0]);

// Crossroad to semaphore state map

aux    green_1, yellow_1, red_1, red_yellow_1,
       green_2, yellow_2, red_2, red_yellow_2;

green_1 == go_1;
yellow_1 == warning_1 | alert_1;
red_1 == stop_1 | go_2 | warning_2;
red_yellow_1 == alert_2 | stop_2;

green_2 == go_2;
yellow_2 == warning_2 | alert_2;
red_2 == stop_2 | go_1 | warning_1;
red_yellow_2 == alert_1 | stop_1;

// Light signals generation

#ifdef SIMULATION
output  red_signal_1, yellow_signal_1, green_signal_1,
       red_signal_2, yellow_signal_2, green_signal_2;
#endif

green_signal_1 == green_1;
yellow_signal_1 == yellow_1 | red_yellow_1 | yellow_blink;
red_signal_1 == red_1 | red_yellow_1;

green_signal_2 == green_2;
yellow_signal_2 == yellow_2 | red_yellow_2 | yellow_blink;
red_signal_2 == red_2 | red_yellow_2;


// Thread management

#ifdef SIMULATION
thread  info_1, info_2, commerce_1, commerce_2;
#endif

safe_thread(info_1);
safe_thread(info_2);
safe_thread(commerce_1);
safe_thread(commerce_2);

#ifdef SIMULATION
simulated_thread(info_1, 2);
simulated_thread(info_2, 2);
simulated_thread(commerce_1,2);
simulated_thread(commerce_2, 2);
#endif

init   ~safe_start(info_1) @ 0, ~safe_kill(info_1) @ 0,
       ~safe_start(commerce_1) @ 0, ~safe_kill(commerce_1) @ 0;

init   ~safe_start(info_2) @ 0, ~safe_kill(info_2) @ 0,
       ~safe_start(commerce_2) @ 0, ~safe_kill(commerce_2) @ 0;

safe_kill(info_1) == since(up_stop_1 | up_disabled, running(info_1));
safe_kill(info_2) == since(up_stop_2 | up_disabled, running(info_2));
safe_kill(commerce_1) == since(up_alert_2 | up_disabled, running(commerce_1));
safe_kill(commerce_2) == since(up_alert_1 | up_disabled, running(commerce_2));

safe_start(commerce_1) == ~disabled & since(down(safe_kill(info_1)), ~safe_kill(commerce_1));
safe_start(commerce_2) == ~disabled & since(down(safe_kill(info_2)), ~safe_kill(commerce_2));
safe_start(info_1) == ~disabled & since(down(safe_kill(commerce_1)), ~safe_kill(info_1));
safe_start(info_2) == ~disabled & since(down(safe_kill(commerce_2)), ~safe_kill(info_2));
```

Source 2: `traffic.btl` – part 2

Controller development

A real-time controller for the traffic light signals has to drive directly all I/O terminals of the connected system. With this new IDE, as soon as the specification is written and compiled, the I/O devices, which are needed to interface the traffic signals with the controller, remain to be declared. After the I/O elements list has been created, the automatic configurator of the IDE generates code for device linking with the inference kernel. The controller is completed by editing the code of the I/O classes just created to embed the I/O devices, controlled by the specification, into the real system.

The development steps are summarized in **Figure 3**.

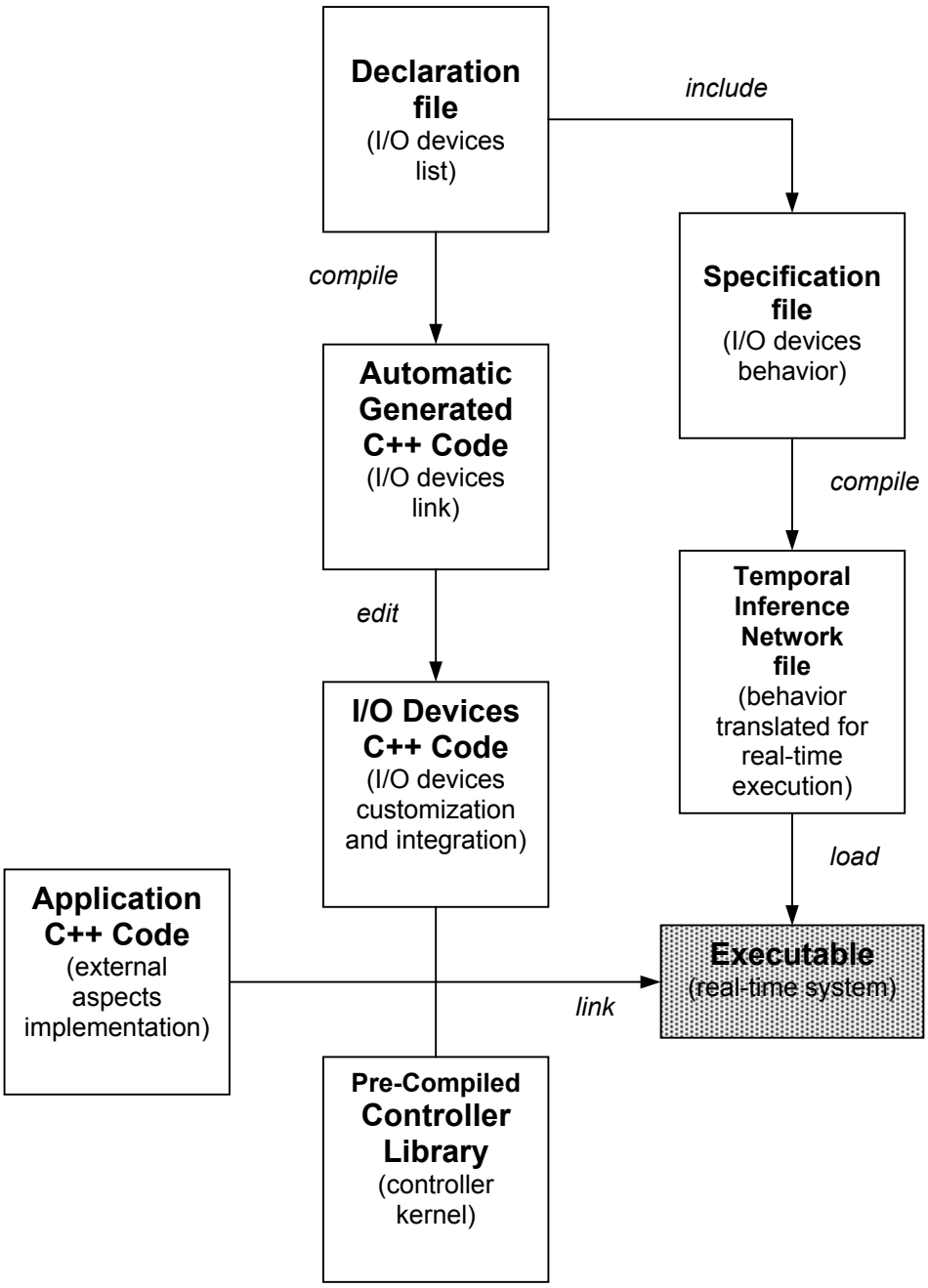

Figure 4: sources-to-executable development steps

1st Step: List the I/O devices

The I/O terminals in this case study are responsible of controlling light switches and the active text panels.
The exhaustive list is presented in **Table 1**; it is the starting point of the development system and influences, taking into account types of declaration, the automatic configuration code written in C++ source, where class names and instances are defined.

Input	Disable_request	
	Emergency_request_1	emergency_request_2
Output	green_signal_1 yellow_signal_1 red_signal_1	green_signal_2 yellow_signal_2 red_signal_2
Thread	info_1 commerce_1	info_2 commerce_2
	Road 1	**Road 2**

Table 1: I/O devices for the traffic signals controller

2nd Step: Customize the devices C++ code

The executable produces C++ files, source and headers, which are composed by:
1. declarations of the I/O classes;
2. link instructions between devices and inference kernel;
3. empty methods which have to be edited for customizing the devices interface;

Each kind of device has a empty method that has to be filled to make the controller, based on logic inference, to perform I/O communication with the real system:
- for an **Input** device the method `io_val eval()` perform the evaluations needed to signal a true, false or none event to the controller;
- an **Output** device, drives the system through a `void action(bool val)`method;
- if a **Thread** is used to create a system thread which is totally under controller directives, the `bool toRun()` procedure performs the operations when this is running.

The automatic generation of code creates a different class for each I/O element of traffic system, but is possible to group I/O element that perform the same action, but on a different component of the system.
In this case, three categories of I/O have been created:
- LightOutput,
- ReqInput,
- PanelThread.

This feature of the IDE avoid code duplication, by using middle-level classes, which are created to interface between Input, Output or TINThreadWithSemaphores (basic class for I/O elements) and final class like Input_disable-request, Output_green_signal_1 , Thread_info_1, which are responsible of singular characteristics of the single I/O elements. In this situation just one method for each kind of I/O class is specified. The category classes provides to contain the references to the element which they operates on, as example a LightOutput contains two Light class pointer that links the device with two "real" lights of the traffic system.

3rd Step: Integrate devices into application C++ code

The devices are instantiated by the automatic configuration code, but before running the logic-based controller it is needed to make them get the right entry point into the real system. Such "glue" code is a delicate part of the project, should be short and easily understandable. For example the object *Output_green_signal_1 green_signal_1* has to set his Light class pointers to the two Light instances (the "green" ones) owned by the traffic lights on road 1 (two opposite verses). Therefore two class derived from PanelThread, like *Thread_info_2 info_2* and *Thread_commerce_2 commerce_2*, refer to the same two panels on road 2.

Development Sources Overview

All the resources needed to built this real-time system are presented in **Source 3**, **Source 4**, **Source 5**, **Source 6**.

traffic_con.bth

The I/O list is the starting point of this development process and it is presented in **Source 3**. This list is the warranty of consistence in the project, it ensures what is specified in the left side of the flow diagram shown in **Figure 4** does found

the respective element on the other side; as example the the *disable_request* input presented in the specification (`traffic.btl`) must corresponds to an input device instantiated and automaticly linked by the C++ code generated in `traffic_con.cpp`.
Additional information are provided in the declarative source of the I/O items, such as the inference network file (`traffic.tin`), which is loaded at run-time from the controller and it has been created by the *tilcox2tin* compiler directly form the specification file. This compiler must not process this information, which are useful for the code generating step; in fact they are written inside pre-processor structure: in this way these lines are skipped by the *tilcox2tin* compiler and considered by the *tinconf* code generator.

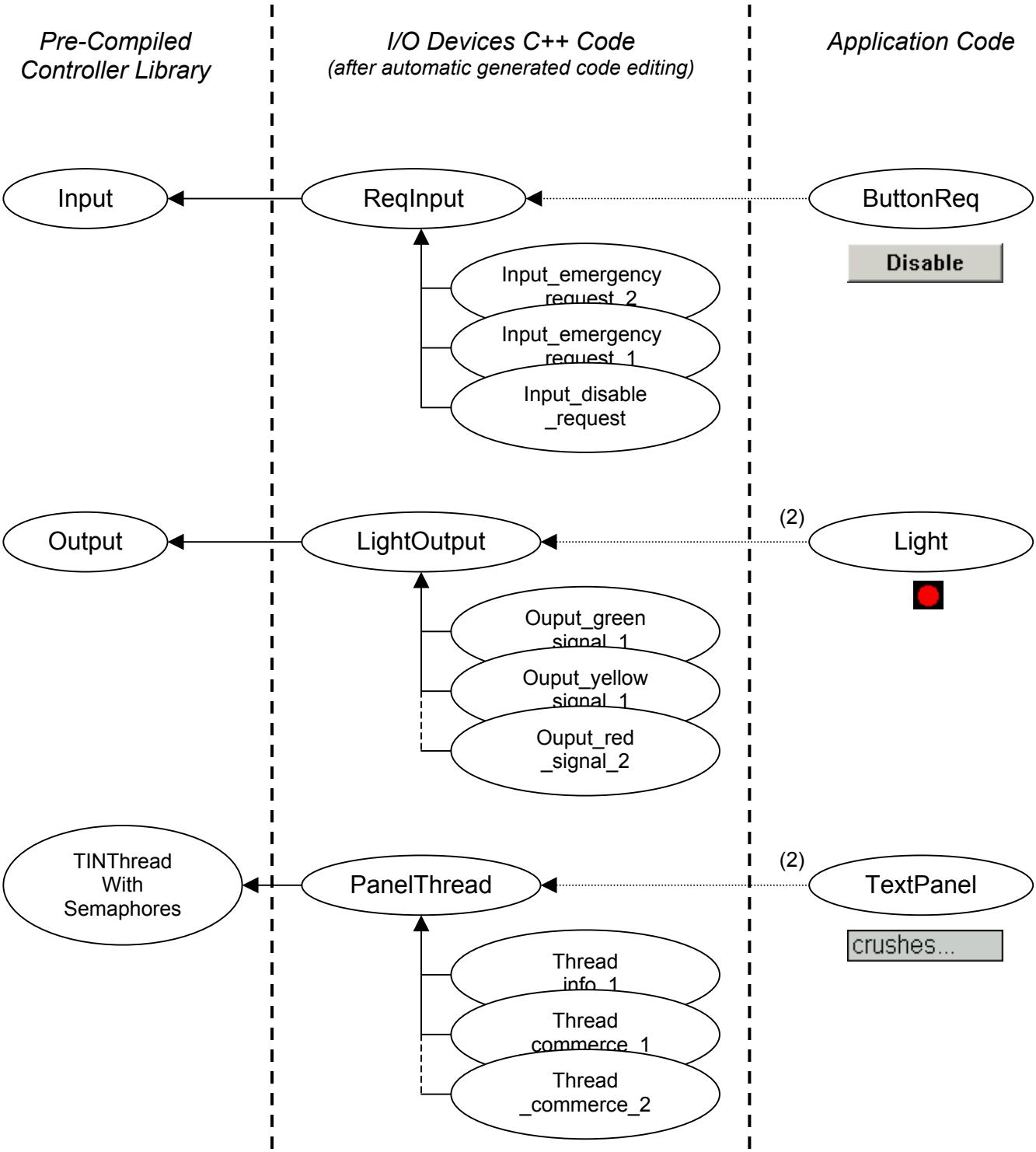

Figure 5: Class Diagram for the devices customization (with module separation)

traffic_con_custom .h/.cpp

In **Source 4** and **Source 5** some C++ code is presented; Some classes, which model I/O items, are defined and implemented.
This code is the result of automatic generation of the classes and editig for customization; for this applications has been edited both interface and implementation. Some methods, needed to link the instance to the application, have been defined in the interface of each class, like *void linkLight1(Light *link)* for LightOutput class. Anyway the implementation is responsible of the real "customization"of each I/O item: the *void action(bool val)* method for the LightOutput class rules which operations follow a true or false value of specification literal like *green_signal_1* or *yellow_signal_2*. It is intuitive that in *bool toRun()* method of PanelThread class are the instruction executed after the beahvior determines a thread start.

traffic_sim.cpp

The last interesting code section is presented in **Source 6**, which is part of the final application. In this source the I/O items are connected with the request, light or panel instances of the application to actively change the state of the traffic signals. An object diagram is drawn in **Figure 6** to explain how this link allows a direct control of the traffic by evaluating events in the inference engine.
This IDE do not allow to manipulate class instances which are automatically generated as global without previously declaring as external in the source file, this declaration get away from accidental erroneous manipulation.

```
input       disable_request;

input       emergency_request_1,
            emergency_request_2;

output      green_signal_1,
            yellow_signal_1,
            red_signal_1;

output      green_signal_2,
            yellow_signal_2,
            red_signal_2;

thread      info_1,
            info_2;

thread      commerce_1,
            commerce_2;

#ifndef __LOGIC

logic       traffic(50);

#endif
```

Source 2: `traffic_con.bth` (list of I/O items)

Executable creation in a common development system

In the IDE it is possible to build the executable file and his resource at the same time. Using simple pre-build step it is possible to compile the specification file (if it is changed) to obtain a TIN file and an initialization file; the first file describe the inference net and it is loaded from the controller at run-time together with the other event list that is processed by the controller before the system starts.
In the same manner a declaration change is recognized by the IDE; it generates again the C++ sources, but only the files that are not editable to protect those important modifications.
In the build step the source file are compiled and linked whit the kernel library to create an executable that perform a real-time system where all the time constant and event relations are formally specified.

```
//Custom output class to control 2 light (of the same color)
class LightOutput : public Output
{
private:
      Light *light1;
      Light *light2;
public:
      LightOutput(const char *name) : Output(name) { light1=light2=NULL;}
      void action(bool val);
      void linkLight1(Light *link) { light1=link; }
      void linkLight2(Light *link) { light2=link; }
};

class Output_green_signal_1 : public LightOutput
{
public:
    Output_green_signal_1() : LightOutput("green_signal_1") {}
};

class Output_yellow_signal_1 : public LightOutput
{
public:
    Output_yellow_signal_1() : LightOutput("yellow_signal_1") {}
};

                                    … < missing part > …

//Custom thread class to control 2 traffic panel
class PanelThread : public TINThreadWithSemaphores
{
private:
      TrafficPanel *panel1;
      TrafficPanel *panel2;
      char scrolltext[100];
      char scrolled[200];

      bool toRun();
public:
      PanelThread(const char *name) :
TINThreadWithSemaphores(name,MAX_SEMS,SAMPLE_TIME) {}
      void linkPanel1(TrafficPanel *link) { panel1=link; }
      void linkPanel2(TrafficPanel *link) { panel2=link; }
      void setScrollText(const char *text);
};

class Thread_info_1 : public PanelThread
{
public:
    Thread_info_1() : PanelThread("info_1")
      { setScrollText("traffic info, jam, crushes...  "); }
};

class Thread_commerce_1 : public PanelThread
{
public:
    Thread_commerce_1() : PanelThread("commerce_1")
      { setScrollText("commercial spot, buy that, buy everything...  "); }
};
```

Source 3: `traffic_con_custom.h` (I/O items interface)

```
io_val ReqInput::eval()
{
	if(!req)
		return(IO_NONE);
	if(req->getState())
		return(IO_TRUE);
	return(IO_FALSE);
}

//Action of a custom Ouput device created to control a light
void LightOutput::action(bool val)
{
	light1->switchLight(val);
	light2->switchLight(val);
}

//Running code of a custom Thread created to scroll a text on a panel
bool PanelThread::toRun()
{
	int lenght, cursor;

	strcpy(scrolled,scrolltext);
	strcat(scrolled,scrolltext);
	lenght=strlen(scrolltext);
	cursor=0;
	while(true)
	{
		panel1->setText(scrolled+cursor);
		panel2->setText(scrolled+cursor);
		cursor=(cursor+1)%lenght;
		sleep(200);
	}
	return(true);
}
```

Source 4: `traffic_con_custom.cpp` (I/O items implementation)

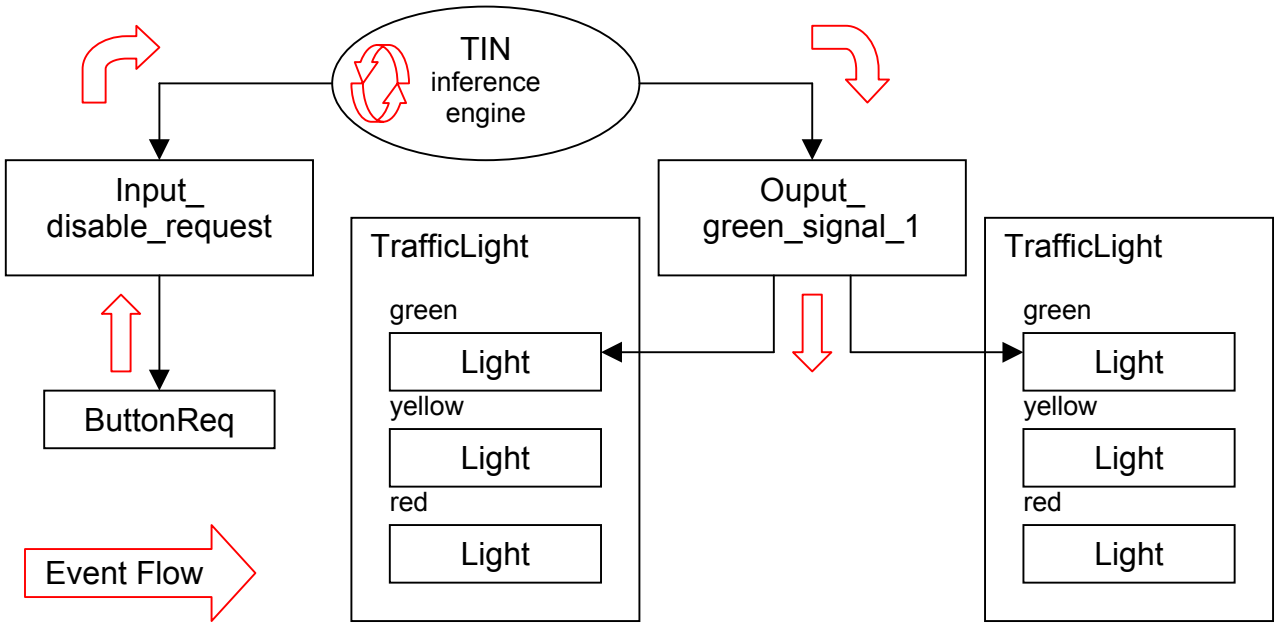

Figure 6: Object diagram of inference-system links

```
// Getting global IO device declared
// in the automatic generated code by TINCONF

// GETTING INPUT
extern Input_disable_request disable_request;
extern Input_emergency_request_1 emergency_request_1;
extern Input_emergency_request_2 emergency_request_2;

// GETTING OUTPUT
extern Output_green_signal_1 green_signal_1;
extern Output_green_signal_2 green_signal_2;
extern Output_yellow_signal_1 yellow_signal_1;

                                    … < missing part > …

// GETTING THREAD
extern Thread_info_1 info_1;
extern Thread_info_2 info_2;
extern Thread_commerce_1 commerce_1;
extern Thread_commerce_2 commerce_2;

                                    … < missing part > …

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{

                                    … < missing part > …

      //Calling config() function
      //...this provides to link all the IO instances to the logic kernel
      config();

      //Linking the real environment to the I/O items

      // INPUT LINKING
      disable_request.linkReq(&disableReq);
      emergency_request_1.linkReq(&emergency1Req);
      emergency_request_2.linkReq(&emergency2Req);

      // OUTPUT LINKING
      green_signal_1.linkLight1(Road1F.linkGreen());
      green_signal_1.linkLight2(Road1B.linkGreen());

      green_signal_2.linkLight1(Road2F.linkGreen());
      green_signal_2.linkLight2(Road2B.linkGreen());

      yellow_signal_1.linkLight1(Road1F.linkYellow());
      yellow_signal_1.linkLight2(Road1B.linkYellow());

                                    … < missing part > …

      // THREAD LINKING
      info_1.linkPanel1(&Panel1F);
      info_1.linkPanel2(&Panel1B);

      commerce_1.linkPanel1(&Panel1F);
      commerce_1.linkPanel2(&Panel1B);
```

Source 5: `traffic_sim.cpp` (application code)

IDE for real-time system: Temporal logic and C++

A controller for crossroad signals: Traffic-Light Simulator

Summary