POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Visual approach to design and development of event-based software for industrial applications

Supervisor

Prof. Massimo PONCINO

Candidate

Gaetano GALASSO

Thesis Tutor

Dott. David BENEDETTI

April 2022

Abstract

The goal of this thesis is to develop a software capable of handling event-based systems[1], providing the user the ability of modelling them thanks to an easy-to-use Graphical User Interface and, then, allowing to generate final code representing the system, which is editable by the user, if needed. The software, called GEM (Generic Event-based Modeling), exploits the framework developed by the Zirak s.r.l.[2] company, which provides the APIs to deal with an event-based system: it allows the user to establish policies for event signalling, event dispatching and event handling behaviour(s). The tool chosen to design and develop the software is the Qt framework[3], given its C++ integration and its academic-use license, but also for its ability to ease the handling of GUIs for software and its ability to connect graphic and business worlds easily, through the usage of design facilities, such as signal/slot paradigm. Currently, the developed User Interface does not provide all the features offered by the GEM framework, but it will be possibly updated later.

Table of Contents

List of Figures

Acronyms

ADT
Abstract Data Type

CAL
Code Abstraction Layer

FSM
Finite State Machine

GEM
Generic Event-based Modeling

GUI
Graphical User Interface

IoT
Internet of Things

OS
Operating System

PDA
Push-Down Automata

POD
Plain Old Data

REST
REpresentational State Transfer

Chapter 1

Introduction

Nowadays, the software design has become very important for several industrial fields and applications, since software can be the final product or the way to reach a goal. All the industrial fields use software, from automotive to aerospace, from gaming to building, and that is the reason why, more than 50 years ago, the *Software Engineering*[4] born: its the principles apply on software development, and in particular to design, develop, maintain, test and evaluate software.
Software engineering has been devised to deal with issues of low-quality software projects. Problems arise when a software does not respect timelines, costs, and some levels of quality. Software Engineering ensures that the application is made consistently, correctly, on time, respecting costs and within requirements. The software engineering demand also emerged to provide for the large rate of change in user requirements and environment on which application can be work.
A software product is rated on how easily it can be used by the end-user and the functionalities it offers. An application must achieve in the following areas:

- **Operational:** it tells how well a software works on operations like correctness, functionality, efficiency, budget, usability, dependability and security.
- **Transitional:** it is fundamental when an application is shifted from one platform/environment to another. So, portability, adaptability and reusability are placed in this field.
- **Maintenance:** it specifies how well an application works in the changing environment, so maintainability, modularity, scalability and flexibility belong to this area.

Furthermore, to fit with all the areas above, Software development lifecycle needs to be respected: it is a steps list in software engineering to develop software application. They can be grouped into:

- **Software requirements**
  It is about the listing, analysis, specification and validation of software requirements. They can be of three types: functional, non-functional and domain requirements. The first set indicates what the software should perform in terms of features: for instance, for bank applications, functional requirements could be like login, logout, send wire transfers, see payments history, etc. Non-functional requirements deal with usability, security, reliability, performance, maintainability, etc. Domain requirements consider the characteristics of a certain domain of projects: in fact, the operational domain and environment impose requirements on the system; so, the domain requirements can be new functional or non-functional requirements, or constraints on the existing ones.

- **Software design**
  It involves the process of defining the whole architecture (e.g., components, interfaces, etc.) of a system. It deals with the interaction between a system and its environment at a high level of abstraction (interface design), with the main components of a system and their properties, interfaces and interactions between them (architectural design) and with internal elements of the system components, their relationships and algorithms (detailed design).

- **Software construction**
  It is the main step of software development and groups coding/programming, testing (unit and integration) and debugging together. However, during this phase, testing is usually performed by the software developer while the software is under developing, to decide if the just written code is ready for the next step.

- **Software testing**
  It is an empirical phase, in which software is subjected to investigation to inform stakeholders about the product quality. It is performed by staff or developers other than the one(s) who developed and wrote the code.

- **Software maintenance**
  It is the last and usually longest step since it refers to the after-shipping support to the costumer(s). It takes care of software maintenance to correct possible faults and to update it to enhance features and performances. Usually, this step requires more than 40% of the project cost.

Software engineering generally starts with the first step, when a stakeholder submits his requirements to a service provider organization. The software development team isolates user, system and functional requirements. The requirement is collected by conducting interviews, studying the existing system or referring to a database. After that, the team checks if the software can be made to satisfy all

the requirements of the stakeholder. Then, the developer(s) establishes a schedule of his plan. System analysis also involves a study of software product limitations. As well as requirement and analysis, a software design is made. The software design implementation starts with code programming in a proper programming language. Software testing is done while coding by the developers and further tests are performed by testing experts.
All the methods and steps just seen above describe the Software development and Software engineering; however, Software engineering is not a set of static and canonical methods only, but the creative aspect is very important, from requirements drawing up to code writing.
Software Engineering is a very wide computer science sector that is exploited in different fields, and a lot of actors already present or just entered into the market bring their know-how.
One of these actors is the *Zirak s.r.l.* company, which devised this thesis project: it was born in 2000 from the grouping of different companies working on the IT field, and in particular on some different business models, which include on-site consulting, turnkey solutions and R&D (that may lead to final products); it has several autonomously managed departments, such as Automotive, IoT and Software, and it has been able to reach two important goals, innovation and costumer satisfaction.

Before talking about reasons that led to devise this thesis project, an explanation about event-based systems is needed; an event-based system is a programming paradigm for application design, in which the program flow is determined by an external event, the main element of the solution: the event producer, that detects the event, notifies it without knowing how it will be consumed and, then, the event consumer, that receives the event notification, elaborates it in a asynchronous way; this is opposed to a request-driven model, in which the consumer makes repetitive requests until the provider notifies the event.

In 2021, Zirak designed and developed a framework to design event-based systems: they are systems replying to events generated from the outside (e.g., the environment, other systems connected). They are present in a lot of industrial fields and applications: for instance, car sensors continuously generate events that are handled from the control units to perform several computations (e.g., the cruise control system); in avionic systems, autopilots exploit event-based system paradigm to control aircrafts, reading their current position, and then controlling a flight control system to guide the vehicle; in smartphone operating systems, the event-driven paradigm is used to handle all the events coming from the user or from the OS itself (e.g., touchscreen, voice assistant, app notifications, etc.). In particular, the framework developed by Zirak born thanks to the experience of the company in the Automotive sector, but it could be exploited for various industrial fields.

Since the framework could be used writing code only, Zirak thought to develop a User Interface to let the user defining its own system more easily, without dealing straight with the framework APIs and helping the user to save time in system deployment thanks to the automatic generation of code that implements what the user defined visually. All these reasons led Zirak to develop a software able to overcome all the inconvenience of the framework usage (e.g., code writing, framework knowledge), and this is what the thesis aims at. In addition to the User Interface developing, the goal of this thesis is to lighten the workload in some Software engineering phases: certainly, from a time point of view, *software design* and *construction* phases benefit from a GUI that let the developer to model a system and generate automatically the code that defines it; even *integration*, that is part of the construction phase, can benefit from the developed GUI, thanks to the ability of the software to create a unique system that integrates together all the defined components; then, *software configuration control* can benefit from the GUI for the same reasons described about the two phases above since, when modifying the system, the developer only has to modify the system through the GUI and generate the code again; finally, *software maintenance* phase, for the same reasons described regarding the software configuration control step, can benefit from the GUI, since it let the developer to modify the system more easily with respect to modify the system using the framework (i.e., C++ code).
Software engineering and its phases can be deepen exploiting the *Guide to the Software Engineering Body of Knowledge* (SWEBOK)[5].

The thesis is composed by four chapters (leaving out this one), starting from an overview and going deeper and deeper: in the next chapter (Chapter 2) an overview of the technical context and state of the art is given, explaining what event-based systems and state machines are, and seeing approaches, paradigms and frameworks used to develop the thesis project (i.e., metaprogramming and Qt framework); then, in Chapter 3, the framework and GUI are introduced, listing all the features they provide to show how powerful they can be, and explaining some details about software designing and implementation; in Chapter 4, possible improvements on both framework and User Interface are listed; finally, in Chapter 5, a recap will summarize the thesis work, giving personal comments about the whole experience too.

Chapter 2

Technical context and state of the art

2.1 State of the art

2.1.1 Event-based systems

According to Red Hat[1], *event-driven system is a programming paradigm for application design, in which the program flow is determined by an external event*, the main element of the solution. This is opposed to a request-driven model (we will see the differences at the end of this subsection).

Nowadays, there are many software designs based on event-driven architecture. They can be created in any programming language because event-driven is not a programming language, but a programming paradigm. An event-driven system is asynchronous since event producers do not know who or what is listening for an event, and the event does not know what the effects of its occurrence are.

An *event*, that is the core of this type of architecture, *is any important occurrence or change in state for system hardware or software.* Its source can be an internal or external input, like a mouse click or a sensor output. An event must not be confused with its notification, which is sent by the system to notify other parts of the system that an event has occurred.

Event-driven systems are based on communication between event producers and consumers. The former set detects the event and notifies it, not knowing how it will be consumed. Once an event has been detected, it is communicated to the consumers through the so-called *event channels*, where the event is processed in an asynchronous way, and then the consumer may elaborate it. The platform that

processes the event executes the right response to it, sending the event to the correct consumer(s). It is a very important step since it is where the occurrence of an event is visible.

So, the event-driven architecture helps to create flexible systems that can adapt to changes and make decisions in real time.

Finally, let's see the main difference between a *RE*presentational *S*tate *T*ransfer system (REST) and an Event-based system: the first one is based on *repetitive requests* until the provider notifies the expected response, following a polling stategy, which may be resource-consuming (Figure 2.1); instead, the latter is based on the event, so the consumer *subscribes to the event notification* and waits for it (Figure 2.2).

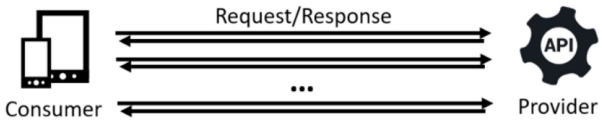

Figure 2.1: Request-driven architecture as in [6].

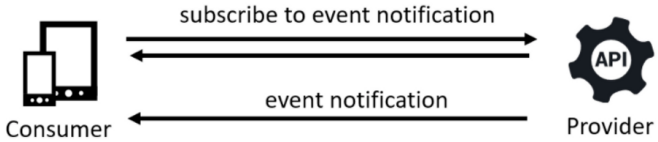

Figure 2.2: Event-based architecture as in [6].

2.1.2 Finite State Machine

According to Yakindu documentation[7], a *state machine is a behavioural model*. It is composed of a *finite* number of states and that is the reason why it is also called finite-state machine (FSM). The machine performs transitions between states and produces outputs based on the current state and a given input. There are two main types of state machines: *Mealy* and *Moore*. Then, there exists a more complex type, defined in the *Harel* model. In order to better understand the outcome of this thesis, we provide an overview about what these types have in common and what are their differences.

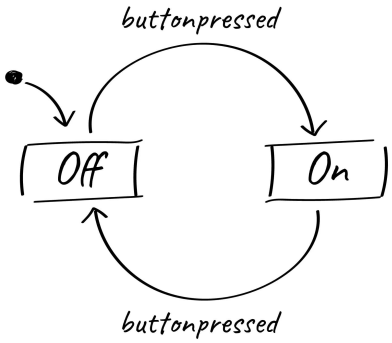

Figure 2.3: State machine handling button pressing as in [7].

The basic objects of a state machine are *states* and *transitions*. *A state is a condition of a system depending on previous inputs and it causes a reaction on following inputs*. The state from which the system computation starts is called the *initial state*. *A transition defines for which input the machine changes state and what will be its destination state*. Depending on the state machine type, states and/or transitions produce outputs. Consider the state machine described in Figure 2.3: it has two states, *Off* and *On*. Off is the initial state, as indicated by the arrow, so it is activated when the state machine starts. The edges between the states show the state transitions. They define which kind of input triggers a state change. Here, the active (current) state is changed from Off to On for the input *buttonpressed*, and back again to Off for the same one. We will extend the simple switch example seen above to highlight the differences between Mealy, Moore and Harel state machines.

- **Moore Machine**
  As said before, there are two main types of FSM. One of these is the Moore machine, named by its inventor Edward Moore in 1956, when the concept was introduced. In this kind of machines, *states produce outputs*, and the output is determined by the current state only. The switch example seen above (Figure 2.3) has been changed into a light switch with different brightness levels (Figure 2.4). The light switch has two buttons: *ON* and *OFF*. Pressing the ON button, the light is turned on and it also toggles through the different brightness levels. Every button pressed raises a corresponding event (*ON_pressed* or *OFF_pressed*) upon which the machine reacts with a state change and an output, that is the brightness level in this case. Since in Moore machines only states produce outputs, one dedicated state per brightness level is needed.

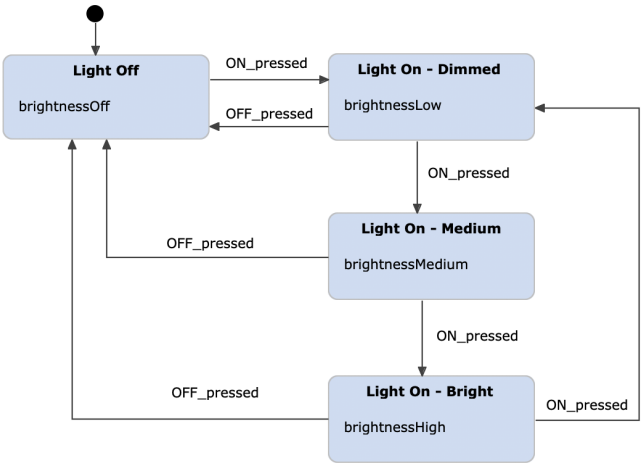

Figure 2.4: Switching light example modelled with Moore machine as in [7].

- **Mealy Machine**
  The Mealy machine concept was introduced by George H. Mealy in 1955. With respect to Moore machines, Mealy ones produce *outputs on transitions only* and not in states. This difference often generates machines with fewer states because more logic can be assigned to transitions.

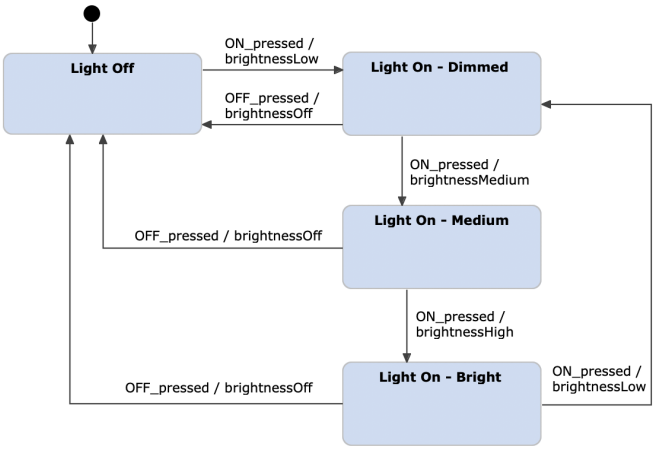

Figure 2.5: Switching light example modelled with Mealy machine as in [7].

Be aware that both state diagrams (Figures 2.4 and 2.5) describe the same system. In fact, *a states diagram modelled with a Moore machine can be always translated into a Mealy machine and vice versa*, without losing any expressiveness.

- **Harel statechart**
  We have just said that Mealy machines can reduce the number of states, but for systems having a lot of states they may become unmanageable.
  David Harel said:

  > *"A complex system cannot be beneficially described in this naive fashion, because of the unmanageable, exponentially growing multitude of states, all of which have to be arranged in a 'flat' non-stratified fashion, resulting in an unstructured, unrealistic, and chaotic state diagram."*
  >
  > *"A state approach must be modular, hierarchical and well-structured".*

  So, he introduced additional concepts like state composition and orthogonality, and he invented the term *"statechart"*, defining it as the mixture between *state diagrams, depth, orthogonality and broadcast communication.*
  Basically, an Harel statechart is a Mealy or Moore machine extended by further concepts that allow to model *complex systems in a more efficient way.*
  Using *composite states* and *sub-diagrams* (they are basically state machines grouped into states), one can build deeper state diagrams, while keeping it clear and well-structured. Orthogonality is expressed using the so-called regions: different sub-state machines that can be executed side by side.
  Moreover, *events* are very important to achieve broadcast communication and let the system to describe complex behaviour. Thanks to *guards*, we can ensure that a given event triggers a transition only if a given condition is satisfied. *Inter-level transitions* (they connect states of different levels and regions), *history states* (used to remember the last active state inside a composite state), *entry, exit and state actions* (routines executed each time the system goes to, stay in or exit from a state) are further Harel statechart elements that make this type of machine so important in describing complex systems' behaviour.

  We said that, describing the same system, Harel statecharts use a fewer number of states with respect to Mealy machines: this is possible thanks to variables, which can be used in input and output expressions. Regarding the same light switch example see before, variables can be used to store the brightness level instead of a different state for each level (Figure 2.6). In this way, we can simplify the statechart by merging all *Light On* states into one and executing the output actions on a self-transition (a transition that goes into the source state). Here we just increment the brightness value each time the transition is activated. So, we can increase the number of brightness levels without adding new states.

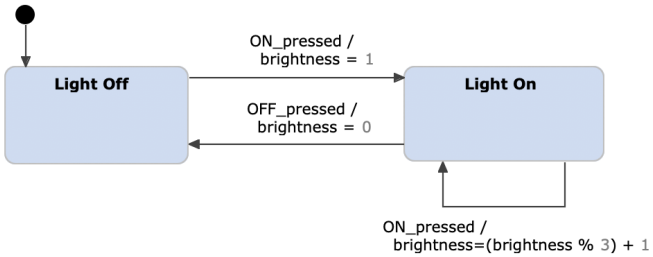

Figure 2.6: Switching light example modelled with Harel statechart as in [7].

To show the use of composite states we can extend the light switch example, adding a motion detection mode (Figure 2.7). When the MOT button is pressed, the motion sensor is activated. Once the sensor detects any motion (event *motion_detected*), the light is turned on at the highest brightness level (*brightness = 3*). This behaviour can be modelled with a composite state that groups the two states *Motion Detected* and *No Motion Detected* together.

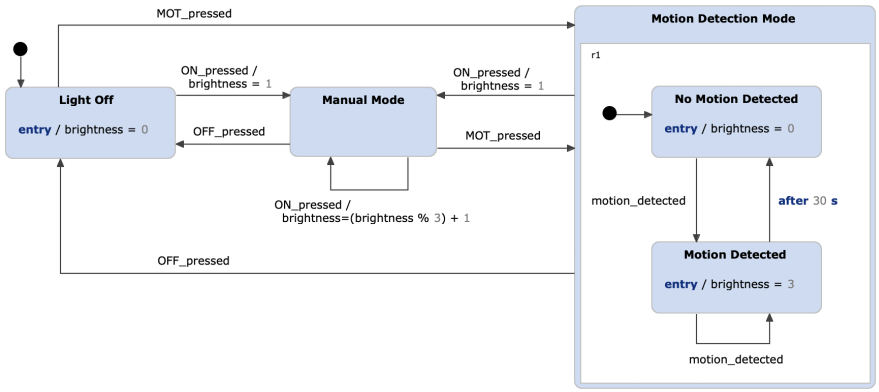

Figure 2.7: Switching light example modelled with Harel statechart and composite states as in [7].

As you can note, Harel statecharts mix the characteristics of Mealy and Moore machines, since *outputs can be produced by states as well as transitions* as reported in Figure 2.7.
We can even go one step further and insert the logic of the Motion Detection Mode into a sub diagram (Figure 2.8). In this way, the system becomes more comprehensive and it shows the operational modes as well as the switching among them in an easier way.

Figure 2.9 shows the differences between the previously described types:

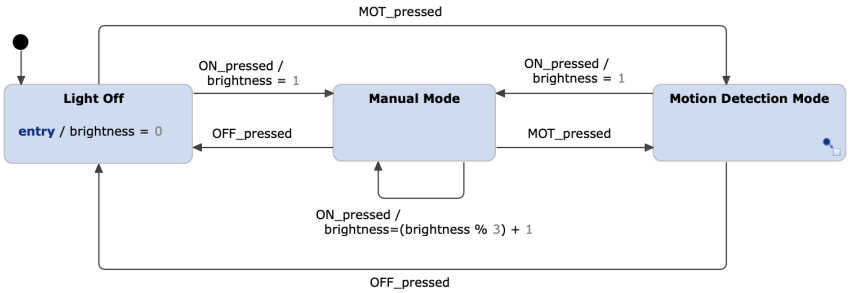

Figure 2.8: Switching light example modelled with Harel statechart and sub diagrams as in [7].

	Mealy	Moore	Harel
States and transitions	✓	✓	✓
Transitions produce output	✓		✓
States produce output		✓	✓
Depth (hierarchies, composite states)			✓
Orthogonality (parallel substatemachines)			✓
Broadcast communication (events)			✓
History, actions, delays, timeouts, conditions			✓

Figure 2.9: Differences between types of state machine described above as in [7].

2.1.3 Toolkits to handle event-based systems and FSMs

Now, we will see three different toolkits, which handle finite state machines and/or event-based systems, analyzing pros and cons, providing a comparison platform with the final product that this thesis aims at developing.

- **Yakindu Statechart tools**[8]
  It is a toolkit for *developing, simulating, and generating executable code for finite state machines*. It is based on the open-source development platform Eclipse.
  It is available in *Standard* and *Professional Edition*, which are free if you use them for non-commercial or educational purposes, otherwise the user shall purchase the license (the latter having a higher price, providing additional features).

  The **Standard Edition** provides the following features:

  – GUI for the graphical editing and representation of state machines

- A simulator to simulate the behaviour of state machines
- Code generators to transform graphical representation of state machines into programming language source code like C, C++, Java and Python
- Custom generator projects exploiting Xtend or Java in order to create any code
- Validator to check for syntactical or semantical issues of the modelled state machine (e.g., a state without transitions)
- Framework to test state machines with unit tests

The **Professional Edition** adds features such as:

- Integration with the C programming language to access variables, types, etc.
- Simulation and debugging tools with breakpoints and snapshots

The main **pros** of Yakindu tool are:

- Handling of state machines with a lot of features like composite states, history nodes, orthogonality, two different execution approaches (event-driven vs cycle-based)
- Presence of simulator, validator and tester
- Code generation in different programming languages

The main **cons** are:

- Absence of elements that handle events in a different way with respect to state machines (e.g., simple event handlers that receive events and do routines according to the event, without dealing with states)
- Licences to buy if you want to develop a system for commercial use (even for the Standard Edition)
- Unique event queue for dispatching events, so no possibility to dispatching events in parallel
- Integration with C and C++ programming languages with Professional Edition only

- **OPNET**[9]
  It is a software that *enables the simulation behaviour and performance of any kind of network*. It provides built-in protocols and device models, and it allows to create and simulate different network topologies.

  Main **pros** of Opnet:

- Network planning and design.
- Validating hardware architecture.
- Protocol modelling.
- Traffic modelling of telecommunication networks.
- Evaluating performance aspects of complex software systems.
- Create/import topology/configuration.
- Simulator to execute designed network.

Main **cons** of Opnet:

- The user needs to purchase the license to use it.
- The set of protocols/devices is fixed.
- No facilities that enable code generation for the modelled network.

- **OMNET++**[10]
  It is a C++ *simulation library and framework, mainly for simulating networks* intended in a wider sense.
  It provides a component architecture for models, and modules are programmed in C++, then grouped into bigger components and models thanks to a high-level language named NED (*NE*twork *D*escription).

  The main **pros** of Omnet++ are:

  - Open-source and free software.
  - Simulation kernel library using C++.
  - The NED topology description language.
  - Simulation IDE based on Eclipse.
  - Interactive simulation runtime GUI.
  - Command-line for simulation execution.
  - Utilities (Makefile creation tool, etc.).
  - Model frameworks developed as independent projects such as sensor networks, internet protocols, etc.
  - SystemC integration.
  - Compatibility with all platforms with a C++ compiler.

  The main **cons** of Omnet++ are:

  - No facilities that enable code generation for the modelled network.
  - Few models available.
  - Open-source and free for non-commercial purposes only.

2.2 Technical context

2.2.1 Templates and Metaprogramming

Usually, it is necessary to define a priori the variable type used as function arguments, as well as class members, due to the static check performed over types by C++ language. This can require duplicating entire code sections only to adapt their use to different types, going against the fundamental principle of code reuse. Let's look to the following piece of code, where *swap_integers* and *swap_strings* functions are basically identical, except for the variable types they have as arguments.

```
1 void swap_integers(int& a, int& b)
2 {
3     int tmp = a;
4     a = b;
5     b = tmp;
6 }
```

```
1 void swap_strings(string& a, string& b)
2 {
3     string tmp = a;
4     a = b;
5     b = tmp;
6 }
```

One of the most interesting aspects of C++ programming language is the presence of a *metaprogramming*[11] system. It enables the execution of operations at compile-time, like elaborating source code, using the keyword *template*: it is used to indicate a generic code section (for instance, a function, a class or a variable, starting from C++14), which implementation can be adapted to several data types.
So, thanks to metaprogramming, it is possible to collapse the definition of the two functions above into only one, so that it is valid for any data type, as shown in the following code snippet:

```
1 template <typename T>
2 void swap_generic(T& a, T& b)
3 {
4     T tmp = a;
5     a = b;
6     b = tmp;
7 }
```

The keyword *template*, followed by a list of parameters *typename NameType* between angle brackets, is used for the *swap_generic* function in order to make it a generic one. During compile step, each time a call to a generic function is found, the compiler applies rules to define the correct value to assign to the *NameType* parameter (in this case, *T*) and it adds to the original source code an "overload" of that function in which the generic parameter is substituted by specific type. This process is called *template instantiation* and it comes before the compile step. This type of use of templates in C++ is an example of *generic programming* and, differently from other object-oriented programming languages, the template definition in C++ constitutes a programming language in itself.
The metaprogramming explained above is the *type metaprogramming* one, but there are other two important metaprogramming types[12]: the value and hybrid metaprogramming; the former is used to compute values at compile-time, instead of during execution, while the latter exploits templates to reduce execution time too.

In GEM framework, template is the main element used to define basically all the objects that it contains, like state machines generic definition, that can be exploited to define state machines of different types.

2.2.2 Introduction to Qt

Qt[3] is a framework for creating and modelling graphical user interfaces and cross-platform applications that can be executed on several different hardware/software platforms such as Linux, Windows, macOS, Android or embedded systems, with little changes or even without changing the code.
Most GUIs created with Qt appear with a native interface, in which case Qt is classified as a widget toolkit. Non-GUI programs can be developed too (terminal programs), such as consoles for servers.
Qt supports different compilers, including the GCC C++ compiler, the Visual Studio suite and PHP. It also provides Qt Quick, that includes a declarative scripting language named QML that exploits JavaScript to provide the logic. With Qt Quick, it is possible to rapidly develop mobile devices applications, while their logic can still be written using native code to achieve the best possible performance, according to the underlying target platform.

Qt relies on the following key concepts:

- **Abstraction of the GUI**
  Qt exploits the native style APIs of the platform in which it is mounted on, assuming that it provides such an API set. Moreover, on some platforms Qt is the native API.

- **Signal/slot paradigm**
  It is a framework construct introduced in Qt for communication between objects. The concept is that UI widgets can send signals containing event information which can be received by other controls using special functions called slots; so, a signal notifies an event that triggers one or more actions (slots).

- **Metaobject compiler**
  The metaobject compiler (MOC) is a tool that is executed on the sources of a Qt program. It interprets certain macros from the C++ code and uses them to generate additional C++ code with meta information about the classes used in the program. This meta information is exploited by Qt to provide programming features not available natively in C++, such as signal/slot paradigm and asynchronous function calls.

- **Language bindings**
  Thanks to this feature, Qt can be used in various programming languages other than C++, such as Python, Javascript, C# and Rust.

Qt has its own set of tools to make easier the cross-platform development; *Qt Creator* is one of these tools, that is the main one used in the creation of GEM software. Qt Creator is a cross-platform IDE for C++ and QML that exploits *Qt Designer*, another Qt tool useful to design the GUI layout, although Qt Designer can still be started as a standalone tool.
In addition to Qt Creator and Qt Design, Qt provides *qmake*, a cross-platform build script generation tool that makes automatic the generation of Makefiles for development projects across different platforms.

So, thanks to all the features of Qt, in particular for its signal/slot paradigm and the Qt Design tool that helped a lot in the designing of the GUI, thanks to the various number of widgets and visual customization it provides, it has been chosen for designing and developing of GEM user interface. Furthermore, since the GEM software is only for academic-use at the moment, the open source (free) Qt license has been chosen. However, if it is converted into a standalone application by Zirak, the paid license will be taken into account.

2.3 Thesis goal

Up to now, we have seen an overview about event-based systems (Section 2.1.1), state machines (Section 2.1.2) and toolkits that handle them (Section 2.1.3); in particular, we have seen how these toolkits deal with event-based systems and FSMs in different ways and with different features. However, we also listed their drawbacks, such as *no ability to deal with generic event handlers, whose state machines are a particular implementation, no deep integration with C and C++ programming languages* and, last but not least, *no chance to generate code automatically.* These are only some of the limitations that the toolkits shown have.

The goal of this thesis is to try to overcome the limitations above grouped in a unique software. Obviously, the toolkits reported in Section 2.1.3 are developed by teams of experienced developers and they have been set up after a lot of hours of designing and development, unlike the limited amount of time of GEM product development; however, this thesis outcome tries to start a software project that will be updated in all its aspects, possibly having a platform with more features and a more pleasant visual aspect than now. In fact, in Chapter 3, we will see that the User Interface does not provides all the framework functionalities, since it is only a starting point for a wider software project (see Chapter 4 for more details about future works and possible improvements about the GEM software).

The idea is to have a software, capable of handling an event-based system that can contain:

- **Event handlers** with their interface in order to deal with incoming events
- **State machines** containing:
  - States
  - Composite states
  - Transitions
- **Signallers** to notify events to event handlers and state machines
- **Dispatching queues** to send events in parallel
- **Binding features** to connect signallers to event handlers and FSMs, selecting the favourite dispatching queue

In order to deal with all these features and many others, the *GEM* (*G*eneric *E*vent-based *M*odeling) framework has been exploited.
GEM is a framework developed by *Zirak s.r.l.*[2] and, as its name implies, *it handles*

generic event-based systems: so, the user can use the large number of available APIs to model an entire event-based system with all the features listed above. But, since the framework is entirely written using high-level programming languages like C and C++, the developer has to know the entire set of APIs and all their functionalities before using the framework. That is the reason why *the GEM framework has been extended with a easy-to-use GUI*, that let the user to model its system with all the features, *without knowing the framework APIs* and especially *decreasing the amount of time needed to arrange the software code*, thanks to the *code generation* functionality (the whole GEM software is contained in the Zirak GitLab repository). In Chapter 3 we are going to see in details both GEM framework and User Interface, in order to better understand why the entire platform (framework+GUI) has been developed and all the benefits it can provide.

Chapter 3

Software Development

In this chapter, we discuss the development of the GEM software: first of all, we will see an overview about the framework, how it is made and the features it provides (Section 3.1); then, we will go deeper into the development of the User Interface, seeing all the design choices made, what it offers to the user and how it works (Section 3.2).

3.1 *G*eneric *E*vent-based *M*odeling framework

The purpose of the framework is to model generic event-based systems through several APIs, and thanks to three main activities:

- **Signalling**
  It is the activity in charge of notifying events to the system and, in particular, to event handlers. It is basically the interface of the system with the external environment. It can be filtered in various ways in order to prevent some events from entering the system.

- **Handling**
  It is the activity in charge of taking events from the outside (i.e., from signallers) and performing actions accordingly. In other words, it is the control activity of the system in which events are consumed. This phase is carried out by generic event handlers and structured state machines.

- **Dispatching**
  It is the activity in charge of transporting events from signallers to handlers. It involves all the communication channels used to perform event dispatching, such as dispatching units: they are queues in which events flow from event producer (signaller) to event consumer (handler). It also lets the system to be

faster in events dispatching and communication thanks to parallel dispatching, if used: in fact, if more dispatching units are used, event(s) can be notified to different handlers simultaneously, assuming that binding process has been performed accordingly.

3.1.1 Features and objects

As discussed above, the purpose of the framework is to model generic event-based systems and, thanks to models, event handlers and state machines, it may realize the design pattern *MVC*, that stands for *M*odel-*V*iew-*C*ontroller. It is an architectural pattern widespread in the software systems development, able to separate the logical data representation from the business logics.

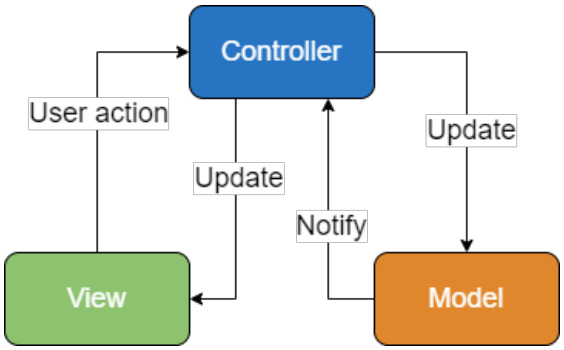

Figure 3.1: Diagram of design pattern MVC.

In Figure 3.1, a diagram of MVC design pattern is reported. The model provides methods for accessing data, useful for the application, the view uses the data contained in the model to provide an interaction between users and system, the controller receives commands from the user (generally through the view) and uses them for modifying the other two components state (view and model).

Now, let's speak about features and objects the GEM framework provides:

- **Signallers**
  Signallers are one of the most important elements into the framework, since they are used to signal whatever event is needed: in fact, they can be declared using any C++ type, be it POD-type or user-defined type, in order to signal that type of events. They can be bound to dispatching queues in order to send their events to one or more receivers. They can be *"on-demand"* signallers, so they are used according to the user needs, *timed* signallers, so they use time-based strategies, *function* signallers, so they signal the outcome of a function call, and *function caller/function callee*, so that they can be used to realize the Qt signal/slot paradigm.

- **Filters**
  They can be applied to signallers and enable source event filtering, and they may differentiate parallel dispatching, so they can be applied to different dispatching queues with different types of filtering. The GEM framework provides several types of filters:
  - *Sampling* filter that generates events every n samples
  - *Threshold* filter that generates events according to a threshold t
  - *Range* filter that generates events according to a range r
  - *Timed* filter that generates events every time interval i
  - *Deviation* filter that generates events according to deviation d from last valid event
  - *Custom* filter that generates events according to a user-defined filter
- **Models**
  They are used to contain static lists of signallers. They realize an index for the whole component and enable compile-time indexing of signallers. Models make easier the communication between sub-components.
- **State Machines**
  State machines are one of the main objects in GEM framework. We already discussed about them in Section 2.1.2 and we have seen the features they can provide to an event-based system, since they handle events performing actions strictly related to them. In GEM framework, state machines can be *primary* or *forwarding* machines: the former can be used as input in the binding process, and the user can set values in the state machine arity; the latter, instead, cannot be used as input in the binding process, and they shall receive arity tuple from the outside. The *arity* of a state machine is a set of parameters that are used to tune the state machine behavior while it is operational; it can be declared when defining state machine or it can be not used at all, defining the so-called empty arity state machine. Moreover, GEM framework supports mixed nesting, it means that a system can be composed of several types of state machines (i.e., primary and forwarding ones). Finally, the framework provides the possibility of dealing with two different policies for state machine structure, *bound-free (flat-reference)* and *object-plugin (object-reference)*: in the former, every action point may refer any callable entity and, if an object, it shall be already included in the call and cannot be changed; in the latter, every action point shall refer an object of type T and the reference object is not included in the call and can be changed during the state machine life-cycle: in fact, an object is a parameter in the state machine arity through which the user may change the reference object.

In the following, the two type of states are described. They are very similar to the ones described when discussing about Harel model (Section 2.1.2).

  - **States**
    States are the main element, together with transitions, composing state machines. They may contain three kinds of actions: *entry actions*, executed when the system enters the state; *step (or state) actions* when the system is into the state; *exit actions* when the system exits from the state. They may contain transitions to other states or to themselves (*loop transitions*). Transitions may contain actions to, called *transitions actions*.
  - **Composite states**
    Composite states are states that may contain one or more state machines, each activated by some guard conditions on the triggering event. If more than one state machine is present, only one of them will start at entry. Once the system exits from a composite state, reset actions are performed, and the current state machine is reset.

- **Event handlers**
  Event handlers are, together with state machines, the unique event listener object provided by the GEM framework. Unlike state machines, they have only a *handle event function* that is executed whenever an event is signalled to it from some signallers.

- **Dispatching**
  Signallers described above need communication channels in order to notify events to related listeners: the channel prototype used is the *dispatching queue*. Signallers and listeners are bound together: the binding may be 1-to-1 or 1-to-N, in order to signal the same event to multiple listeners in a serial fashion, using the same dispatching queue. If we need *parallel* dispatching, we can use as much queues as we want. But, with serial dispatching, one thread is needed only, unlike the parallel one that uses as many threads as the number of queues used. During binding phase (when listener is connected to signaller through a dispatching queue) one can choose the queue he wants, otherwise the default dispatching queue will be chosen automatically.

- **Binding**
  Binding is the way through which signallers are connected to listeners (event handlers and state machines). As mentioned before, one can choose the dispatching queue to exploit, otherwise the default queue is chosen if no queue is specified. There is also the possibility to unbind two elements, so disconnecting them and preventing communication with each other.

- **Timers**
  Finally, timers allow to trigger actions on a timing-basis: they can be *one-shot* (they work only once) or *periodic* (they trigger over a predefined amount of time). They also allow to configure/change the clock type at OS-level (i.e., the period between time ticks).

3.2 Visual approach to software modeling

We listed all the objects provided by the GEM framework (see Section 3.1.1) but, in order to use them, we have to know the APIs they provide. The goal of this thesis, as already said in Section 2.3, is to provide to the user a visual environment easing the use of the framework, while modeling the software system. In the following, we will discuss how the GEM software is made and what it provides to the user, how it integrates the GEM framework and the main issues found during its development. It is very important to remind that, as already said in Section 2.3, the User Interface does not offer all the functionalities provided by the GEM framework, which will be possibly added in future release.

3.2.1 Architectural composition

We may now delve into the GEM software implementation, looking at its architecture model. Figure 3.2 shows the software architecture, highlighting its partition in three modules: the GEM GUI, the Business Logics and the GEM framework. The main function of the Business Logics block is to support the GUI layer while the user is interacting with visual objects; moreover, such a block uses the GEM framework when the system code must be generated.
It must be pointed that, in Figure 3.2, components in Actions Logics block are the visual representation of GEM framework components (e.g., signaller component in Actions Logics is the visual representation of signaller in GEM framework).

In the following, an overview of the components that build the GEM software is given:

- **Drawing Logics**
  This logic layer aims at providing the user the ability to represent the software system by means of figurative language; it also provides interaction logics that ease the user in navigating through the software components. Now we see more in detail how it has been designed and how it works.

  As you can see from Figure 3.3, the drawing area, called *DragWidget* in GEM GUI, occupies most of the application area, since it is used to define the system, inserting signallers, event handlers, state machines and bindings

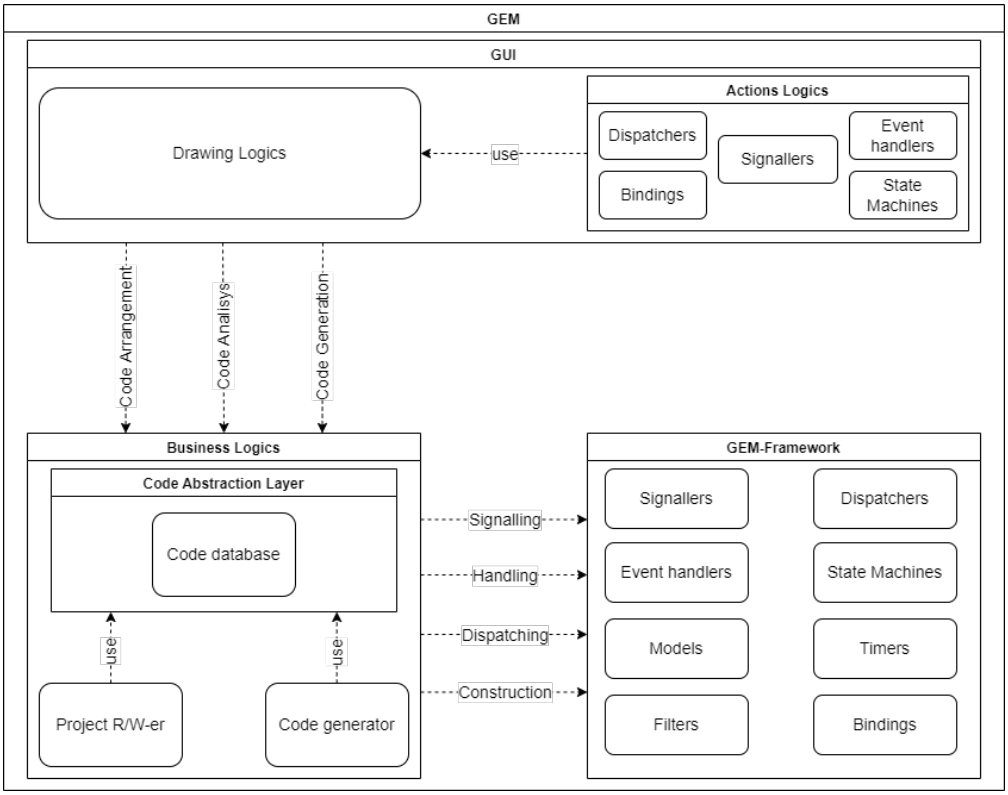

Figure 3.2: GEM User Interface architectural composition.

between objects. The same Qt object (i.e., DragWidget) is used in the so-called *State Machine Frame*, that is the window in which states and transitions of a state machine are handled. As its name implies, this widget provides drag and drop functionalities of objects to provide the user the ability to move objects wherever it wants.

Drag Widget area offers several functionalities that can be grouped into two parts, one for handling drag/drop actions and objects menus and one for handling connections between objects (i.e., bindings and transitions), even if they are strictly connected between them, since whenever an object with bindings or transitions connected to it is moved, they are moved together with the object.
The set of features that handles drag/drop actions and objects menus leverages the underlying system of events for drag, drop and mouse movement/actions supported by Qt. Whenever an object contained in the Drag Widget is left clicked, the *QMouseEvent* event is generated and the object name (e.g., the name of a state machine), type (e.g., state machine, signaller, etc.) and icon

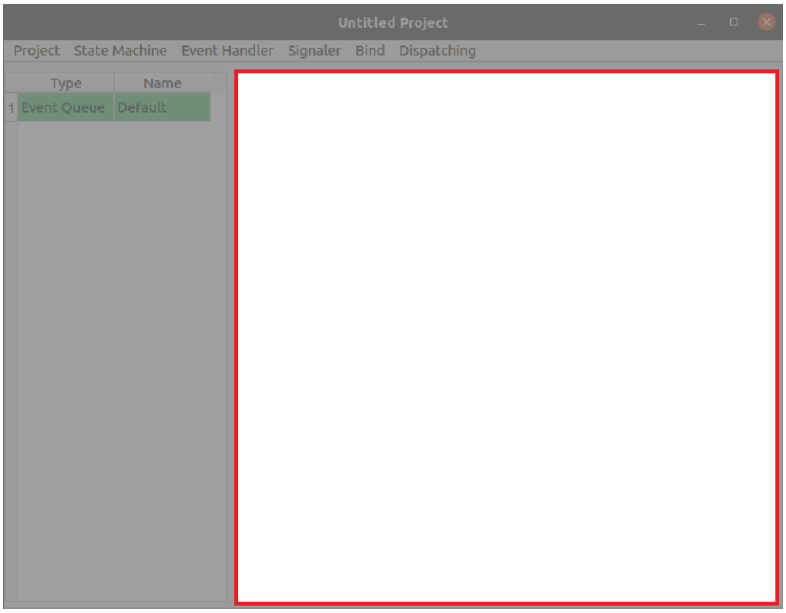

Figure 3.3: Drawing area of GEM User Interface.

are retrieved; then, thanks to *QDragEnterEvent* and *QDragMoveEvent* events, the GEM software understands if the moved object is contained and moved into the Drag Widget, or if it is allowed to be moved: for instance, if the clicked object is a bind or a transition, the event is rejected since they are not allowed to be moved, as they are only when objects connected to them are moved. Instead, if an object is right clicked, the proper menu is shown, providing the set of actions allowed on the desired object (e.g., editing object name, remove object, etc.).
The other set of features, which handles creation and movement of bindings and transitions, unlike drag, drop and mouse events, was not already supported by Qt, but it has been designed and implemented by scratch (see Section 3.2.4). Whenever the creation of a connection between two objects (i.e., bindings and transitions) is needed, the corresponding method is called (*createBind* or *createTransition*); instead, whenever an object with at least one bind or transition is moved, the *moveBind* or *moveTransition* methods are called.

- **Actions Logics**
  This logic layer aims at providing the high-level actions, needed to perform the main operations over the drawn software system. It deals with all the objects that are manageable within GEM software: signallers, state machines, event handlers, dispatching queues and bindings. Depending on the action and on the object, it works in a different way: for instance, if the user wants to

create an object, it can use the GEM UI toolbar and then, the proper object menu (Figure 3.4). Moreover, in order to deal with transitions inside state machines, the user can exploit the actions accessible from the proper state's window (Figure 3.5).

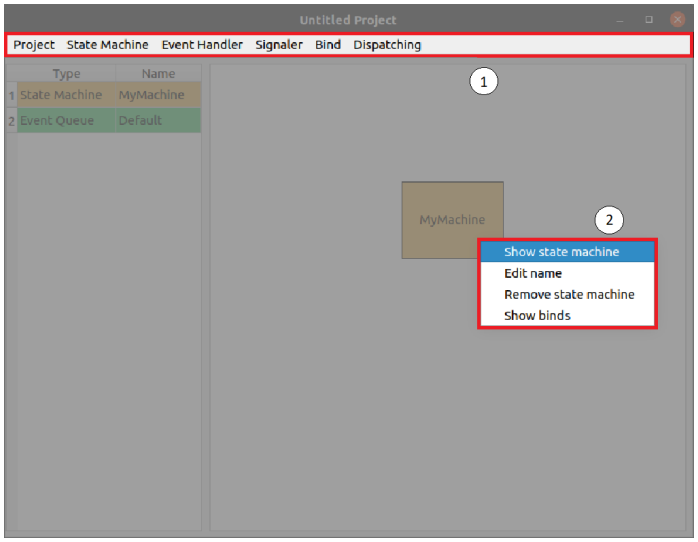

Figure 3.4: Actions accessible from toolbar (1) and menu (2).

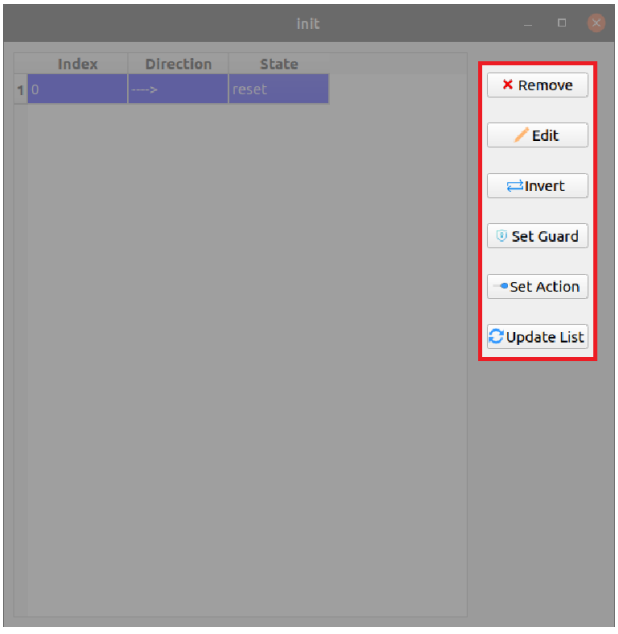

Figure 3.5: Actions accessible from state's transitions window.

- **Code Abstraction Layer**
  The Code Abstraction Layer is one of the most important elements in GEM software since it represents the "bridge" between GUI and framework. In fact, it aims at providing high-level APIs able to represent the framework actual APIs, decoupling the representation and the operation layers. This eases the maintenance and extensibility of the code, if the framework changes/expands (except for those changes that would involve further code layers in the system: in the UI layout, for example).
  In the GEM software code, the Code Abstraction Layer is a component that provides the APIs needed to handle the objects the GUI supports: for instance, it provides APIs to create, edit or delete objects.

- **Code Database**
  This layer aims at storing the high-level representation of the code currently involved in the system drawn by the user, tracing all the relationships between software components and symbols; in other words, what is drawn in the UI has a corresponding data structure in this layer. The Code Database is included in the Code Abstraction Layer, since they are strictly related: for instance, if the user wants to create a new object (e.g., a signaller), the proper API, included in the Code Abstraction Layer, accesses the Code Database to insert the new object, with all the parameters set by the user (e.g., signaller name, signaller type, etc.).

- **Project R/W-er**
  The layer aims at realizing the logics needed to read and write the project information from/in the project format file (i.e., *.gem*), enabling the user to store all the information for an ongoing project and resuming it later, avoiding the need to modelling the entire project at once.

- **Code Generator**
  The Code Generator layer aims at realizing the logics needed to generate the actual code required to implement the system drawn in the UI by the user. It exploits the Code Abstraction Layer to retrieve the objects defined by the user and, then, it generates code using the APIs provided by the framework in a directory chosen by the user, following a predefined arrangement for code and directory tree.

3.2.2 Features and Use Cases

In this Section, we discuss all the features the GEM software provides. In Figure 3.6, the use case diagram is shown, and we analyze it more in detail in the following. It must be pointed out that a use case diagram is a graphical representation of

the interactions between user and system; its name derives from the use cases composing it (e.g., manage signallers, configure dispatching, etc.). It should be shown the user(s) interacting with the use cases but, since it is always the same (i.e., the developer using the GEM software), for sake of visibility it has not been reported. The "extend" relation is used when a use case (i.e., project handling) enhances another use case (i.e., draw system); instead, the "use" relation (also known as dependency relation) is used when a use case (i.e., generate code) relies on another use case (i.e., draw system) for its implementation.

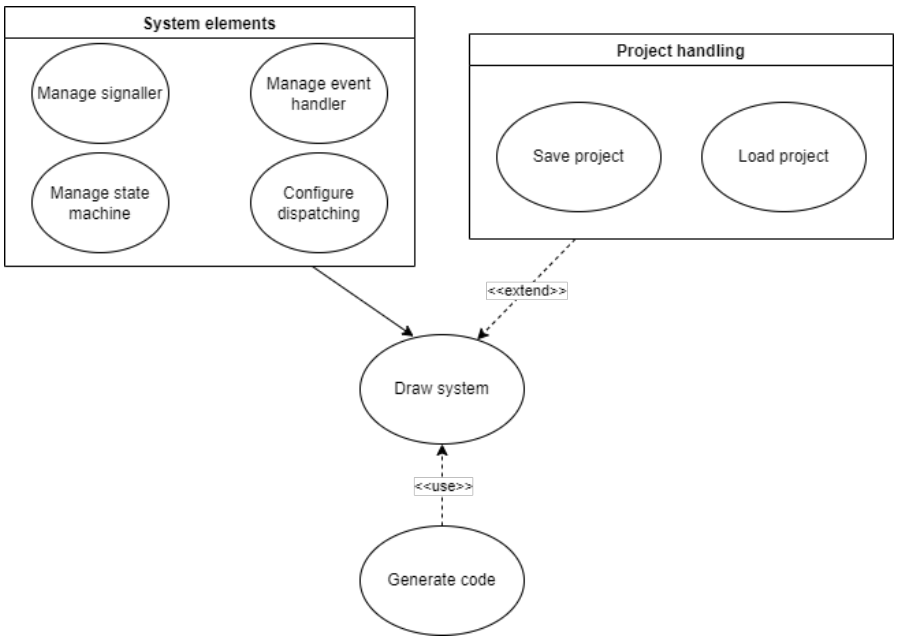

Figure 3.6: Use case diagram of GEM software.

- **Manage signallers**
  We discussed about signallers in Section 3.1.1: they are used to signal events to the system: they always are one of the two objects involved in bindings (the other one can be an event handler or a state machine). In GEM software, signallers have only two parameters: name of the signaller and its type; the type is an important parameter, since it indicates what type of events the signaller generates. The type should be one among the set of built-in types supported by C++ language, such as int, float, double, etc.
  The GEM software allows the user to create a signaller through toolbar (Figure 3.4 (1)), indicating its name and type, to edit a signaller through its menu (Figure 3.4 (2)), modifying the name and/or the type of the signaller, and to remove a signaller by means of its menu (Figure 3.4 (2)).

- **Manage event handlers**
  We discussed about event handlers in Section 3.1.1: as their name implies, they are used to handle events by means of function called *handleEvent*. In GEM software, event handlers have three parameters: event handler name, type and function; in particular, the type, unlike signallers, indicates the event handler class name, since every event handler is identified by its name and its class name (the class that contains the handleEvent function). The function parameter contains the handleEvent function implementation.
  The GEM software allows the user to create an event handler through toolbar (Figure 3.4 (1)), indicating its name and class name, edit an event handler through its menu (Figure 3.4 (2)), modifying its parameters (name, class name and function implementation), and to remove an event handler through its menu (Figure 3.4 (2)).

- **Manage state machines**
  We discussed about state machines in Section 3.1.1: they are one of the most complex objects provided by GEM software. In fact, in GEM software they need several parameters: name, type (primary or forwarding, even if only primary state machines are admitted now by the GUI), init (or entry) state (the state from which the execution starts), states and transitions.
  The GEM software allows the user to create a state machine through toolbar (Figure 3.4 (1)), indicating its name, to edit a state machine name (Figure 3.4 (2)), and to remove a state machine through its own menu (Figure 3.4 (2)).
  As reported by Figure 3.4 (2), the user has the ability to inspect a state machine, dealing with states and transitions:

  – **States**
    We discussed about states in Section 3.1.1. In GEM software, states have their name and actions: the latter can be of three types, depending on when they are executed; in fact, states can have entry actions, executed when the system enters a state, step (or state) actions, executed when the system is into the state and an event leads no transition to activate, and exit actions, executed when the system exits from the state.
    The GEM software allows the user to create states inside a state machine through the toolbar (Figure 3.7 (1)), to edit a state (Figure 3.7 (2)), modifying its name and actions, to set it as init state (Figure 3.7 (2)), and to remove a state through its own menu (Figure 3.7 (2)).

  – **Transitions**
    We discussed about transitions in Section 3.1.1, while dealing with states. In GEM software, transitions have an index that uniquely identifies them, source and target states as parameters (they are the same in case of loop transitions), guard that defines the condition for which the transition is

activated, and transition actions that are executed whenever a transition is taken.
The GEM software allows the user to create a transition through the toolbar (Figure 3.7 (1)), defining whether it is a loop or not, source and target states, to edit a transition through the state's transitions window (Figure 3.5), modifying its source and target states (the user can also change the transition from normal to loop one and vice versa), its guard and transition actions and even inverting a transition (changing target and source roles), and to remove a transition through its menu or state's transition window (Figure 3.5).

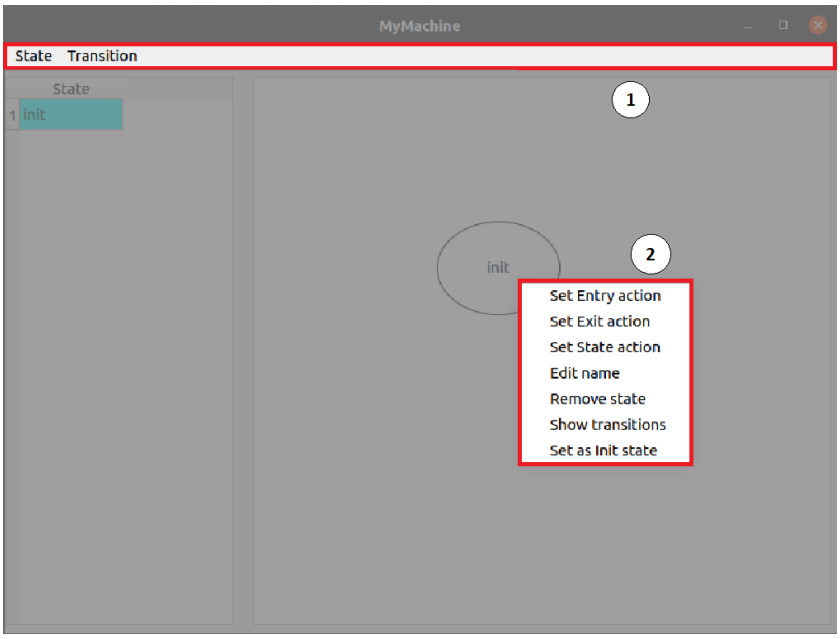

Figure 3.7: Actions accessible from toolbar (1) and menu (2) in state machine window.

- **Configure dispatching**
  We discussed about dispatching queues and bindings in Section 3.1.1: they are important in the events communication, since they let signallers to notify events to consumers (i.e., generic event handlers and state machines). In particular, binding is the connection step between a signaller and a consumer over a dispatching queue, but the connection itself is named *bind*.

  – **Dispatching queues**
    In GEM software, dispatching queues have only their name as unique parameter. When starting a new project, a *Default* queue is already

present, but the user can create (and remove too) others through the toolbar (Figure 3.4 (1)). In particular, when removing a queue, if it is assigned to one or more binds, the user can decide if removing those binds too, or redirecting them to the Default queue.

- **Binds**
  In GEM software, binds have, similarly to transitions, an index to uniquely identify them, the signaller and the handler they belong to, and the dispatching queue exploited for the communication.
  They can be created through the toolbar (Figure 3.4 (1)), edited through their menus (Figure 3.4 (2)), giving the user the ability to modify the signaller, the handler and the dispatching queue, and removed through their related menus (Figure 3.4 (2)).
  In addition, the user can check the binds in which an object is involved, or check in which bind a dispatching queue is involved: the former can be done through the menu of every object that can be involved in a bind (i.e., signallers, generic event handlers and state machines), selecting the *"Show binds"* option (Figure 3.4 (2)); the latter can be done by double clicking on a dispatching queue name in the left-side list (Figure 3.4).

- **Project handling**
  We discussed about Project R/W-er in Section 3.2.1: it is a logical module that is in charge of saving and resuming ongoing projects, without the need of modelling the system all at once. It is split in two parts, one for saving and one for loading the project:

  - **Save project**
    As its name suggests, it is used to save an ongoing project, that will be possibly resumed later. It exploits the Code Abstraction Layer to retrieve all the objects defined by the user and then, thanks to the *QDataStream* Qt object, they are serialized and saved on a file, which position in the file system is chosen by the user. In addition to the object parameters we discussed above, the position in the Drawing area (Figure 3.3) is saved too, so that, when loading the project, they can be positioned in the same place they were when the project was saved. Finally, when the project is saved and the user chooses the name of the project file, the project name is saved with the same name given to the file. The project name is assigned also to the directory containing the generated code, and that is the reason why, during first saving, the software asks the user to save the project before generating the code.
    The user can save a project simply through the toolbar in Figure 3.4 (1), clicking on *Project -> Save project* .

  - **Load project**
    As its name suggests, it is used to load all the information for a project, saved during a previous work session. It works using the opposite process undertaken by its counterpart: thanks to the *QDataStream* Qt object, it reads the project file, loading the objects together with their parameters in the Code Abstraction Layer (i.e., in the Code Database) and updating the GUI using the visual representation of the imported objects, respecting the position they had during the saving project phase.
    The user can load a project simply through the toolbar in Figure 3.4 (1), clicking on *Project -> Load project* .

- **Generate code**
  We discussed about Code generator in Section 3.2.1: it is a module used to generate automatically the code about the modelled system. It exploits the Code Abstraction Layer to retrieve all the objects defined by the user and then, according to the GEM framework APIs, it writes on files the code representing the modelled system.
  The code generation phase starts from signallers, without which the system has not sense due to the definition of an event-based system (signallers are fundamental for the notification of the incoming events): if no signallers are found in the modelled system, the code generation stops, showing the corresponding warning to the user; otherwise, the code generation process continues, scanning all the other objects and generating the code accordingly. Each object type has two dedicated files, an header (.h) and a source (.cpp) file: the former contains the linking to the GEM framework, the declaration of all the objects of that type (e.g., all the signallers defined by the user) with the *extern* keyword, in order to enable their usage in files outside the project (i.e., the whole system) and the *init* function prototype, that is used to initialize all the declared objects; the ".cpp" file contains the linking to the corresponding header file described above, the definition of all the objects of that type (e.g., all the signallers defined by the user) and the *init* function implementation, in which the objects are initialized and checked whether their initialization was successful or not.
  The code generation process just described is the same for all the objects, except for state machines, that have a dedicated header and source file each: in fact, state machines are the objects that contains most of the information in GEM software, and writing the code related to all the state machines defined by the user in one single file would make the maintenance more difficult, due to the implementation required for entry, exit, state, guard and transition actions, but also the implementation of the state machine structure, such as transitions between states.

3.2.3 Usage scenarios

After discussing features and use cases, in this section we see some usage scenarios of the GEM software. For sake of reading, only some scenarios will be shown.

- **Scenario 1: Create signaller**
  Precondition: No precondition needed.
  Post condition: signaller is created.

  1. The user clicks on *Signaller* → *Add Signaller...* in the toolbar (Figure 3.8).

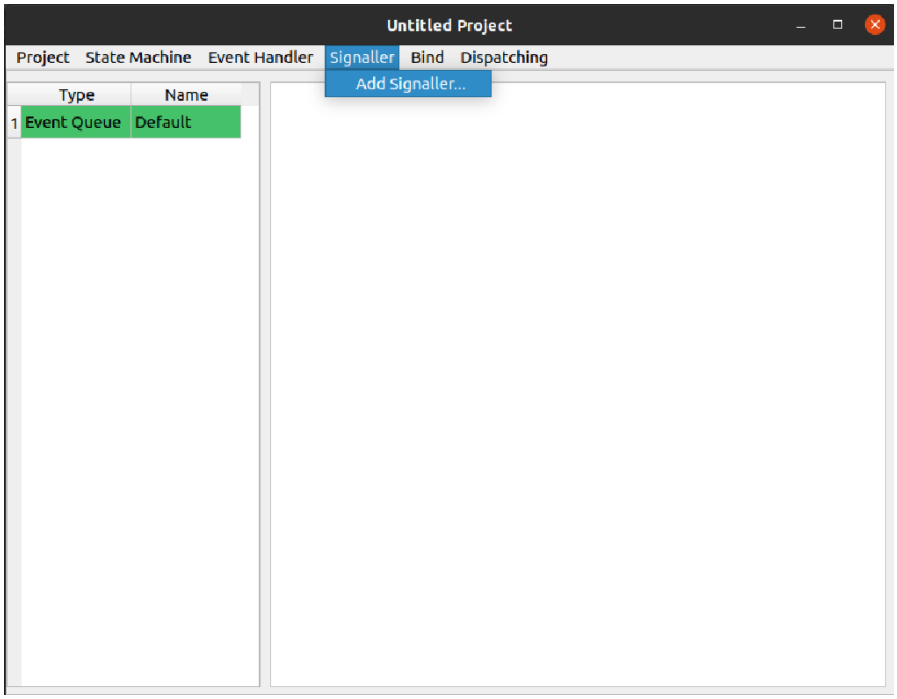

Figure 3.8: Create a signaller: step 1.

  2. The user inserts signaller name and type (Figure 3.9).

  3. The signaller is created and the user can move it wherever he wants (Figure 3.10).

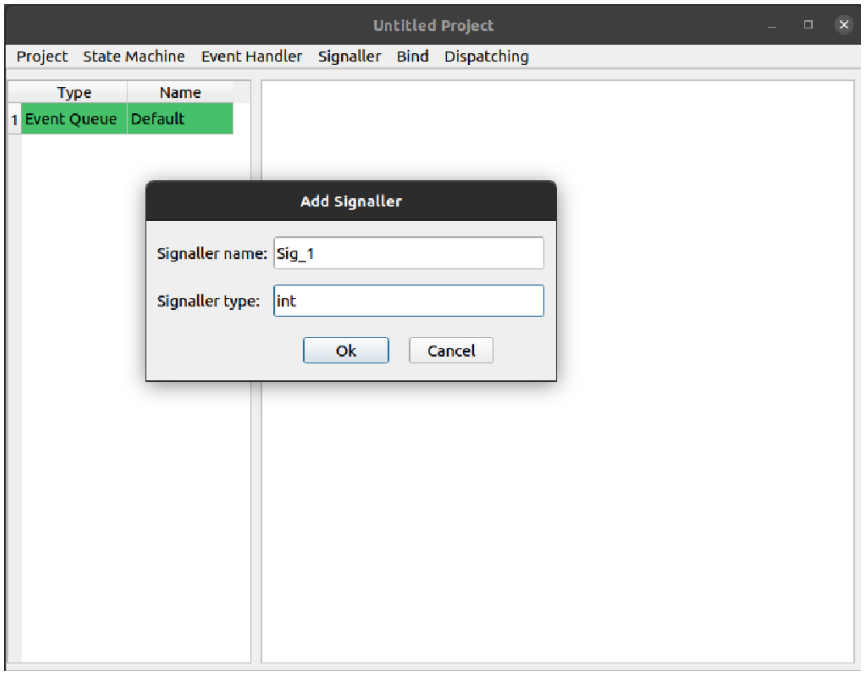

Figure 3.9: Create a signaller: step 2.

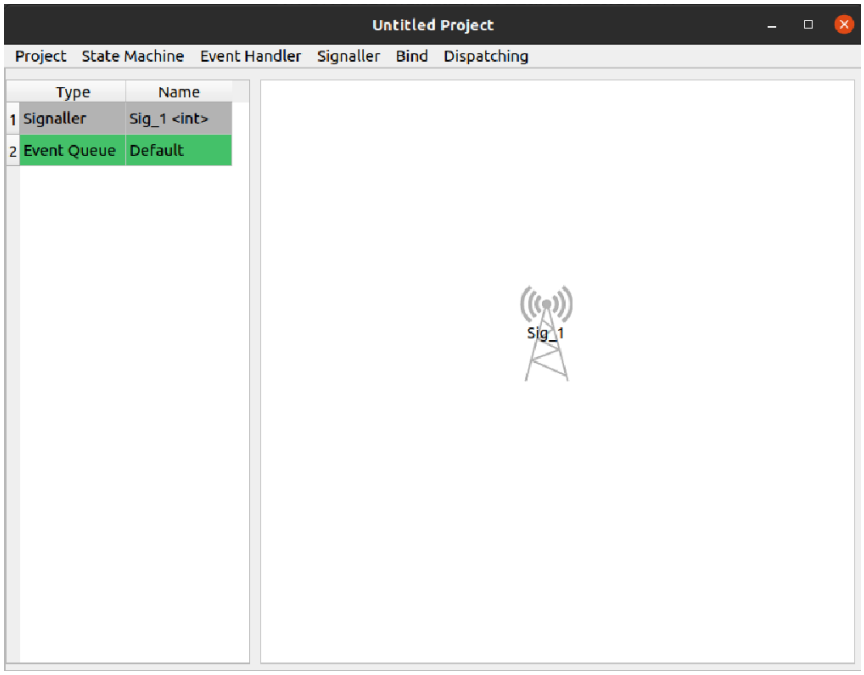

Figure 3.10: Create a signaller: step 3.

- **Scenario 2: Create bind**
  Precondition: signaller(s) and handler(s) (generic event handler and/or state machine) already created.
  Post condition: bind is created.

  1. The user clicks on *Bind* → *Create bind...* in the toolbar (Figure 3.11).

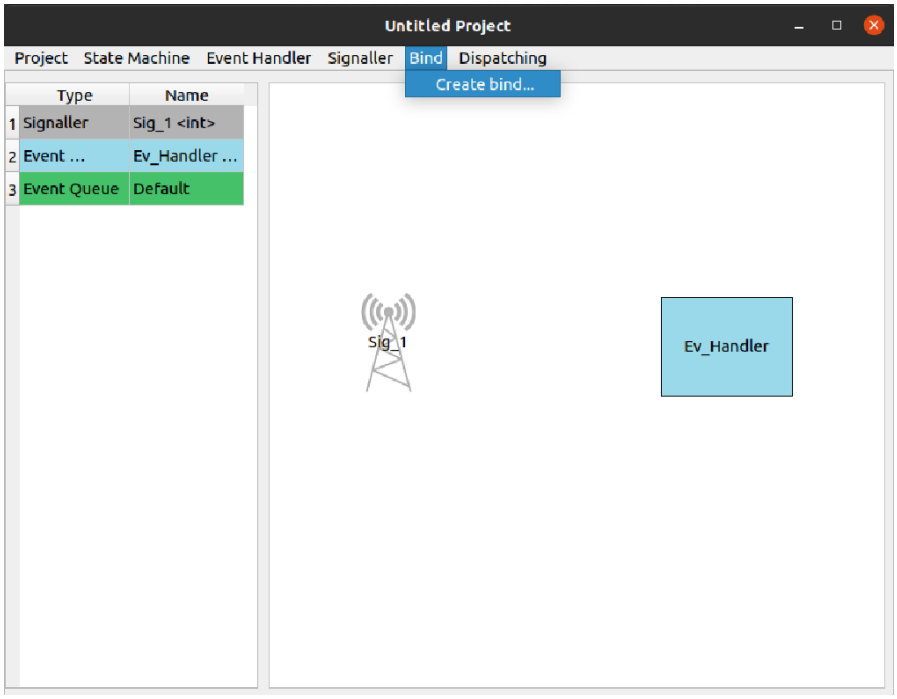

Figure 3.11: Create a bind: step 1.

  2. The user selects the signaller and the handler to bind together and the queue to assign to the bind (Figure 3.12).
  3. The bind is created (Figure 3.13).

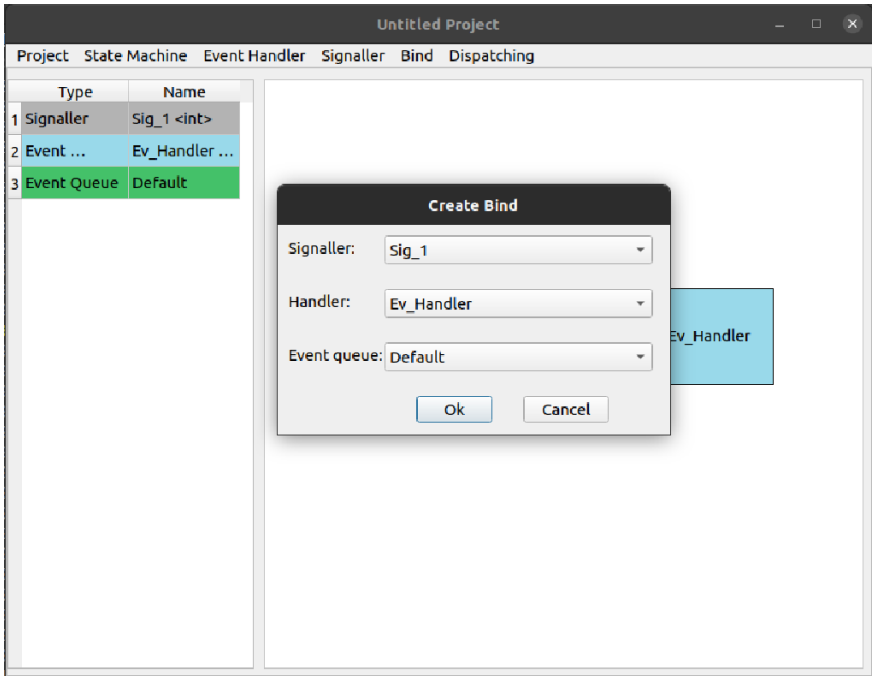

Figure 3.12: Create a bind: step 2.

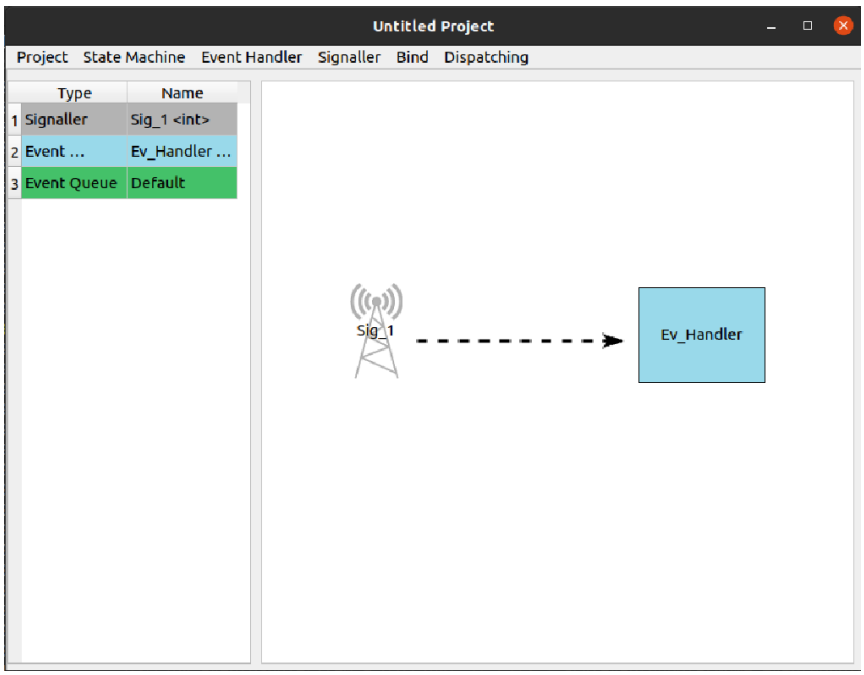

Figure 3.13: Create a bind: step 3.

- **Scenario 3: Show binds (objects)**
  Precondition: binds already created.
  Post condition: binds are shown.

  1. The user right clicks on a signaller, event handler or state machine and then clicks on *Show binds* in the menu (Figure 3.14).

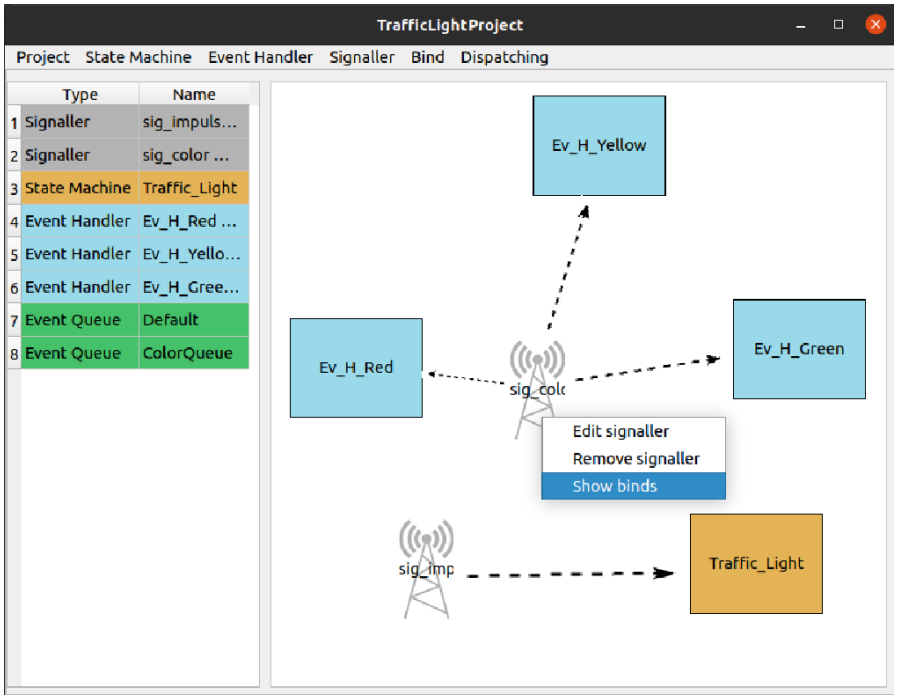

Figure 3.14: Show binds (objects): step 1.

  2. A list of binds involving the chosen object is shown (Figure 3.15).

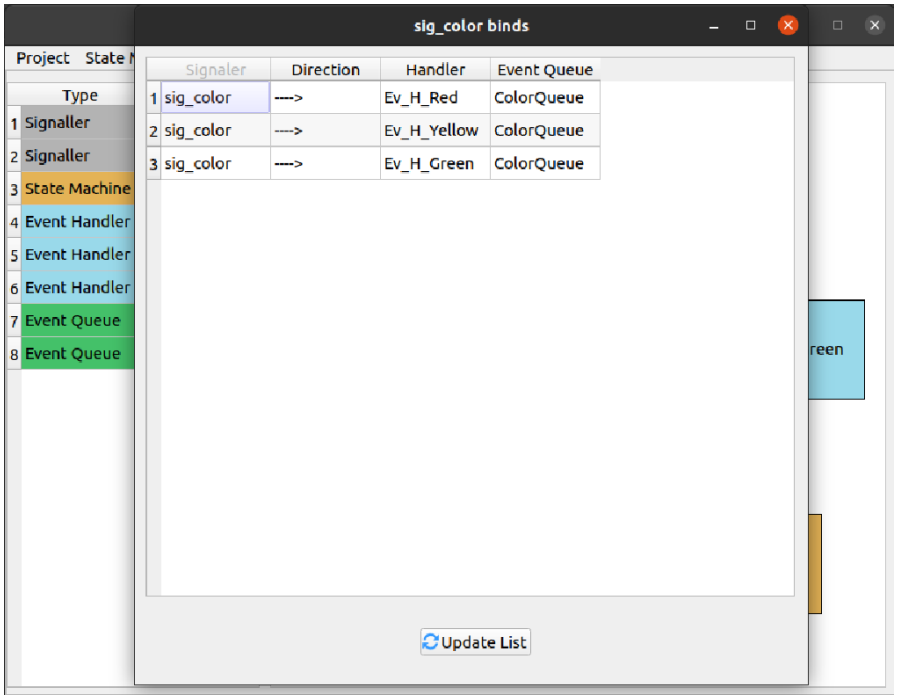

Figure 3.15: Show binds (objects): step 2.

- **Scenario 3 bis: Show binds (queues)**
  Precondition: binds already created.
  Post condition: binds are shown.

  1. The user perform a double-click on a queue in the list on the left side of the main window (Figure 3.16).
  2. A list of binds involving the chosen queue is shown (Figure 3.17).

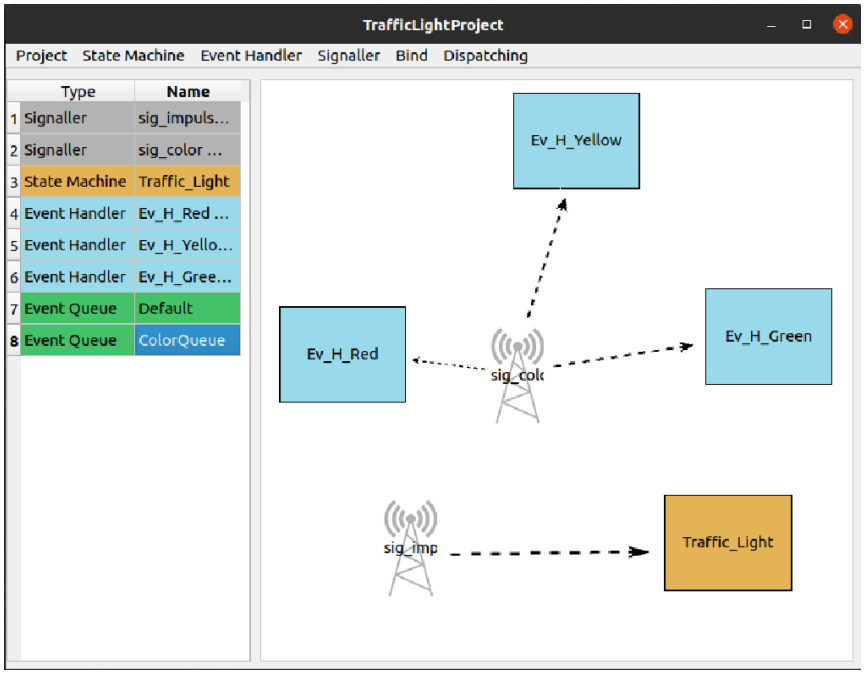

Figure 3.16: Show binds (queues): step 1.

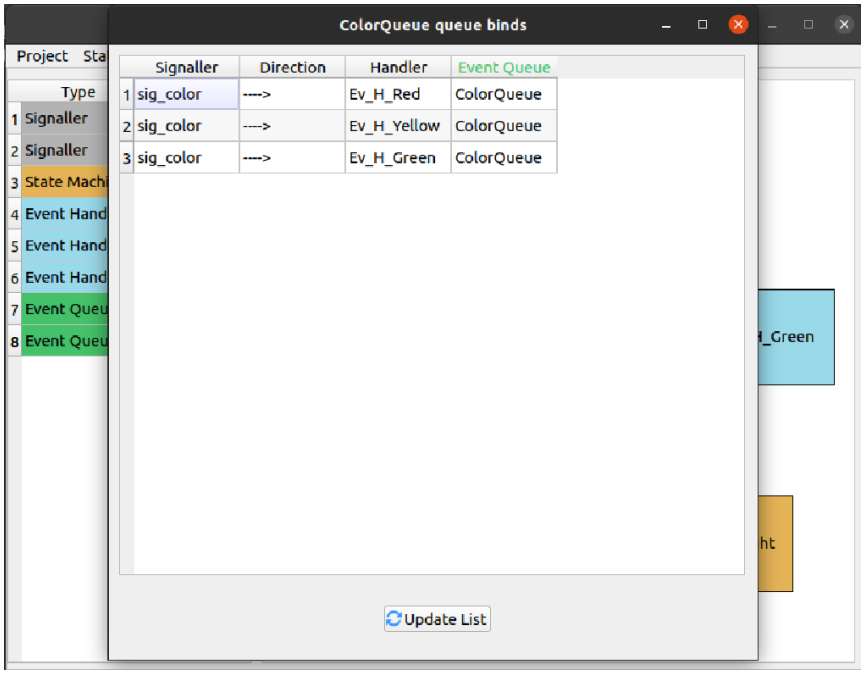

Figure 3.17: Show binds (queues): step 2.

- **Scenario 4: Create transition**
  Precondition: state machine already created, at least one state created.
  Post condition: transition is created.

  1. The user does right click on a state machine and then does click on *Show state machine* in the menu (Figure 3.18).

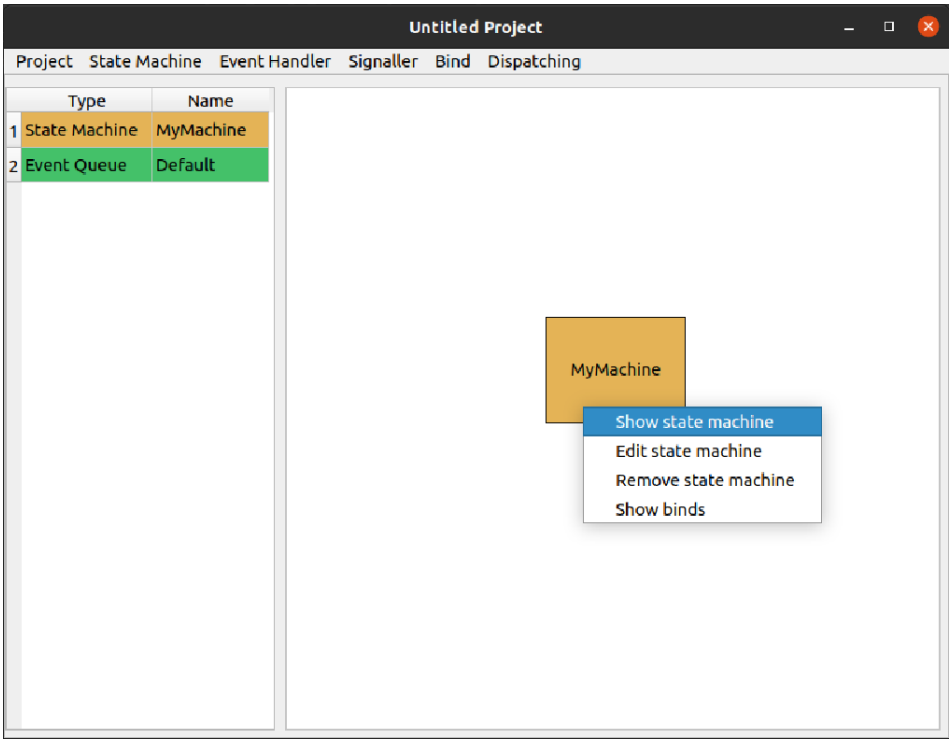

Figure 3.18: Create a transition: step 1.

2. The user does click on *Transition* → *Add Transition...* in the toolbar (Figure 3.19).

3. The user selects the source and target states of the transition to be created, and the transition type in the *Arrow* combo box (if Loop type is chosen, when changing source/target state, the other changes accordingly since, in loop transitions, source and target states must be the same) (Figure 3.20).

4. The transition is created (Figure 3.21).

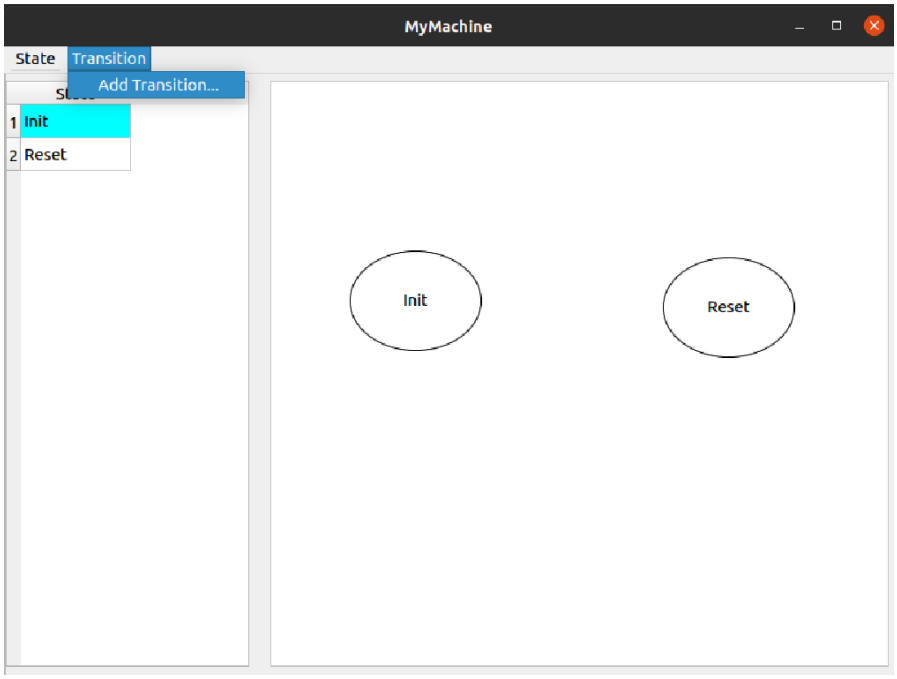

Figure 3.19: Create a transition: step 2.

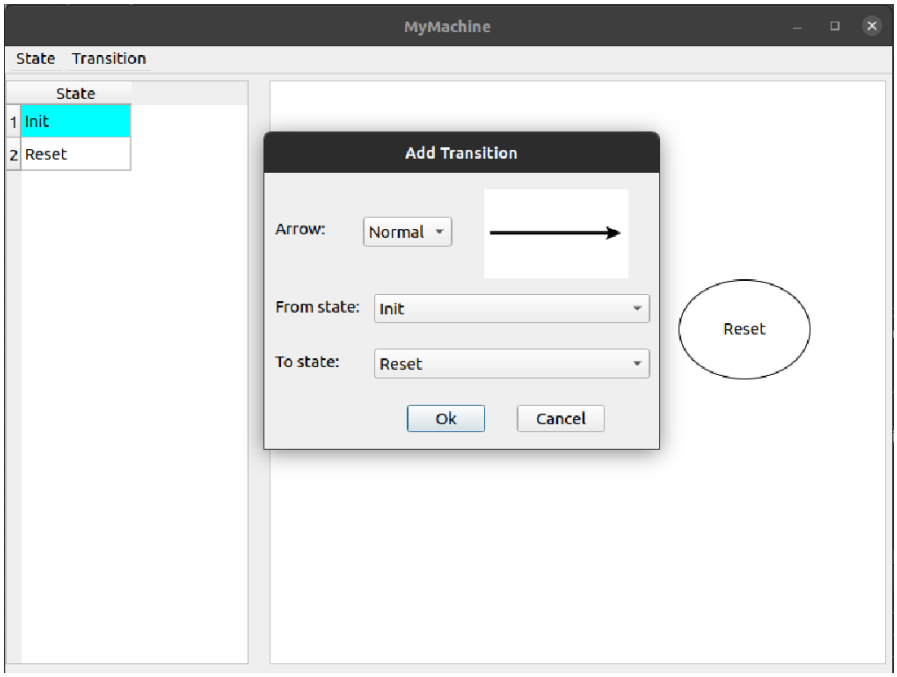

Figure 3.20: Create a transition: step 3.

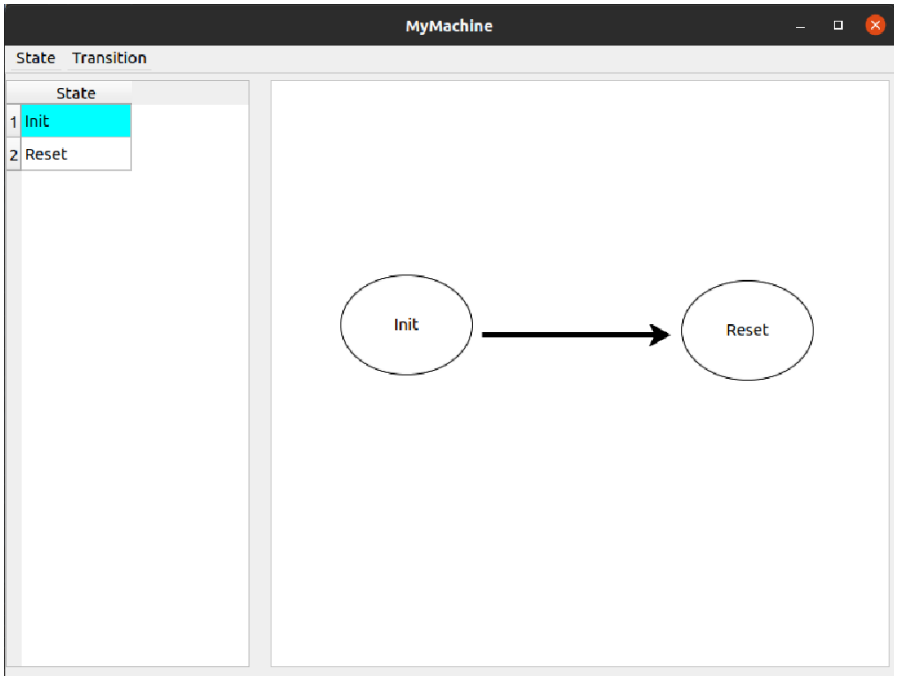

Figure 3.21: Create a transition: step 4.

- **Scenario 5: Set guard actions for a transition**
  Precondition: transition already created.
  Post condition: guard actions are set.

  1. The user does right click on a state machine and then does click on *Show state machine* in the menu (Figure 3.22).

  2. The user does right click on a state from which the involved transition comes out and then does click on *Show transitions* in the menu (Figure 3.23).

  3. The user does click on a transition and then does click on *Set guard* button (not clickable button if, in that transitions list, the chosen one is an inward transition, since guard actions are checked only when the system goes out from a state) (Figure 3.24).

  4. The user defines the guard actions and then does click on *Ok* button (Figure 3.25).

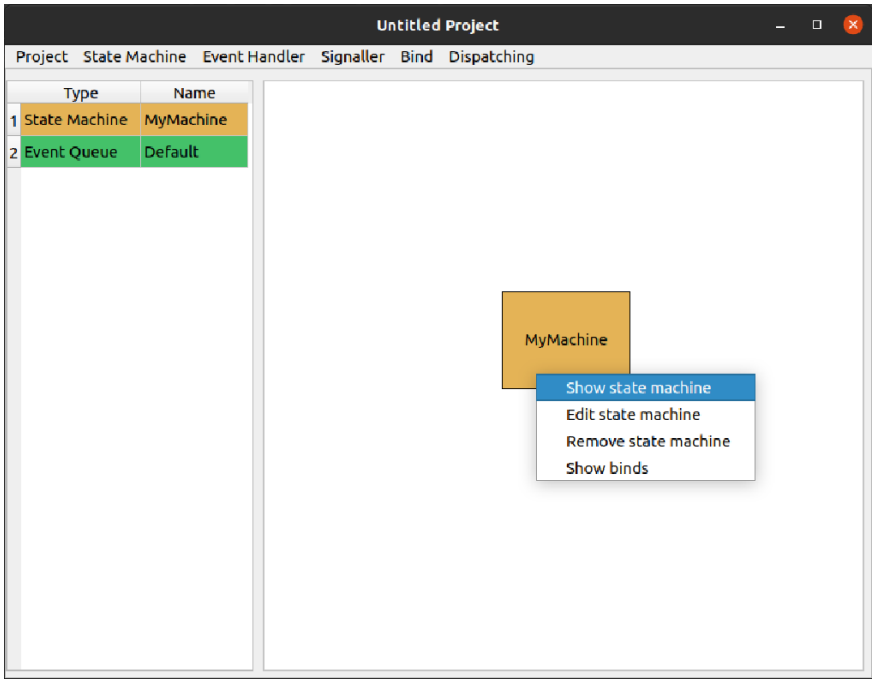

Figure 3.22: Set guard actions: step 1.

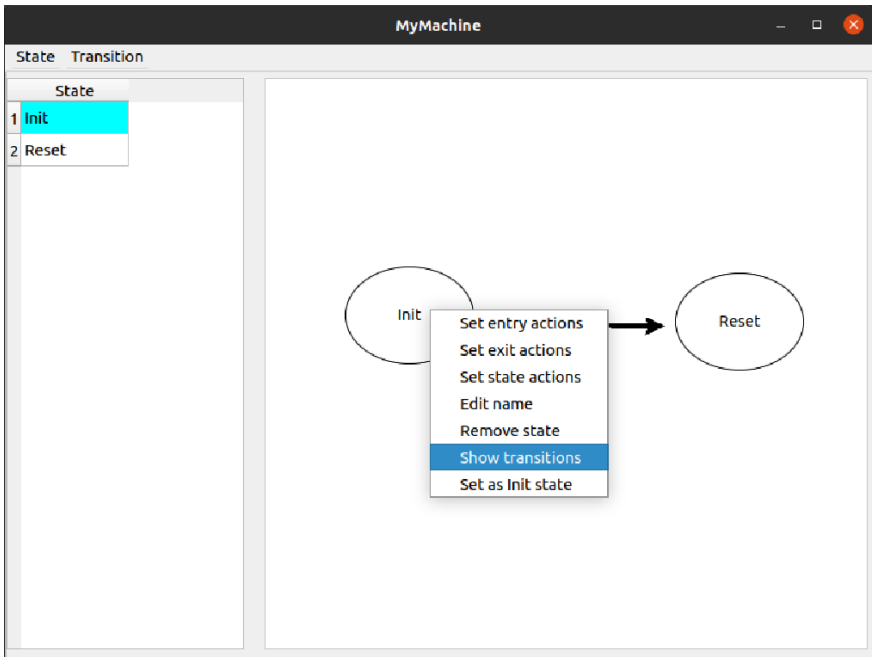

Figure 3.23: Set guard actions: step 2.

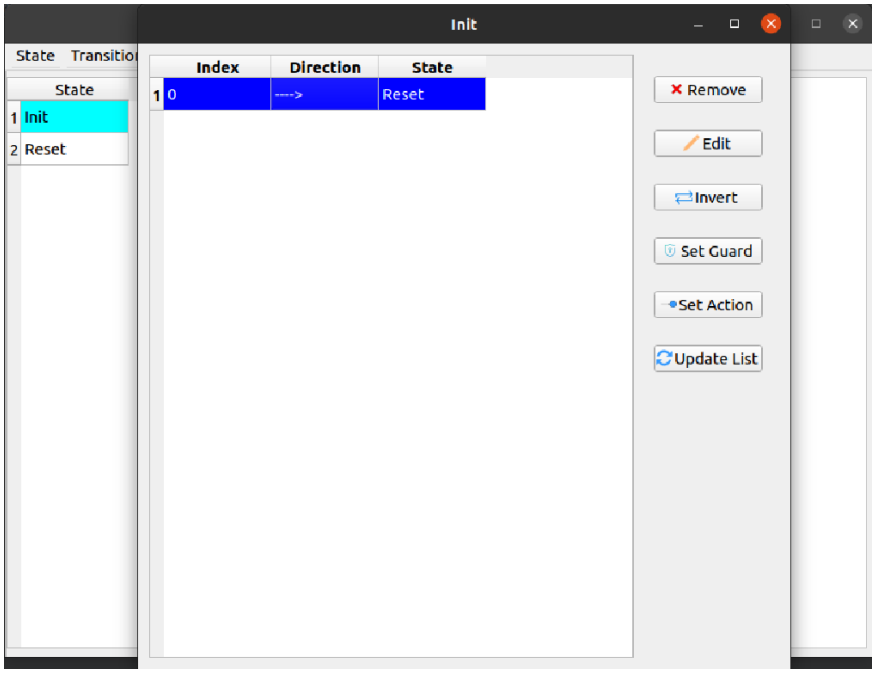

Figure 3.24: Set guard actions: step 3.

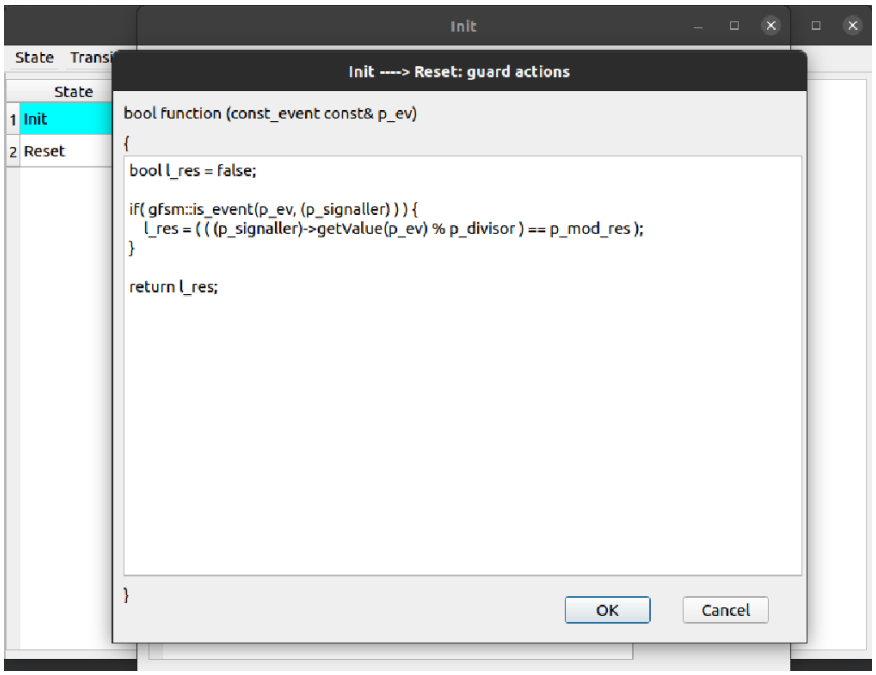

Figure 3.25: Set guard actions: step 4.

- **Scenario 6: Set entry actions for a state**
  Precondition: state already created.
  Post condition: entry actions are set.

  1. The user does right click on a state machine and then does click on *Show state machine* in the menu (Figure 3.26).

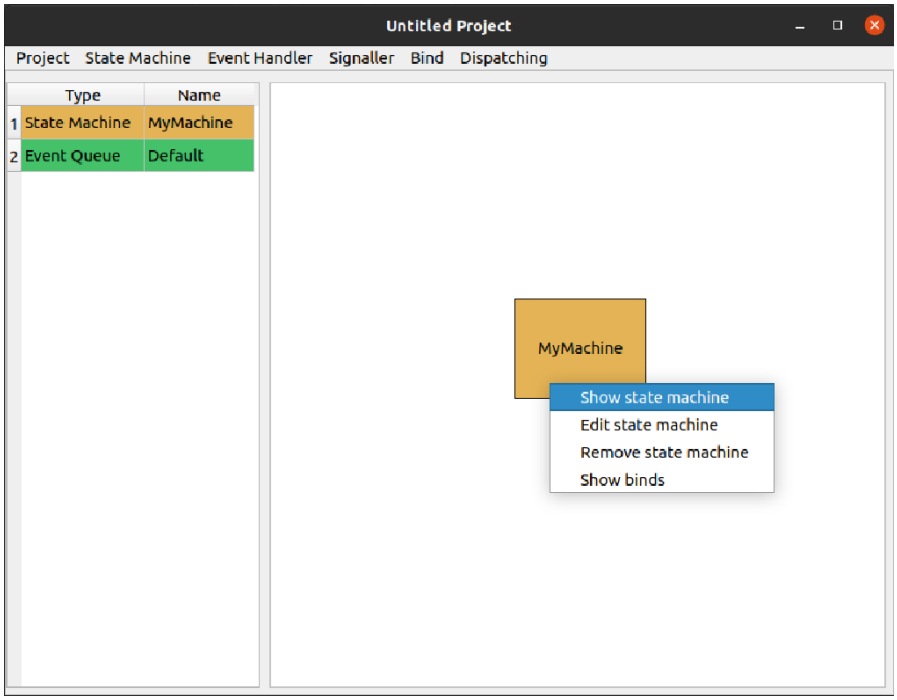

Figure 3.26: Set entry actions: step 1.

  2. The user does right click on a state and then does click on *Set entry actions* in the menu (Figure 3.27).

  3. The user defines the entry actions and then does click on *Ok* button (Figure 3.28).

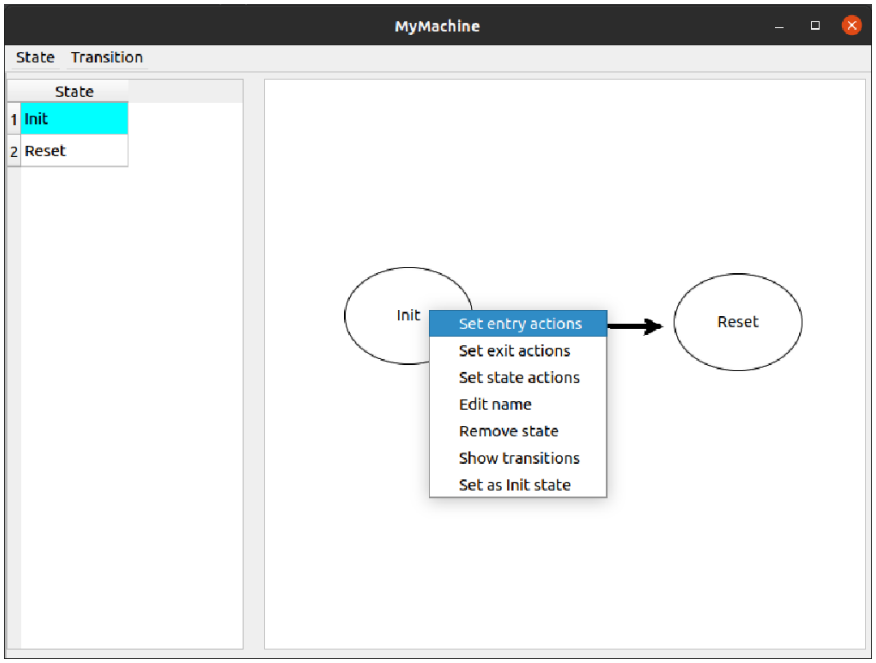

Figure 3.27: Set entry actions: step 2.

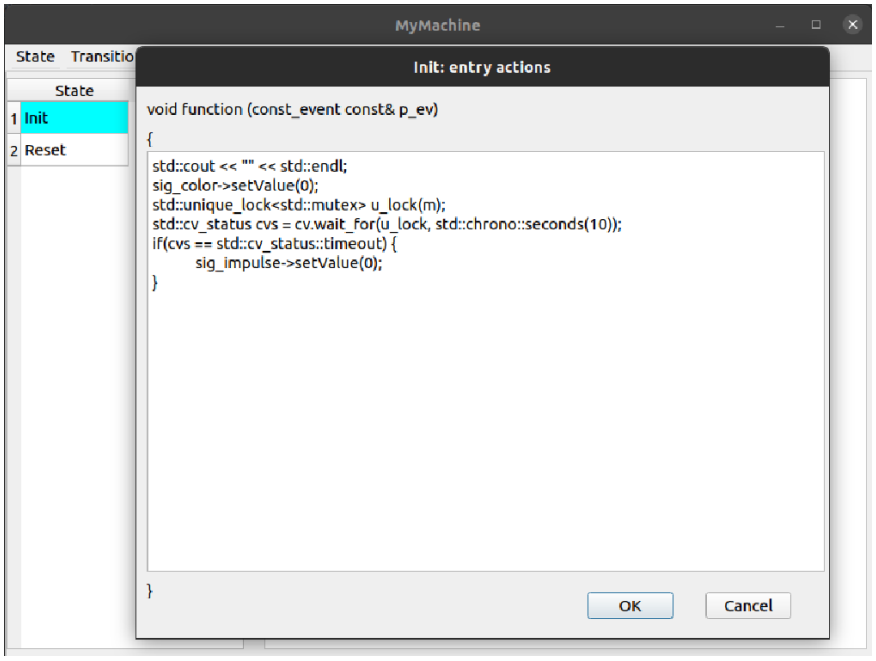

Figure 3.28: Set entry actions: step 3.

3.2.4 Development approach and issues found

So far, we presented all the features that both the framework and GUI offer, and then we have seen some usage scenarios about different functionalities they provide. However, the design and development process required effort, features to implement and how to do that, and how to schedule the development activities according to their intrinsic dependencies. Hence, in this section, we delve into the GEM software design, in order to provide a clear breakdown for the activities involved together with the effort needed for this thesis project. In the following, for sake of reading, the GEM software design and implementation activity is analyzed, dividing the description into several primary steps.

- **Framework acquaintance**
  As a first step, the GEM framework has been studied, with the purpose of getting familiar with it. All the APIs provided has been analyzed, understanding their context of working, their dependencies and their preconditions; then, some test programs has been implemented in order to see how the framework behaves and how it must be used. This had also the purpose of observing the emerging properties of the whole framework, such as constraints in the code development and arrangement, or rules that shall be taken into account during code generation and integration with extern user code.

- **Requirements elicitation**
  The next step in the software design has been the elicitation of project requirements, that is an important step, defining the perimeter of the software project, that must be respected throughout the development and from which an evaluation for the effort needed may be provided. The diagram reported in Figure 3.6 is the result of this step. As already pointed out in Section 2.3, the GUI does not offer all the framework's functionalities, and this facilities tuning also derives from this step: in fact, listing all the requirements and use cases the software should have had, the limited available amount of time for developing the software has been taken into account, and this led to a functional requirements and use cases adjustment with respect to all the functionalities the GEM framework offers.

- **Support facilities selection and integration**
  It has been a very important step concerning the GUI design, since it was devoted to searching all the relevant libraries of Qt to design the visual aspect of the GUI and to implement visual features, such as drag and drop of components (i.e., signallers, event handlers, state machines and states). In particular, in order to implement the drag and drop feature, the *Draggable Icons Example*[13] provided by the Qt documentation has been analyzed and tested, and then it has been used as the basis for the *Drawing Area* (Section

3.2.1).
Although other facilities and examples regarding drag and drop functionality have been investigated, such as *Drag and Drop Puzzle Example*[14] and *Drag and Drop Robot Example*[15], they were not as useful as *Draggable Icons Example*, since the puzzle example requires to drop puzzle pieces in predetermined positions, robot example requires to drop color icons on robot body only, while draggable icons example lets the user free to drop icons wherever it wants, and this is perfect for GEM draggable components (i.e., signallers, states, etc.).

- **Code Abstraction Layer design**
  This activity involved the design of the core component in the system, that is in charge of defining the logic entities mapped to their corresponding visual representation, and tracking their usage in the GUI, as extensively discussed in Section 3.2.1.
  In GEM software code, the Code Abstraction Layer is composed of several *ADTs* (i.e., Abstract Data Types), represented as *struct* entities through C++ language, each containing the parameters for the component it defines (see Section 3.2.2 for component parameters): for instance, the state *struct* has the name of the state, the three action types (entry, step/state and exit), the transitions connected to it (both outward and inward) and the position the state has in the drawing area. More complex definition requires the state machine *struct*, containing state machine name, its type (i.e., default is primary) and init state, the position of the state machine component in the drawing area, and states, transitions and binds belonging to it.
  When a component *struct* contains the reference to another component (e.g., transitions connected to a state, states belonging to a state machine), the definition of that reference could be of two different types: the *struct* may contain an array of indexes for those components which do not belong to that component only (it is the case of transitions in state *struct* and binds in state machine *struct*, since both connect two components), or it may contain an array of components that belong to that component only (it is the case of states and transitions in state machine *struct*, since both components belong to that state machine only). Furthermore, we need to say that, due to the Qt visual objects handling, each component needs a unique identifier to be distinguished from the others: for components such as signallers or states, their name has been chosen as identifier; instead, for transitions and binds that do not have proper names, a numerical index has been chosen to identify them. So, when dealing with both visual and non-visual components, their unique identifiers (i.e., name or index) are exploited to edit or deleting them.

```
1     struct statemachine {
2     QString name;
3     QString init_state;
4     QString sm_type;
5     std::deque<state> states;
6     std::deque<transition> transitions;
7     QPoint pos;
8     std::deque<int> bindings;
9     };
10
11    struct transition{
12    int index;
13    QString from_state;
14    QString to_state;
15    QString guard_action;
16    QString action;
17    QString arrow;
18    QPoint pos;
19    int w,h;
20    QPixmap pixmap;
21    };
22
23    std::deque<statemachine> statemachines;
24
```

Once defined the *struct* for all the components, we need a container for them. Primarily, the *std::vector* structure was used but, after optimization considerations that belong to Software Engineering, it was replaced by the *std::deque* structure: in fact, *std::deque* structure is a double-ended queue and, in memory, it is allocated in blocks of equal size chained together, unlike *std::vector* that is allocated in a contiguous fashion. This difference in how they are implemented leads the *std::deque* to be useful when growing or shrinking the data collection from one of the two ends, or when dealing with very large data sizes: in particular, in the latter case, the *std::deque* is better than *std::vector* due to the large cost of reallocation required by *std::vector*, and due to the possibility of running out of memory, since a contiguous memory block is always needed. So, state machines, event handlers, signallers, binds and queues are defined as *std::deque*, each one with their corresponding *struct* as type of its *std::deque* (e.g., state machine *struct* for state machines *std::deque*, event handler *struct* for event handlers *std::deque*, etc.), unlike dispatching queue component that has not its own *struct* due to its unique parameter (i.e., queue name), so it is defined as a *std::deque* of string type. States and transitions, as said above, belong to the state machine in which they are contained, so they are defined as *std::deque* with their *struct* type, but

their *std::deque* definition occurs in the state machine *struct* definition (see code section above). So, when accessing an event handler, it is only required to access the *std::deque* referring to event handlers, while accessing a state requires the access to the state machine *std::deque* to retrieve the proper state machine, and then accessing the states *std::deque* inside the state machine accessed before to retrieve the proper state.

- **User Interface design**
  The UI design has been the longest part of the project, since it involved the design and implementation of the entire User Interface in all its parts. As discussed above, in the *Support facilities selection and integration* paragraph, the searching of Qt libraries for implementing the drag an drop actions has been the basis for the implementation of the GUI. Once done that, the creation of the components started. The most demanding job of the UI design has been the transitions (and binds) creation, since they are represented by arrows that connect states (or event producer/consumer in the binds case), so we needed a way to display these arrows in a proper way: after a preliminary review on how set arrows angles and dimensions, the *findChild* Qt API has been chosen to retrieve information about an object (e.g., position, dimension, etc.) using its name (i.e., name of states in transitions case). So, in the case of transitions, when *findChild* Qt API returns position of source and target states, the *createTransition* function, mentioned in Section 3.2.1 when discussing about Drawing Logics, computes the angle, position and dimensions (i.e., height and width) to apply to the arrow figure, called *pixmap* in Qt, relying on information about states position provided by Qt. The entire process described above is repeated when a state, with at least one transition connected to it, is moved: since every state has trace of transitions connected to it, each time a state is moved, Qt exploits these information to recreate all the transitions involved. After that, the implementation of all other components, together with their actions (i.e., creation, editing, deletion), has been done. The binds implementation has been the last step about UI design, and it has been managed with the same logic of transitions described above.

- **Save/Load project implementation**
  After finishing the UI design, with all the components and features implemented, the save/load project feature was realized. We discussed about this functionality in Sections 3.2.1 and 3.2.2, when dealing with Project R/W-er and Project handling. First of all, the Qt documentation part covering the Qt file handling features[16] has been analyzed, and then used as basis for the save/load project design. Once understood how to implement it, the save and load project facilities have been implemented: for this purpose, the *QDataStream* type provided by Qt has been exploited, thanks to its ability to serialize information passed to an object of that type (i.e., all the information contained in Code Abstraction Layer); when loading a project, the information stored in the project file need to be taken out in the same order they were inserted.

- **Code generator implementation**
  The last step of GEM software design has been the code generator implementation: we discussed about that feature in Sections 3.2.1 and 3.2.2, when dealing with code generation. Before implementing the code regarding this feature, a preliminary review about preconditions and post-conditions for components mapping, translation order for CAL objects, code arrangement and directory tree has been performed. Regarding preconditions and post-conditions for components mapping, and translation order for CAL objects, it has been established that signallers, event handlers, dispatching queues and states are the components that can be handled and translated at first, without dependencies; then, after defining states, transitions can be handled and, consequently, state machines. Finally, once defined all the components above, binds can be handled. These conclusions have been reached thanks to the review on the GEM framework APIs and components dependencies (e.g., a bind cannot be defined without definition of components it connects).
  Instead, regarding code arrangement and directory tree, the structure in Figure 3.29 has been designed. When generating code, the GEM software creates the project folder with the same project name chosen when saving project; inside it, three main folders are contained: *gem_framework*, *inc* and *src*: the first one contains all about the framework, with all the header and source files required for using it; the second one contains all the headers about the modelled system (i.e., *private_inc*) and a folder in which the user can add its own header files (i.e., *user_inc*); the last one contains all the source files about the modelled system (i.e., *private_src*) and a folder in which the user can add its own source files, and in which the *main.cpp* file, containing the *main()* function, is created (i.e., *user_src*).

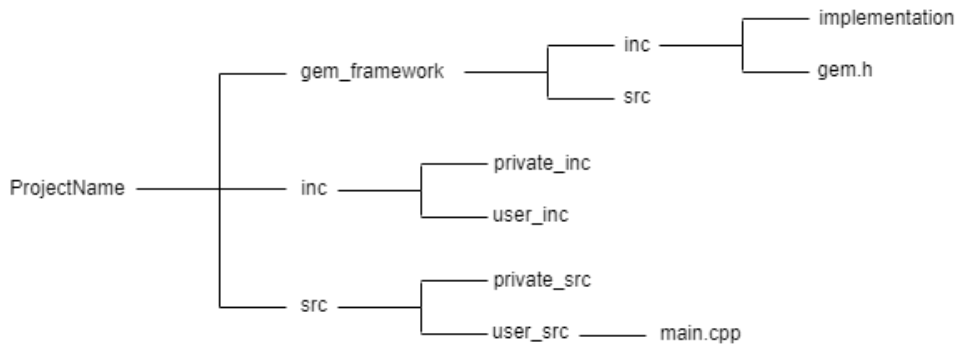

Figure 3.29: Directory tree of generated code.

Even if the generated files contain the behavior of the modelled system, the user should add some code in order to start the system in a proper way: for instance, if inside the *handleEvent* function of event handlers are inserted some particular definitions (e.g., *std::cout* for displaying messages), the user has to include the required header file (*iostream* in the case of *std::cout*). Moreover, the *main()* function is generated in such a way that the program execution stops immediately, due to the presence of *stop* framework API call right after the *start* framework API call; so, if the user wants to check whether the modelled system behaves as expected, it has to modify the *main()* function body too.
The compile process shall be led manually since, at the moment, the GEM software does not provide automatic compiling facility (e.g., make, CMake). However, the automatic compile facility can be enhanced in future versions of the software.
The command to compile the code has to include all the header files (i.e., headers of the framework, headers of the components generated by the GEM software and possible header files created by the user), source files (i.e., sources of the framework, sources of the components generated by the GEM software and possible source files created by the user) and the *main.cpp* file path, as well as the name to assign to the object file created after the compiling process.

Chapter 4

Future works

In this chapter, we delve into possible improvements in the GEM software, both in terms of additional functionalities that the software may provide to the user, and from the standpoint of further facilities that may be integrated in the underlying framework.
It is worth noting that all the features described in Chapter 3 are encompassed by the first release of the GEM software; instead, the features described in this chapter have been identified as possible improvements for future release of the software.

4.1 Improvements to the framework

The GEM framework provides several functionalities to handle event-based systems. However, it can be enhanced with other facilities, such as:

- **Additional state machine types**
  At the moment, the GEM framework supports primary and forwarding state machines, that can be configured (i.e., nested) to represent any kind of behavior. However, it can be useful to let the user the ability to choose a specific type of state machine: for instance, a state machine that behaves like a sequence recognizer could be useful to be already integrated in the GEM framework (e.g., the user should only define the sequence of events to recognize). The PDA state machine (*P*ush-*D*own *A*utomata) could be useful to implement in the set of state machines supported by the framework too: PDAs are classical state machines using a stack as support data structure during operation; primary example in which such state machines are useful is the recognizing of pairwise parenthesis. Finally, Mealy and Moore state machines (see Section 2.1.2) can be another type of state machine to integrate in the GEM framework, since they are widespread.

- **Priority-based events dispatching**
  As discussed in Section 3.1.1, when dealing with dispatching, the GEM framework provide to the user the ability to define as many dispatching units as it wants, in order to exploit parallel dispatching. However, both in serial and parallel dispatching, events are consumed using a FIFO (First In First Out) policy, and without applying any priority to the events.

- **Generalization of arity configuration**
  Another aspect that may be improved is the configuration of the state machines arity: at the moment, the user can set the state machine arity, providing only raw values (e.g., for an integer parameter, the user may pass only values to the API); in some systems it may be useful to have the capability to configure the arity using a broader set of means, allowing the user to implement as much as possible the variability of the system. This is case of any object that is callable: static functions, functors, lambdas, etc. This way, the framework would be enable the user to model the target system using more flexible APIs.

- **Timed synchronization**
  Before dealing with this possible improvement, a brief explanation about synchronized (sync) state is needed: when it enters, it spawns several parallel threads, and then it provides result(s) to the user. The Flynn taxonomy[17] classifies computer architectures, according to the multiplicity of data and instructions they handle: there are *Single Instruction Single Data* architectures, in which operations are executed sequentially; *Single Instruction Multiple Data* architectures, in which the same instruction is executed on different processing units and with different sets of data; *Multiple Instruction Single Data* architectures, in which different instructions works on the same data; *Multiple Instruction Multiple Data* architectures, in which different instructions work on different data. In the sync states context, we denote the *"function"* as the *"instruction"* and the *"parameter set"* as the *"data"*. The sync state enables the handling of different multiplicities of functions and parameter sets, such as: *Single Function Single Parameter Set* (a function returning only one value); *Single Function Multiple Parameter Sets* (a function returning more than one value); *Multiple Function Single Parameter Set* (more than one function returning only one value); *Multiple Function Multiple Parameter Sets* (more than one function returning more than one value).
  A timed synchronization is the combination of a sync state and a timer: after a certain amount of time, so when the timer expires, the system does not wait for the sync state result(s) anymore; it may be useful in real-time systems that have time threshold to respect.

- **Networked dispatching**
  It involves two different systems and, in particular, the communication between them. Let's take into account a remote and a local system: networked dispatching (also known as remote dispatching) is a type of events dispatching in which the remote dispatching unit receives an event from a signaller therein and sends it to a local signaller or dispatcher. It can be useful if we want different system to communicate between them, signalling events one for each other.

4.2 Improvements to the GUI

The set of improvements regarding the GUI can be divided into two groups: the former concerning the functionalities of the framework that are not yet included in the GUI and that would enhance the user capabilities in software design, and the latter concerning the features missing in the whole GEM software.
According to this distinction, the possible improvements to the GUI are:

- **Framework functionalities missing for the user**
  - **Timed signallers**
    We mentioned them in 3.1.1, when dealing with signallers. They are used to signal events of any type (like standard signallers) using a timed-window strategy.
  - **Function signallers**
    We mentioned them in 3.1.1, when dealing with signallers. They are used to signal events produced by a function.
  - **Function caller**
    We mentioned them in 3.1.1, when dealing with signallers. They are used to call functions running on other threads (i.e., implementing the signal/slot strategy provided by Qt framework).
  - **Function callee**
    We mentioned them in 3.1.1, when dealing with signallers. They are used to provide wrappers for functions, which react to remote calls coming from a *function caller* object.
  - **Event filters**
    We mentioned them in 3.1.1, when dealing with filters. They can be applied to signallers, in order to let events be accepted or filtered out, according to the policy defined by the filter applied. Such filtering may be also defined by means of a composed strategy: in this case, the user may define nested filters, that act as composed functions. In GEM framework,

they are available in different configurations: sampling filters, used to generate events every n samples; threshold filters, used to generate events according to a threshold t; range filters, used to generate events according to a range r; timed filters, used to generate events every time interval i; deviation filters, used to generate events according to deviation d from the last valid event; custom filters, used to generate events according to a user-defined filter.

– **Event models**
We mentioned them in 3.1.1, when dealing with models. They are used to arrange the full set of signallers (regardless of their type) used by the process and ease the access to the current state to the other software components.

– **Event waiters**
They are used to receive events of type T from a designated signaller (whichever its type is), forwarding them to subscribed functions according to associated filters (e.g., "forward all events from this signaller having their value equal to 13 to this associated function", or "forward all events from this signaller having their value less than 3.14 to this associated function").

– **Composite states**
We mentioned them in 3.1.1, when dealing with composite states. They are used to allow the user to build multi-layer logic FSMs: when entered, such a state may trigger sub state machine, according to the event that triggered the entering. Once the system exits from a composite state, reset actions may be performed, and the current active state machine is reset.

– **Arity for state machines**
We mentioned them in 3.1.1, when dealing with state machines. The arity of a state machine is a set of parameters that are used to tune the state machine behavior while it is operational; it can be declared when defining state machine or it can be not used at all, defining the so-called empty arity state machine. At the moment, the GEM software supports the empty arity by default.

– **Object-reference state machines**
We mentioned them in 3.1.1, when dealing with state machines. They are also called *object-bound* state machines; that is, state machines in which every action point shall refer an object of type T: such an object reference is included in the arity of the state machine and the user can change it during the state machine life-cycle.

  - **Forwarding state machines**
    We mentioned them in 3.1.1, when dealing with state machines. They are used as the main skeleton of the logic state machine: they provide the ability to forward the input arity, together with the current dispatched event. They cannot be used as input in the binding process, and they shall receive arity tuple from the outside.
  - **Timers**
    We mentioned them in 3.1.1, when dealing with timers. They allow to trigger actions on a timing-basis: they can be *one-shot* (they work only once) or *periodic* (they trigger over a predefined amount of time). They also allow to configure/change the clock type at OS-level (i.e., the period between time ticks).

- **Other functionalities missing in GEM software**
  - **Protected sections**
    They are useful when overwriting files during code generation: when the user starts the code generation process concerning a system whose code has already been generated, files are overwritten, discarding all the changes the user did to the code. Protected sections are, as their name suggests, code sections that are preserved during code generation.
    The most useful part in which insert protected sections is the *main.cpp* file, since it is the unique generated file that, at the moment, cannot be modified through the GUI (all the other files can be edited by means of acting on the corresponding software component).
    In Figure 4.1, the possible protected sections in *main.cpp* file are highlighted: the first one can be used to contain the headers and global variables; the second one can be used to contain local variables or other code lines useful for the user; the third one is the *if* body, so it can contain the actions to perform in case of system initialization failure; the fourth protected section may be used to perform some actions before starting the system; the fifth one can be used to perform some actions after the system started (e.g., the user can insert the blocking of *main()* function, in order to avoid the immediate stop of dispatching activities, until some event(s) come(s), as discussed in Section 3.2.4, when dealing with *Code generator* implementation, at the end of the paragraph); the sixth protected section may contain actions to perform after the system stops; the last one can be used to write whatever the user wants (e.g., new methods/functions declaration).

```
#include "gem_initializer.h"
/*
1. PROTECTED SECTION
*/
int main(int argc, char *argv[]) {
  /*
  2. PROTECTED SECTION
  */
  if( !(__init()) ) {
    /*
    3. PROTECTED SECTION
    */
  } else {
    /*
    4. PROTECTED SECTION
    */
    gfsm::start();
    /*
    5. PROTECTED SECTION
    */
    gfsm::stop();
    /*
    6. PROTECTED SECTION
    */
  }
  return 0;

}
/*
7. PROTECTED SECTION
*/
```

Figure 4.1: Protected sections in *main.cpp* file.

– **Compiling process**
As discussed in Section 3.2.4, when dealing with *Code generator* implementation, at the end of the paragraph, the GEM software does not provide automatic compiling facility, but the compile process shall be performed manually. However, a possible improvement could be the one of making this process automatic: a *makefile* can be exploited to declare all the rules regarding the *make* command, which is used to compile the code and to generate the object file; on top of that, a *CMakeLists* file may be created, and exploiting *CMake* tool we may ease the user in Makefile generation/updating, when writing code outside the GEM software. Furthermore, an additional feature might be allowing to compile the code straight from the GUI, enabling the immediate feedback for the user and saving time during system design.

– **Increased drawing area**
  In Section 3.2.1, we discussed about *Drawing Logics* and, in particular, about *Drawing Area* of GEM UI (see Figure 3.3). At the moment, it can contain few components due to its size, but in future it would be possibly enlarged to contain more components.

– **Exclusion of components from code generation**
  As discussed above, when dealing with *protected sections*, we mentioned the problem of code overwriting: a possible solution could be to let the user choosing the code of which components to generate again (i.e., to overwrite), so that the added code after the previous code generation is not overwritten.

– **Editable *main()* function**
  A possible improvement to the GUI could be the possibility of letting the user to write the *main.cpp* file using the GUI; this way, the user would not need to edit the *main.cpp* code after generation.

Chapter 5

Conclusions

The goal of this thesis is to develop a software capable of handling event-based systems, providing the user the ability of modelling them thanks to an easy-to-use Graphical User Interface and, then, allowing to generate final code representing the system, which is editable by the user, if needed.
An event-based system is a programming paradigm for application design, in which the program flow is determined by external events, the main element of the whole architecture: the event producer detects the events, and notifies them without knowing how they will be consumed; then, the event consumer, that receives the event notifications, elaborates them in a asynchronous way. This is opposed to the polling model, in which the consumer makes repetitive requests until the provider notifies the event.
The software, called GEM (*G*eneric *E*vent-based *M*odeling), exploits the framework developed by the Zirak s.r.l. company, which provides a set of APIs to deal with an event-based system: the framework relies on C++ language, and leverages template metaprogramming extensively, in order to adapt its generic code to the developer needs. For instance, two swap functions, one used for swapping two integer variables content and another one used for swapping two string variables content, can be collapsed into only one in order to make it valid for any data type.
The tool chosen to design and develop the software is Qt framework, given its C++ integration and its academic-use license, but also its ability to ease the handling of GUIs for software and its ability to connect graphic and business worlds easily, through the usage of design facilities, such as signal/slot paradigm.
The GEM software allows the user to establish policies for event signalling, event dispatching and event handling behaviour(s). Events are notified by signallers, each with a type that depends on the value type notified: they are bound to event handlers. Once received, events can be consumed by generic event handlers or by state machines, according to their defined behaviour. Furthermore, the user can decide whether to signal events in a serial or in a parallel way thanks to the

chance of defining multiple dispatching queues: in fact, if the user wants to notify an event at the same time towards different objects (i.e., event handlers and state machines), it can bind the same signaller together with the two target objects using two different dispatching queue.
The GEM software provides the user the ability of saving the project about the modelled system and to resume it later, without the limitation of defining the system all at once.
As we discussed above, the final step is the code generation, that provides a well-organised directory in which the user can find all the objects (with their parameters) translated using the GEM framework; then, the user can edit the code to finalize the system behaviour (e.g., edit the *main()* function).
The User Interface does not provide all the features offered by the GEM framework due to the limited amount of time, but it will be possibly updated later. Moreover, the GEM framework will be enhanced as well, with other features such as synchronized states and priority-based events dispatching. Finally, the User Interface will be updated from a visual interaction point-of-view, making it more pleasant to the user, in addition to the missing features (e.g., timers, filters, etc.).
Summarizing, since the framework is entirely written using high-level programming languages like C and C++, the developer should know the entire set of APIs and all their functionalities before using the framework. That is the reason why the GEM framework has been extended with a easy-to-use GUI, which lets the user to model its system with all the features, *without dealing straight with the framework APIs* and especially *decreasing the amount of time needed to arrange the software code*, thanks to the code generation facility.

As discussed in Chapter 1, event-based systems are present in many industrial fields and have as many applications, such as automotive, avionic/aerospace and operating systems, to cite some of them. Moreover, as discussed in Section 2.3, there are toolkits that deal with event-based systems in different ways and with different features, but they have some drawbacks, such as no ability to deal with generic event handlers, no deep integration with C and C++ programming languages and no chance to generate code through an automatic process. The added value of this thesis is to develop a software capable of managing different software components from which an event-based system can benefit, such as signallers, generic event handlers, state machines and dispatchers. Moreover, some drawbacks from other software on the market may be overcome, such as the ability of handling projects (i.e., save and load project facilities) and the ability of generating code automatically. All the features it provides, together with the benefits it can bring to the industrial fields and applications mentioned above, make the GEM software a promising and rising toolkit, possibly with additional features that would increase its value and usefulness.

In conclusion, I would like to mention the hard and soft skills this thesis allowed to learn: first of all, it allowed to deepened the knowledge about several aspects of Software Engineering, going into detail of its principles and steps, which helped a lot in this project design and implementation; in particular, activities such as tasks (re-)scheduling, effort (re-)evaluation and risks (re-)assessment, performed during the whole project, were fundamental in order to meet the final deadline for this thesis. Furthermore, it allowed to strongly enhance the knowledge about software design and programming, especially with Qt framework and C++ programming language; in addition, it allowed to learn about different theoretical topics, such as event-based systems, state machines, metaprogramming and templates. Finally, it allowed to improve skills about writing a thesis and Latex environment, exploited to write it.

Bibliography

[1] Red Hat. *What is event-driven architecture?* Sept. 2019. URL: https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture. (accessed: 07.02.2022) (cit. on pp. i, 5).

[2] Zirak Website. URL: https://www.zirak.it/ (cit. on pp. i, 17).

[3] Qt Website. URL: https://www.qt.io/ (cit. on pp. i, 15).

[4] The Economic Times. *Definition of 'Software Engineering'*. URL: https://economictimes.indiatimes.com/definition/software-engineering. (accessed: 14.03.2022) (cit. on p. 1).

[5] R.E. Fairley P. Bourque, ed. *Guide to Software Engineering Body of Knowledge*. Version 3.0. IEEE Computer Society, 2014. URL: www.swebok.org (cit. on p. 4).

[6] Anvita Bajpai. *Event-Driven vs Request-Driven (RESTful) Architecture in Microservices*. Dec. 2020. URL: https://www.techtalksbyanvita.com/post/event-driven-vs-request-driven-rest-architecture. (accessed: 07.02.2022) (cit. on p. 6).

[7] Yakindu. *What is a state machine?* URL: https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines?hsLang=de. (accessed: 03.02.2022) (cit. on pp. 6–8, 10, 11).

[8] Yakindu. *What are YAKINDU Statechart Tools?* URL: https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_yakindu_statechart_tools?hsLang=de. (accessed: 08.02.2022) (cit. on p. 11).

[9] OPNET. *OPNET Network Simulator*. URL: https://opnetprojects.com/opnet-network-simulator/. (accessed: 08.02.2022) (cit. on p. 12).

[10] OMNeT. *What is OMNeT++?* URL: https://omnetpp.org/intro/. (accessed: 08.02.2022) (cit. on p. 13).

[11] Gabriella Giordano. *Introduzione ai template*. Mar. 2019. URL: https://www.html.it/pag/375063/template-metaprogrammazione/. (accessed: 11.02.2022) (cit. on p. 14).

[12] Douglas Gregor David Vandevoorde Nicolai M. Josuttis. *C++ Metaprogramming*. Nov. 2017. URL: https://www.informit.com/articles/article.aspx?p=2832416. (accessed: 14.03.2022) (cit. on p. 15).

[13] Qt. *Draggable Icons Example*. URL: https://doc.qt.io/qt-5/qtwidgets-draganddrop-draggableicons-example.html. (accessed: 16.03.2022) (cit. on p. 47).

[14] Qt. *Drag and Drop Puzzle Example*. URL: https://doc.qt.io/qt-5/qtwidgets-draganddrop-puzzle-example.html. (accessed: 20.03.2022) (cit. on p. 48).

[15] Qt. *Drag and Drop Robot Example*. URL: https://doc.qt.io/qt-5/qtwidgets-graphicsview-dragdroprobot-example.html. (accessed: 20.03.2022) (cit. on p. 48).

[16] Qt. *Part 6 - Loading and Saving*. URL: https://doc.qt.io/qt-5/qtwidgets-tutorials-addressbook-part6-example.html. (accessed: 17.03.2022) (cit. on p. 51).

[17] Wikipedia. *Flynn's taxonomy*. URL: https://en.wikipedia.org/wiki/Flynn%27s_taxonomy. (accessed: 23.03.2022) (cit. on p. 54).