POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Redesign and improvement of a test system configuration tool

Supervisor
prof. Matteo Sonza Reorda

Candidate
Giuseppe Pipero

Internship Supervisor
Company: Infineon Technologies AG
Subsidiary: KAI GmbH
dr. Benjamin Steinwender

December 2021

"Ai miei genitori, che mi hanno insegnato l'umiltà e il rispetto.
A Laura, che ha sempre creduto in me.
Ad Alberto, Giulia, Noemi, Rosy, Samuele, Vanessa
e a tutte le persone che mi amano.
A Benedetto, per tutte le volte che mi ha salvato
quando stavo per cadere."

Abstract

Life testing is a fundamental procedure to know the reliability of the devices, to ensure their operation in all conditions. At Kompetenzzentrum Automobil- und Industrie-Elektronik (KAI), a new test system is under development, which is described by a test-plan created and verified by a visual software. Each test-plan is composed of finite-state machines (FSMs), which allow a modular composition and extension of a test sequence. The states of an FSM contain Lua code, a script language used for executing test routines.

Over the time, it became necessary to adapt the tool to build test-plans to new technologies, make it more performing and improve it to be used in a more complex context. The tool is divided into two separated modules: the *Test-plan Builder*, which provides a graphical user interface (GUI) to graphically create a test-plan; the *Test-plan Checker*, used to verify the correctness of the whole test structure.

This thesis focuses both on the user interface and the functionalities. The goal is to support the hierarchical design of the FSMs, in order to create a simplified view of more complex test-plans and to better compact their representation. Moreover, the entire visual platform is redesigned to make it more responsive and user-friendly. This is obtained by creating the interface from scratch, making a more efficient graph design tool, improving the code writing thanks to a real-time code suggestion and completion, and adding several enhancements like a modular layout, an undo-redo system, and a clever search function. Finally, it is necessary to update the *Checker* to support the Lua version 5.3, improve the syntax error recognition ability, and the verification speed.

Contents

Chapter 1

Introduction

Stress testing is a core activity of semiconductor industries to understand the reliability of their devices, typically performed using automated test systems. At KAI, a test system called modular power stress (MoPS) is currently being studied and developed [1], [2]. It is a distributed test system capable of running different tests on multiple Devices Under Test (DUTs) controlled by a single local machine. MoPS is divided into two hierarchical layers: the Host Layer and the Target Layer, as shown in Figure 1.1.

Host Layer Represents the host machine, which controls the overall test flow, communicates with the targets, reads the measured data.

Target Layer Contains the list of targets. Each target executes the test, drives and monitors a DUT. They are accessed via Ethernet from the Host Layer.

The test system consists of FSMs, one for the Host and one for each Target, and it is stored using the JavaScript object notation (JSON) format. The FSM attached to a Target defines the test code to be executed on the device, based on the triggered state. The states of the FSM contain Lua code, while the transitions allow moving to the next state to be executed. Through specific Lua commands it is possible to trigger a transition to the specified Target FSMs, in order to define the test execution flow.

Before the implementation of the test system configuration tool, the test-plans were manually written in a text editor. Writing complex test-plans was really hard and uncomfortable, and checking the correctness of the FSMs structure and Lua

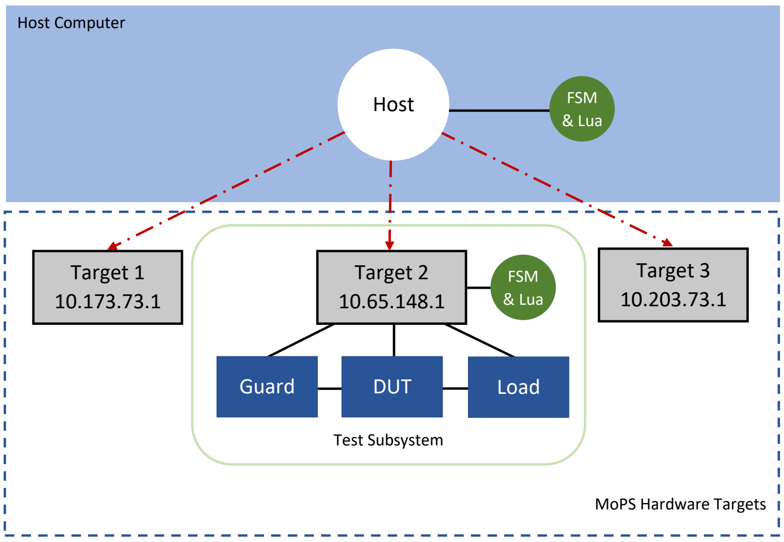


Figure 1.1. MoPS Software Architecture (Image created by Benjamin Steinwender [3] and modified by Giuseppe Pipero)

code was frustrating. The current tool solved this problem by providing a GUI to design the FSMs, and a verification system to check for errors, giving a visual feedback to the user [4]. This really helped test engineers to write complex MoPS specific test plans and verify the entire structure, obtaining a robust and error resistant test-plan.

1.1 Motivation

The current *Test-plan Builder* simplifies drawing FSM diagrams for the MoPS layer. With time, however, it became necessary to optimize the test design process to meet the growing need for performance and design. Creating a complex test-plan could be difficult due to the slow responsiveness of the drawing tool, and the high delay of the checking phase in case of large FSMs.

The tool should provide an easier graph representation and a more modern and user-friendly GUI, guaranteeing a smoother graph design mechanism and better performances in terms of speed. Additionally, it should offer the possibility to define really large and complex test-plans through sub-FSMs organized hierarchically. Finally, the updates of the Lua language require an adaptation of the Lua verification system to support the last introduced features.

1.2 Goals

The goal of this thesis can be summarized into two sub-tasks. First, redesigning the user interface to improve usability and introduce new visual features. This includes the following steps:

- Creating a modern GUI,
- Changing the FSM graphical representation,
- Improving the visual graph designing process, in terms of fluency and simpleness,
- Introducing hierarchical representation of FSMs,
- Creating a better code suggestion and code completion mechanism,
- Improving the test-plan error visibility giving an overview of all generated errors,
- Generating a better automatic layout for graphs, and
- Designing a search/replace panel.

The second task includes the test-plan validation update and performance improvements. This step should be accountable of the following:

- Upgrading the Lua parser to recognize Lua 5.3 features,
- Updating the validation mechanism to check the correctness of sub-FSMs,
- Improving the error recognition supporting new types of errors,
- Improving the undo/redo functionality to make it more robust, and
- Optimizing the overall performances by reducing the validation time.

The entire application is developed using Java, relying on the JavaFX platform for representing the GUI.

1.3 Problem Statement

This thesis focuses on the graphical and functional improvements of the existing *Test-plan Builder*. Therefore, the following questions arise:

- How to improve the user interface and the graph design?
  - How to organize the main layout?
  - How to visualize the different FSMs?
  - How to optimize the responsiveness of the FSMs' building process?
  - How to improve the FSM representation?
- Which strategy should be adopted to implement hierarchical FSMs?
  - How many hierarchical layers should be supported?
  - How to graphically represent a sub-FSM?
  - How to save/export a project with hierarchical FSMs?
  - How to validate the test-plan structure?
- How to improve the test-plan validation?
  - How to support Lua 5.3?
  - How to speed up the checking process?
  - What other Lua errors should be recognized?

1.4 Thesis Outline

This section gives some advice, to better understand the structure of this thesis. Chapter 1 represents the introduction, where the motivation, goals, and problem statement are explained. It is really important for getting an overview of the thesis project. Chapter 2 presents some basic concepts about the technologies used to implement the system. An expert in coding can skip most of these concepts, but it is important to read Section 2.7 and Section 2.8. The former is related to a fundamental library widely used in the project, whereas the latter helps to understand how the user interface was created. Chapter 3 explains the theoretical concepts of the system and of the test-plan structure, which are fundamental to understand the goal of the thesis. Chapter 4 proposes a solution for implementing the tasks described in Section 1.2, whereas the specific implementation, related both to the functional and graphical aspects, is explained in Chapter 5. Finally, Chapter 6 contains an evaluation of the developed tool, and the conclusion follows in Chapter 7.

Chapter 2

Technologies used

This chapter illustrates some basic concepts of the technologies used in the project, focusing on the reason why they were adopted. The key features of the chosen programming language are also explained. Finally, the framework to build the User Interface (UI), and its relevant libraries are presented.

2.1 JSON

JSON is a data-interchange format built as a collection of attribute-value pairs and array data types, used to store and transmit data objects. It is human-readable and easy to be parsed and generated by machines [5]. Compared to eXtensible Markup Language (XML), it has some advantages, as it is easier to read and write, and supports arrays [6].

Gson is a Java library developed by Google to convert Java Objects into their JSON representation, and vice versa [7]. Its main advantage is the ability to parse a JSON string directly to a class instance, based on a matching between the document attributes with the class ones. In some cases, however, the Gson default representation is not suitable, especially when dealing with library classes (i.e. *DateTime*), or when an object contains custom nested objects, and a proper output format is required. Gson allows, therefore, the registration of custom serializers and deserializers for a specific class. Listing 2.1 shows how Gson can be used with a user-defined serializer: lines one to five define the serializer for the *DateTime* class. It implements the method *serialize*, which receives the source object and returns a *JsonElement*. Lines nine to ten register the adapter to the *DateTime* class of the *GsonBuilder* instance.

Listing 2.1: Gson example

```
private class DateTimeSerializer implements JsonSerializer<DateTime> {
    public JsonElement serialize(DateTime src, Type typeOfSrc,
                                 JsonSerializationContext context) {
        return new JsonPrimitive(src.toString());
    }
}

//...

GsonBuilder gson = new GsonBuilder();
gson.registerTypeAdapter(DateTime.class, new DateTimeSerializer());
```

2.2 Lua

Lua is an efficient, powerful, and lightweight scripting language, designed to be used as a general-purpose extension language. It provides a small set of general features that can be extended to fit different contexts and problem types. Thus, it presents a mechanism of *fallbacks* that allows programmers to extend the semantic of the language. For example, it does not support object-oriented features, like classes and inheritance, but they can be implemented with *metatables* [8]. It also implements advanced features such as automatic memory management with incremental garbage collection, and coroutines, to support parallel programming [9].

Lua is the language adopted by the MoPS system (see Chapter 1) for executing test routines [3]. MoPS has recently switched to Lua 5.3, as it implemented the integer data type, the native support for bitwise operation (previously possible only through a specific library), and support for both 64-bit and 32-bit platforms [10].

2.3 Reflection

Reflection is a feature of the Java programming language which allows an executing Java program to examine itself and manipulate internal properties of the program (i.e. object attributes) [11]. Reflection has several drawbacks, especially in terms of speed and traceability, since reflection calls are slower than direct calls, and it can be tricky to find the portion of code that caused a reflective method call to fail. Anyhow, it can be adopted where the performance is not important. It is used in the project to define a flexible Settings page and to alter the application configuration with a user-defined file. The same goal can be achieved without exploiting Reflection, but it would require much more effort, and a lot more lines of code. Listing 2.2 shows how Reflection can be used to get the name and value of the fields of an object.

Listing 2.2: Reflection example

```java
public void getObjectFields(Object o){
    for (Field f : o.getClass().getDeclaredFields()){
        String fieldName = f.getName();
        Object fieldValue = f.get(o);
    }
}
```

2.4 Singleton

The Singleton pattern restricts the instantiation of a class to one single instance, usually accessible globally [12]. The class is responsible for instantiating itself only once. The constructor is declared *private* or *protected*, to ensure that the class can never be instantiated from outside itself. It is typically used to manage access to shared resources, like a database, or the configuration settings of an application. An example of a Java Singleton class is shown in Listing 2.3.

2.5 Stream

The Stream application programming interface (API) is one of the main features introduced in Java 8 [13], and it is used to process collections of objects. A Stream is not a data structure and never changes the original one. It is made of a pipeline of intermediate operations lazily executed, and each intermediate operation returns a stream as result; the pipeline ends with a terminal operation which consumes the stream and returns the actual result[1]. This allows the expression of the code in a more compact and declarative style. Listing 2.4 shows how Streams can be used to convert a list of *Person* objects into a *HashMap* where the key is the person identifier, and the value is the person object.

[1]A Guide to Java Streams: https://stackify.com/streams-guide-java-8/

Listing 2.3: Singleton class example

```java
public class MySingleton {

    //create class instance
    private static final MySingleton instance = new MySingleton();

    private EagerInitializedSingleton() {
        //...
    }

    //get class instance
    public static MySingleton getInstance() {
        return instance;
    }
}
```

Listing 2.4: Stream example

```java
public Map<Integer, Person> listToMap(List<Person> list) {
    Map<Integer, Person> map = list.stream().collect(
            Collectors.toMap(Person::getId,Function.identity()));
        return map;
}
```

2.6 Thread pool

A thread pool is a software design pattern for achieving concurrent execution of operations. Using multiple threads for performing short-lived tasks can affect the performance due to the creation and destruction overhead. A thread pool, instead, can be used to execute multiple tasks on a fixed number of threads, that are created just once when the pool is instantiated and then are reused to execute an operation. Therefore, the overhead due to the thread handling is restricted to the initial creation of the pool, resulting in better performances and stability[2]. It also allows better control of the threads and their lifecycle. The thread pool is initialized with a fixed number of threads and then, each time a new operation is scheduled, a task is put in a queue and will wait until a thread becomes available to execute it.

[2]Thread pool: `https://w.wiki/42nH`

Java provides several implementations of a thread pool, each one with different characteristics [14]. Among these, the *ExecutorService* allows the creation of a thread pool with a fixed size, and defines some methods to control the state of each task and to get the result of its execution. An example of the implementation of an *ExecutorService* is shown in Listing 2.5.

Listing 2.5: Thread Pool example

```
ExecutorService executorService = Executors.newFixedThreadPool(5);
Future<String> future = executorService.submit(() -> "Hello World");
// some operations
String result = future.get();
```

The first line instantiates a thread pool with five threads, while the second line submits a runnable task for execution and returns a *Future* object representing that task. The *Future* can be later used to get the execution result of the task.

2.7 LuaJ

LuaJ is a lightweight Java interpreter for the Lua language, which implements a virtual machine to run the Lua source code[3]. It also defines a parser, based on a JavaCC grammar. It is the core element of the code verification and error recognition of the *Test-plan Builder*. The parser analyzes the code and creates a structured representation of it, in order to recognize and access the different parts of the source code.

LuaJ implements a strategy to traverse the source code based on the *Visitor* pattern [15]. It can be used by defining a class which inherit the *Visitor* class and overrides a *visit* method for each type of syntax element to be visited. When a specific syntax element is encountered, the corresponding *visit* method is triggered.

An example of a *Visitor* capable of recognizing function call expressions is shown in Listing 2.6.

[3]LuaJ: http://luaj.sourceforge.net

Listing 2.6: Visitor example

```java
public class FunctionCall extends Visitor {

    @Override
    public void visit(Exp.FuncCall exp) {
        super.visit(exp);

        //scan passed arguments
        for (Exp arg : exp.args.exps) {
            //do something
        }
    }
}
```

2.8 JavaFX

The following GUI libraries were evaluated to implement the graphical part of the project:

AWT It is the oldest Java GUI framework, mature and well documented. Anyhow, it is outdated and not the best option to create a rich and modern user interface.

Swing It is the successor to AWT, providing a richer set of graphical components. It was widely used in the past, but has now been replaced by more modern frameworks. The existing *Test-plan Builder* was built using Swing.

JavaFX It is the latest UI library, offering a more modern layout rather than the other tools. It supports mobile devices, therefore, any application written in JavaFX can be executed on a mobile operating system. Since it is one of the newest Java GUI libraries, it is suitable for creating more complex and modern user interfaces than the older frameworks like Swing and AWT. The only disadvantage is that, being younger than its counterparts, it offers a smaller set of third-party components.

JavaFX defines several interfaces to directly bind the components to the data, in order to have a robust and consistent layout [16]. Some of these interfaces are:

Property wraps an existing Java object, adding some functionalities for observing and binding it. Each property has methods to listen for changes to its value, Listing 2.7: Property usage example

```java
DoubleProperty zoomLevel = new SimpleDoubleProperty(100);
zoomLevel.addListener((observable, oldValue, newValue) -> {
        doSomething(newValue.doubleValue());
});
```

link it to another property, get and set its value. Listing 2.7 shows how a *ChangeListener* can be attached to a property in order to be notified each time that its value changes.

Bindings are a way to link objects together, enforcing a relationship in which one object is dependent on at least another one. When an object is bound, its value is automatically updated each time the object to which it is bound to is modified, without worrying about the implementation of the updating process[4]. The main strength of JavaFX is that even the GUI components can be bound. This means that the value shown by a component can be directly linked to a property and automatically updated each time that the property changes. *Properties* and *Bindings* are widely used to build the project UI consistently, and to automatically update it. An example of how a *Label* can be bound to a string representing the title of a graph is shown in Listing 2.8.

Listing 2.8: Bindings example

```java
public class Graph {
   private final StringProperty name = new SimpleStringProperty();
    //...
}

//UI showing a graph
public class GraphPanel{
   public GraphPanel(Graph graph){
        //...
       Label name = new Label();
       name.textProperty().bind(graph.getName());
   }
}
```

The third-party JavaFX components used in this project are:

[4]JavaFX Properties and Bindings: https://edencoding.com/javafx-properties-and-binding-a-complete-guide

FXGraph A graph visualizer supporting node dragging and resizing, zooming, panning, directed edges[5].

RichTextFX A memory-efficient and customizable text area, supporting different text styling and syntax highlighting[6].

ControlsFX A library providing a rich set of custom UI elements and APIs for JavaFX[7].

[5]FXGraph: `https://git.io/JuzSD`

[6]RichTextFX: `https://git.io/Juz9B`

[7]ControlsFX: `https://controlsfx.github.io/`

Chapter 3

State of the Art

This chapter presents the state of the art of the current system. First, an overview of the MoPS test structure is presented. Afterward, the test-plan designing process and its output format are fully described. Finally, the test-plan verification strategy is explained.

3.1 The MoPS FSM

The MoPS test system is described by FSMs, one for the Host and one for each Target. An FSM consists of states, containing Lua code, and events, to determine the next state to be executed. When the MoPS system is in a state, its Lua code is executed. At the end of the execution, a check for triggered events is performed. If an event related to an existing transition is found, the FSM changes its state moving to the destination node of that transition. If there are no triggered events, the *@else* one is executed [3].

The typical structure of a MoPS FSM is composed of the IDLE state, where the execution starts, and a *start* event from the IDLE to the next state. The IDLE state has no code and its *start* event is triggered to move to the next state connected to it. The start state is also the final one and, at this point, the controller can be stopped, or the FSM structure may be changed. Thus, in order to design a valid FSM, each node in the path must be able to reach the IDLE state. Each state has an *@else* event that can be connected to itself or to another state. This state is executed when no other transitions are triggered, to keep the execution flow. There are two different types of events:

- Hardware events, expressed with the prefix @, that are triggered in the hardware.
- Software events, triggered in the software via the corresponding APIs used in Lua code.

An example of a MoPS FSM is shown in Figure 3.1.

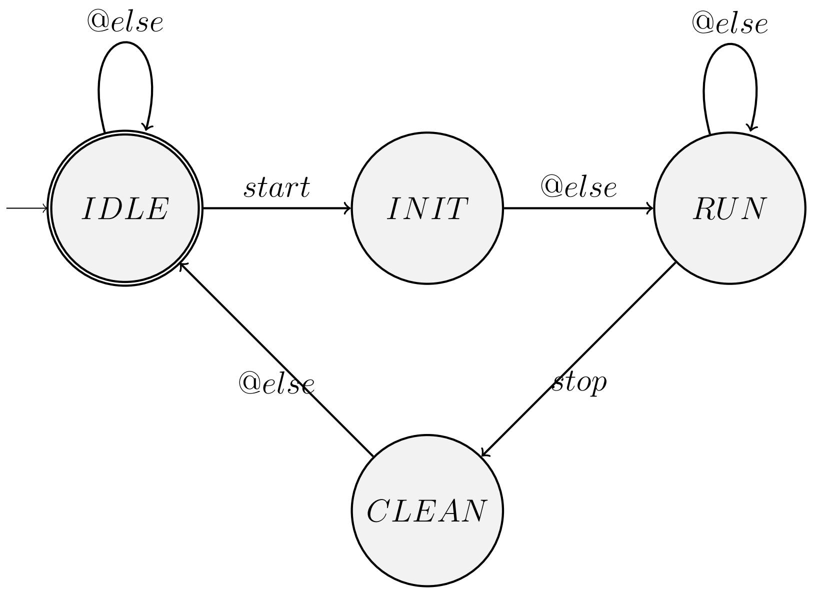


Figure 3.1. MoPS FSM example

3.2 The MoPS TestPlan

The MoPS system defines a specific format to store a test-plan, using the JSON syntax. The goal of the *Test-plan Builder*, indeed, is to produce an output conforming to this format. Figure 3.2 describes the data structure of a test-plan.

The most relevant fields are:

FSM Represents an FSM. It is identified by the *type* (Host, Target), and by a *label*. An FSM is also composed of a list of states and events.

Function Represents a state of an FSM. It is described by a *name*, to uniquely identify the state, and by the *code*, which is the Lua code invoked when the MoPS system executes the state.

Transition Is a transition between two nodes. When it is triggered, the connected node is executed. It is identified by three attributes: *current*, representing the source node of the transition; *event*, describing the transition name; *next*, representing the target node.

OvenPlan Represents the connection point between the hardware devices and the corresponding FSMs. The relevant fields of an OvenPlan instance are:

Slot The location of the DUT in the test system.

DUT The name assigned to the DUT.

HW Target IP or hostname address of the physical device to be linked to an FSM.

uC FSM The FSM linked to the specified Hardware (HW) target.

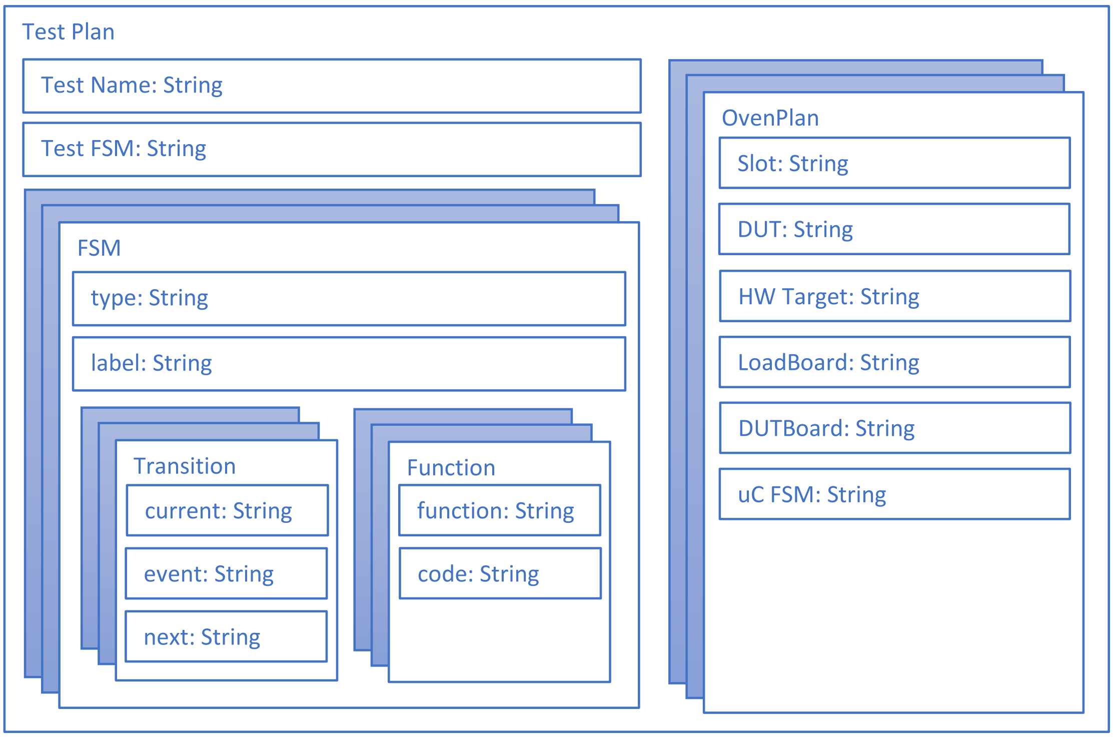


Figure 3.2. MoPS TestPlan structure (Image created by Klaus Plankensteiner [4] and modified by Giuseppe Pipero)

3.3 MoPS and SAM APIs

MoPS-CORE and software architecture for MoPS (SAM) APIs provide the required functions to control and execute a test. In particular, SAM is used in the Host controller to coordinate the Target machines, whereas the MoPS-CORE API is used to test the hardware devices belonging to the CORE family. These APIs are described in an online documentation, which is fetched by the *Test-plan Builder* to populate the network data cache, used by the autocomplete system and during the graph check.

3.4 Electronic Data Sheets

The Electronic Data Sheet (EDS) is a file which describes the specifications of a HW target. These files are located on a web server and used by the *Test-plan Builder* for the test-plan verification process (see Section 5.5.4). The most relevant fields of an EDS are:

hwInfo Describes some information about the device, like the MoPS-CORE API version supported by the Micro Controller (µC), the IP, and the MAC address of the target device. The IP address (or the hostname) is used in the OvenPlan to select the target device to be mapped to an FSM.

events Represents the list of hardware events supported by the device. This information is exploited by the checker to verify if the hardware events used in the FSM are valid.

modules Defines the list of supported modules that can be instantiated, and the possible instance names. Among the modules provided by the MoPS-CORE API documentation, only the ones specified in this list can be used.

3.5 TestPlan generation

This section summarizes the test-plan designing phase of the existing system. It provides a GUI containing two main components: a toolbar and the main panel. The toolbar defines some actions to interact with the graph, whereas the main panel, implemented as a tabbed layout, is used to render the primary parts of the application. The core part is the TestPlan tab, where it is possible to define the FSMs. The TestPlan is divided into two fixed graph areas, one showing the Host machine and another one containing the list of Targets. Therefore, the Host is always present and only one Target at a time can be visible. The area containing the Targets is enclosed in a tabbed panel, showing the list of available Targets. Moreover, a new target graph can be added via the *plus* tab. An example of this interface is shown in Figure 3.3.

3.5.1 FSM drawing

Each graph area is split into two parts:

Graph Panel Provides a drawing space to draw and connect nodes. Each node is represented as a rounded square, and the IDLE state is the start and stop state, thus, in order to define a valid FSM, a loop must be formed and each node must be able to reach the IDLE state.

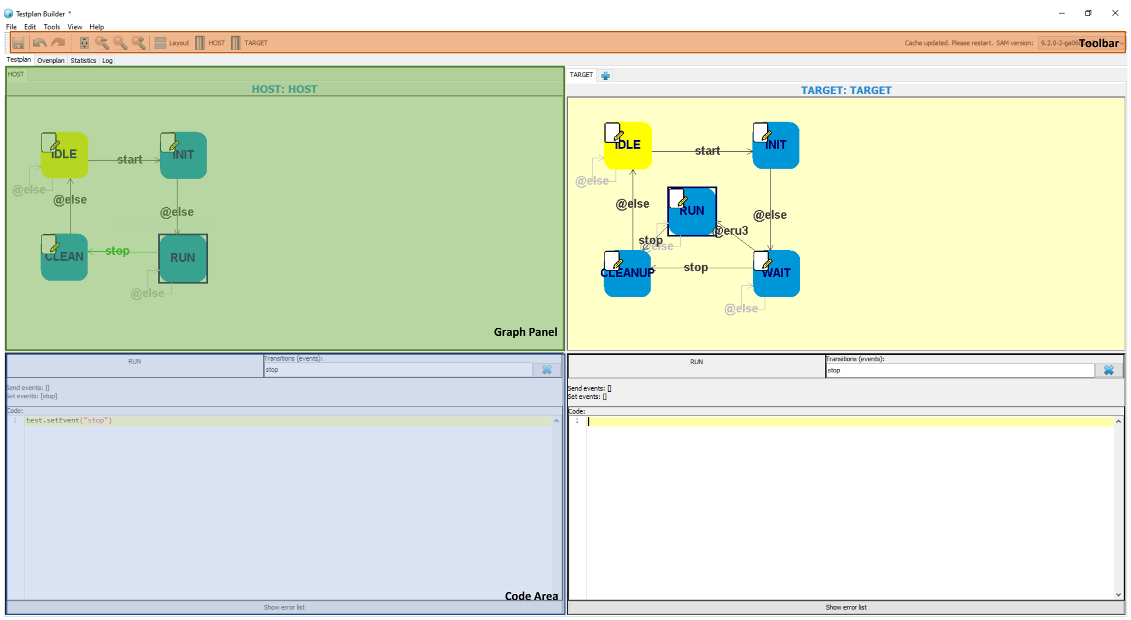
Code Area Contains a text area with a Lua syntax highlighter, and it is used to show and edit the code of the selected node. Code warnings/errors are highlighted and listed in an error panel at the bottom of the component.

The drawing panel is represented by a (possibly) infinite canvas, zoomable and pannable, whose size is dynamically increased each time a node goes over the border. The main graphical elements that can be drawn to create an FSM are:

Node Represents an FSM state. It is internally described as a class containing a name (that must be unique), and the Lua code as a string. The main graphical properties are the color, size and position in the canvas. It can also be selected, resized and dragged in the Graph Panel.

Edge Represents an FSM transition, mainly described by a name, a source and a target node, that must be unique among all edges starting from the same node. It is not allowed to have multiple transitions between two nodes. The list of transitions of the selected node is visible above the Code Area. This is the only place where edges can be renamed or removed, since transitions are not selectable.

Note It is a graphical comment used to describe a part of the FSM. Its goal is to increase the readability of the graph, but it is not part of the FSM, thus it is not exported in the test-plan file.



RUN

Transitions (events):
stop

Send events: []
Set events: [stop]

Code:

```
test.setEvent("stop")
```

RUN

Transitions (events):
stop

Send events: []
Set events: []

Code:

Figure 3.3.   Test-plan Builder - Main interface

Multiple nodes selection is allowed through shortcuts. Selected nodes can be dragged, copied or removed. Copied nodes, that preserve the transitions among them, can be pasted in the same or in another graph. Additionally, a select area is implemented to select all nodes enclosed in the drawn area.

When the application starts, or a new project is initialized, two graphs are created by default: a Host FSM named *HOST*, and a Target one named *TARGET*. The graph names must be unique.

3.5.2 Lua code input

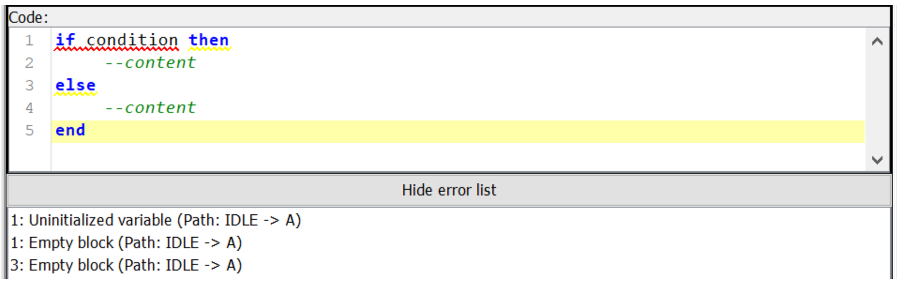
When a single node is selected, its code is shown in the Code Area of its FSM. This area contains the list of transitions having the selected node as source, a text area to display and edit the Lua code and, below, a list showing code warnings/errors. This text area is implemented using a third-party library, named RSyntaxTextArea[1], which provides line numbering and automatic Lua code highlighting. The auto-suggestion of modules, instead, is achieved through the AutoComplete library,

[1]RSyntaxTextArea: `https://git.io/vprA5`

released by the same creators of RSyntaxTextArea. This guarantees that the two libraries can be easily configured and interconnected[2].

The autocomplete system is not automatically activated while the user is typing, but it must be opened through the shortcut Ctrl + Space. The suggested modules are retrieved from the SAM and MoPS-CORE APIs, downloaded and cached by the application from their remote location. Additionally, the variables storing class instances are suggested. Other variables and custom functions, instead, are not recognized.

The code editing process is tightly connected to the *Test-plan Checker* module, in order to identify syntax errors in runtime. This heavy operation, executed in background, is triggered every time that the code, or the graph changes. The process uses the LuaJ visitors, implemented in the *Checker* part, to analyze the code and find possible errors, that are returned to the Code Area. These errors are shown in the error list and highlighted in the code, as shown in Figure 3.4.


Code:
```
if condition then
	--content
else
	--content
end
```
Hide error list

```
1: Uninitialized variable (Path: IDLE -> A)
1: Empty block (Path: IDLE -> A)
3: Empty block (Path: IDLE -> A)
```

Figure 3.4. TestPlan Builder - Code area

3.5.3 OvenPlan

Defining the OvenPlan is one of the main steps of the test-plan generation. It resembles a table where each entry creates a connection between a HW target, and its FSM. The structure of an OvenPlan instance is defined in Section 3.2. An FSM can be linked to more than one HW targets, if they all belong to the same family, but a hardware device can be coupled only to one FSM.

When an FSM is linked to a HW target (based on its hostname or IP address), its EDS is loaded, and the corresponding modules are retrieved from the MoPS-CORE

[2]AutoComplete: https://git.io/JuNGa

API cache, based on the version indicated in the EDS. In this way, these modules are recognized in the Code Area of the linked FSM, and also suggested in the autocomplete panel. In the case of multiple HW targets linked to the same Target, only the modules in common between their EDSs are loaded.

3.6 TestPlan verification

The verification of the test-plan is perhaps the most important and critical part of the tool. Generating a robust and working test file is, indeed, one of the main goals of the project. This section describes more in detail how the test-plan is verified in terms of correctness. The *Test-plan Checker* is implemented as a separated module, therefore, it can also be executed independently of the *Builder* part. The verification process receives the input in a suitable data structure, which emulates the format described in Section 3.2. If the *Checker* is used as standalone software to check a test-plan file, its JSON content is parsed into the supported data structure; if the *Checker* is used in the *Test-plan Builder*, instead, the data is already provided in the correct format. In the *Test-plan Builder* most of the checks are performed during the export phase. In runtime, indeed, only the code syntax and the graph structure are analyzed. In the following, the different type of checks are described.

3.6.1 FSM check

Checking the FSM means verifying the correctness of its structure, and the possible execution flow, defined by triggering events. The following checks are performed:

Dead end nodes These are nodes which cannot reach the IDLE state.

Lonely nodes These are nodes that cannot be visited, since an incoming transition is missing.

Undefined Hardware events The list of valid hardware events is defined in the EDS file. If a Target FSM uses an event that is not present in the list, an error is detected.

Undefined Software events These events are triggered in the Lua code using specific functions. An event can be triggered in the own FSM, using the function *setEvent(name)*, or from the Host to one or more Targets, using the function *sendEvent(name)*. If a software event on a Target is not triggered, or the Target FSM triggers an invalid event on itself, an error is generated. This process does not check if the events triggered by the Host, using either *setEvent* or *sendEvent*, are valid.

3.6.2 Lua check

The Lua check recognizes syntax errors as well as bad coding practices, in order to generate a correct and valid script. It is important to highlight that the code of a single node is not independent, but it is tied to the previous nodes in the same path. A variable, for example, can be declared in a state and used by a subsequent one. Moreover, multiple paths, from the start to the final state, are possible. Therefore, the code of all nodes in a path is merged and checked, and the same operation is performed for each possible path. Since proprietary APIs are used, it is necessary to prepend an additional code to the merged script, containing the definition of the API functions, otherwise, a call to an API function will not be recognized by the code analyzer, as its declaration is missing.

Once the script is ready, it is analyzed by the LuaJ parser, described in Section 2.7. A *Visitor* is used for each type of element to be recognized. This approach makes the Lua checker a modular system, as more visitors can be added to improve the recognition. The following situations, related to general Lua syntax, are analyzed:

Always the same It is related to a statement that produces always the same value (i.e. $a + 0$ or $a == a$).

Bad coding practice It is not a syntax error and does not lead to a wrong code, but some code practices should be avoided since they can generate vulnerabilities and stability issues. For example, the *goto* instruction should not be used and, therefore, an error is raised in that case.

Empty code block It indicates a block that the user has (most probably) forgotten to fill, like an empty function or a conditional branch without any code. As well as the bad coding practices, it is not a syntax error, but it is important to report this situation to the user.

Undefined functions It is related to the call of a function that is neither defined in the current scope, nor in the outer visible scopes.

Undefined class Lua is not an object-oriented programming language, but it is possible to emulate the concept of a class by defining meta-tables [17]. Class methods can be defined and accessed using the syntax *instance:functionName()*. Therefore, it is important to analyze the correct use of classes and methods.

Undefined tables In Lua is possible to define tables and assign functions to them, that can be called using the syntax *table.functionName()*. Thus, it is necessary to verify that used tables and table functions are previously declared.

Infinite loops To guarantee the generation of a working code, a check for infinite loops is performed. In particular, it checks if the condition is always true or if no variable of the condition statement is modified in the body, preventing the script to leave the loop block.

Uninitialized variables Last but not least, it is necessary to check if used variables are initialized in a visible scope.

In addition to these checks, some project-specific Lua errors, related to the usage of MoPS and SAM APIs, are recognized:

MoPS-CORE API It provides both modules and classes. Therefore, a strategy to check if functions called from MoPS modules or classes are valid is implemented. Valid elements are computed based on the MoPS-CORE API assigned to the considered FSM.

SAM API It provides functions for the Host FSM. As well as for the MoPS-CORE, it is necessary to check if functions used from a SAM module exist. Valid SAM functions are retrieved from the cached SAM API, based on the selected version.

3.6.3 OvenPlan check

The OvenPlan, explained in Section 3.5.3, has specific constraints and, therefore, must be checked before exporting the test. In particular, the DUT field must be unique, since it represents a physical device, and there should be at most one OvenPlan entry for each HW target. The following checks are performed to verify the OvenPlan structure:

Missing values It is not allowed to have entries with empty fields. Therefore, a check is executed to guarantee that all rows are completely filled.

Double values The fields *Slot* and *DUT* must be unique, since they represent respectively the location of a DUT, and a physical device to be tested. The same check is also applied to the *HW Target*, as a physical device can be assigned to only one Target FSM.

Target FSM It is necessary to check if the used FSMs exist and if they are of type Target. This is very difficult to happen if the visual builder is used, as it allows specifying only the right FSMs, but it could occur when the *Checker* is used to verify a test file, since it may have been manually modified.

Finally, it is important to highlight that the OvenPlan checking is not executed in runtime, but only before exporting the test.

Chapter 4

Proposed solution

This chapter describes the implementation choices of the new project, focusing on the functionalities to be kept from the previous tool, and the graphical and functional parts to be improved.

4.1 Implementation choices

The first question that comes to mind is how to improve the tool. It is better to modify the existing project or to redesign the entire platform starting from scratch? This is sometimes a difficult choice, which depends, among other things, on the available time and on the desired level of freedom in the project modifications. Modifying the existing project means to be much more constrained on the previous technologies and, in some cases, finding and improving an unknown code related to a specific functionality may need more time than recreating it. Since the goal of the thesis doesn't focus only on the back-end functionalities but also on the UI, in order to build a modern and flexible interface, with a more responsive graph design tool, the choice of recreating the entire project from scratch was followed.

Java was chosen as the main programming language, as it provides easy and rich frameworks to build the UI, and it would also allow keeping some functionalities of the previous software. Therefore, the *Test-plan Builder*, was totally recreated, whereas the *Checker* part, already working, was extended and improved. Regarding the GUI framework to adopt, among the alternatives described in Section 2.8, JavaFX was chosen, as it is more suitable for modern layouts.

4.2 Analysis of improvements

4.2.1 Graphical improvements

Concerning the application layout in general, a more modern and single-page application can be designed, optimizing the available space and improving the overall usability of the system. In the previous tool, indeed, only two FSMs can be shown at the same time: the Host and one Target. The goal, instead, is to provide a modular interface where it is possible to open all the desired graphs and reorganize their position on the page. Another functionality to be improved is the graph designing process: it is difficult to deal with a large graph since it becomes too lagging. Furthermore, pan and zoom operations can behave differently to provide better usability of the design tool. When zooming in, for example, it does not follow the caret position, and this can be annoying to the user. This goal can be achieved using a lightweight and responsive third-party library to manage the graph, extended and modified to meet the project requirements.

Concerning the graph designing, the FSM graphical representation could be improved: currently, to preserve the structure of the MoPS FSM, a loop must be created to define a valid graph. This may reduce the readability, and the ease of interaction with the graph, especially in the case of a large FSM, where many transitions must reach the IDLE state. In this thesis, a representation without loops is proposed, thanks to the implementation of final states. This representation offers the opportunity to implement a better auto layout system, as in the current tool, when a test file is imported, it is rendered using a circular layout, which is not clearly readable. By removing the loop, instead, it is possible to introduce an automatic hierarchical layout, placing the nodes as a tree structure.

The autocompletion mechanism is also an area that could be improved. In the current tool, it can be triggered only through a specific shortcut, and the dynamic suggestion is limited to the class instances. Additionally, since the scope is not considered, a class instance is suggested even if it is not reachable, confusing the user who will think that, instead, the element is usable in that piece of code. The idea is to improve the overall suggestion system triggering it automatically, while the user is typing the code, in order to guarantee an experience similar to an integrated development environment (IDE). Furthermore, the dynamic recognition should be expanded to recognize custom variables, functions and class instances based on the current scope.

Concerning the visual feedback of errors, the system could be better: currently, the only way to check if a graph contains some errors is to open it and search for nodes with a red border, indicating the presence of errors. When dealing with many Targets and/or large graphs, it may be difficult to have immediate feedback about the correctness of the test-plan. This issue can be addressed by implementing a list of all detected errors, in order to give an instant notification to the user about the existing errors, without the need to analyze each graph. One last important graphical feature that could be improved is the search panel, as it does not allow searching either locally in the open Code area, or to specify the set of FSMs where to search.

4.2.2 Functional improvements

The second part of improvements is related to the functionalities. First, since the Lua version used in the MoPS system was updated to version 5.3, it is necessary to adapt the *Test-plan Builder* to support it. The supported version does not depend on the tool itself but on the library used to parse the Lua code. In fact, LuaJ can recognize up to version 5.2, and the current tool is constrained to this. The search for a new Lua parser capable of supporting the latest Lua versions was unsuccessful, since most of them are outdated, and the most updated still remains LuaJ. Therefore, the idea is to extend the current LuaJ project, as the code is freely accessible, in order to recognize the syntax elements of Lua 5.3.

The test-plan validation, already robust, can be improved: the check related to the events used is not complete, since it does not verify the correctness of events triggered from the Host (to itself or to a Target), and this could lead to a wrong test file in output. Therefore, a live check on the validity of used events is necessary. Furthermore, some controls are currently performed only during the export phase, and the user does not realize the errors until the test is exported. Thus, it is better to perform more live checks (i.e. the OvenPlan structure) to improve the robustness of the tool. Another challenge concerns the performance improvements of the validation process, in terms of speed. The current system takes a few seconds to check the entire test-plan in case of a large graph. Since a check is executed at every change, this could be frustrating as it reduces the usability of the tool. After an analysis of the system, it turned out that the bottleneck in the verification is the execution of the LuaJ visitors to find code errors. To improve the performances a parallel solution can be adopted, in order to execute all checks concurrently, as it may drastically reduce the execution time. Finally, the undo/redo functionality could work better: it currently makes a snapshot of the entire test every time that something is modified. The proposed solution is to implement a local history of changes for each graph and for the OvenPlan, so that each undo or redo operation affects only a specific area of the project. Moreover, a page showing all changes can be implemented, to directly restore the graph to a specific snapshot.

4.2.3 Sub FSM

A separate explanation must be done for the main functionality implemented in this thesis: the hierarchical design of an FSM. It is a graphical feature which allows the definition of very complex graphs, compacting the representation and simplifying the readability. The general idea is to have a different kind of node that can be opened to put other states inside it. In order to start implementing this feature, several questions come to mind. How a sub-FSM should be graphically represented, internally and externally? How many hierarchical levels should be supported? To answer these questions it is possible to start from a first assumption: a sub-FSM internally holds a common FSM, with entering transitions coming from the parent graph, and exiting transition directed to the parent. Therefore, a sub-FSM can be drawn as a node containing input and output ports. Each transition directed to that node determines an input port, whereas each transition leaving the sub-FSM (internally) creates an output port. Figure 4.1 shows a mockup of the described idea.

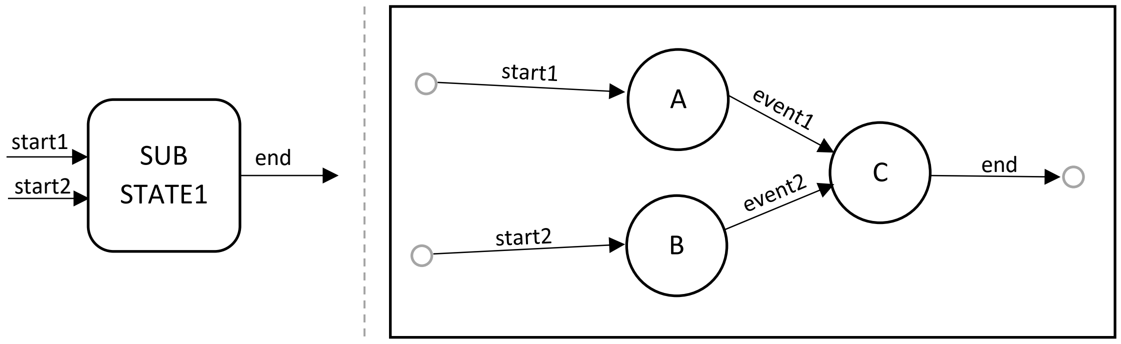


Figure 4.1. Sub FSM external (left) and internal (right) representation

To better understand how it works, it is important to notice that a sub-FSM is not independent, but it is strictly connected to the parent, since the execution flow from the IDLE to a final state in the root graph must be preserved. This means that the parent should provide a transition to connect itself to a specific node of the sub-FSM, and the latter must define an exit point to come back to the parent FSM. After considering different alternatives, it turned out that the best solution to represent it externally is through a node with multiple entering and exiting transitions, respectively related to input and output ports. Internally, instead, the same data structure used for an FSM can be used, with the only difference that it should contain some entering points, one per input port, and an exit point, to connect a node to an external one. Thanks to this implementation a sub-FSM is considered a normal FSM, and it can be rendered in a Graph panel and opened as a new tab. At this point, the second question is not important anymore, as a sub graph is stored and manipulated as a normal graph and, therefore, can contain other sub-FSMs inside, obtaining a (possibly) infinite number of nested layers. It is worth noting that this is a graphical feature only, since the original plain structure of the FSM must be preserved when the test-plan is exported, as the software used to execute it does not support hierarchical FSMs.

Chapter 5

Implementation

This chapter presents the implementation details of the new *Test-plan Builder*, based on the proposed solution described in Chapter 4. First, the extension of the LuaJ library and the adaptation of the graph design library to meet the MoPS FSM structure are presented. After that, an overview of the user interface, with the related modules and their interconnections, is shown. Then, these modules are explained from a more detailed point of view. The explanation continues with a detailed description of the sub-FSM, the core modules of the application, concluding with the checker improvements.

5.1 LuaJ extension

The current *Test-plan Builder* relies on LuaJ to verify the code and check for syntax errors. As explained in Section 4.2.2, the MoPS system moved to Lua 5.3, whereas the maximum version supported by LuaJ is 5.2. Therefore, the LuaJ library was extended to support the bitwise operation recognition, the main new feature of Lua 5.3 (see Section 2.2). To achieve this goal, some preliminary research about how LuaJ works was necessary. It turned out that the LuaJ parser works using a grammar compiled with JavaCC, which is a parser generator for Java applications. It takes a grammar file as input, containing lexical specification and parsing rules, and generates a Java parser. Thus, the grammar was modified by adding the bitwise operators of Lua 5.3 (*and, or, xor, left shift, right shift*), and the unary *not* operator to the corresponding parsing rule, as shown in Listing 5.1. In this way, each operator is converted into a specific token so that the parser can recognize it. Then, the new grammar was compiled into a Java program using JavaCC and imported in the LuaJ project, replacing the old parser with the new one.

It is important to notice that LuaJ is not only a parser, but it is able to run any Lua code. However, this improvement affects only the parser, since it is the part useful for the project goals.

Listing 5.1: LuaJ bitwise operators

```
int Binop():
{}
{
    |  "&"  { return Lua.OP_BAND;  }
    |  "|"  { return Lua.OP_BOR;  }
    |  "~"  { return Lua.OP_XOR;  }
    |  "<<"  { return Lua.OP_SL;  }
    |  ">>"  { return Lua.OP_SR;  }
    //...
}
```

Finally, the modified LuaJ was deployed and imported as a library in the new *Test-plan Builder* project.

5.2 Graph design mechanism

Another preliminary action is related to the implementation of a visual component to draw an FSM and interact with it. After some research, it emerged that there are no external libraries with the specific purpose of creating and representing an FSM. Therefore, the decision was to use a third-party and open-source library as a starting point, customizing it based on the project needs. The *FXGraph* library, a simple graph visualizer, was chosen for this goal. The advantage of following this strategy instead of creating the entire component from scratch is that the back-end logic to store and render the data is already implemented. Furthermore, this tool already offers an infinite, zoomable and pannable pane where the graphical components are rendered and, above all, provides a structured way to implement different kinds of shapes and edges. The library, indeed, defines two interfaces, *ICell* and *IEdge* that are implemented by two corresponding abstract classes which provide the main properties (name, size, color, etc.) of nodes and transitions. By inheriting these classes, it is possible to define concrete classes that deal with the graphical properties only, in order to represent a specific shape. Nodes and transitions are stored in separated lists defined in a *Model* class. The core module is, instead, the *Graph* class which interconnects the *Canvas* (the pane where the items are drawn) with the *Model* and returns the component to be attached to the UI to display the graph.

Before implementing custom shapes, the default abstract class of a cell was modified adding a string to hold the Lua code. Then, three types of nodes have been created:

CircleCell Represents a standard FSM state with a circular shape.

CommentCell Represents a rectangular area to write a comment. It uses a *WebView* to render a string comment written in HyperText Markup Language (HTML). This allows customizing the text changing the size, adding a list of items, etc.

RectangleCell Defines the external representation of a sub-FSM. Thanks to a different shape, it can be easily distinguished among the other nodes.

Some interaction events are already applied for each node and the *Canvas*, to allow node dragging, resizing, as well as panning and zooming of the drawing area. These events, mainly related to mouse and keyboard actions, were extended to improve the interaction with the graph. In particular:

- The *Canvas* click event was improved to allow the insertion of a state in the clicked area, and to draw a transition between two nodes.
- The selection of multiple nodes has been implemented, as well as the dragging and deletion of the selected nodes.
- The selection of a single edge was supported.
- A select area was implemented in the *Canvas* to select all nodes enclosed in the drawn area.

The select area was created as a rectangle with a semitransparent color, and its visualization depends on the evolution of the mouse interaction with the *Canvas*, caught by the following events: *mousePressed*, *mouseDragged* and *mouseReleased*. When the user simulates the drawing of a shape using the mouse, the *mousePressed* event is called: at this point, the select area is shown and relocated on the clicked point. While the user is dragging the mouse keeping the button clicked, the *mouseDragged* event is triggered. The difference between the new position of the cursor and the start position of the select area is used to compute the size of the rectangle, which is modified accordingly. At the same time, all nodes enclosed in the rectangle are selected. A node is considered inside the rectangle either if it intersects the area, or if it is totally contained on it. Finally, the action ends when the user releases the mouse button: the *mouseReleased* function is called, and the select area is hidden.

A final improvement concerns the transition drawing. The graphical representation of an edge was modified by adding an *attach point* at the beginning of the line. By pressing and dragging this point it is possible to connect the line to another node. Furthermore, with a double click on the point, the line is removed.

5.3 Main interface

This section describes the user interface of the system, starting from a high-level description of the different modules, proceeding with a detailed analysis of each of them. The GUI, inspired by a typical IDE, was designed as a single-page application, with the aim of maximizing the user experience and making the system more user-friendly, trying to avoid as many external dialogs as possible. It is divided into four main areas, as represented in Figure 5.1:

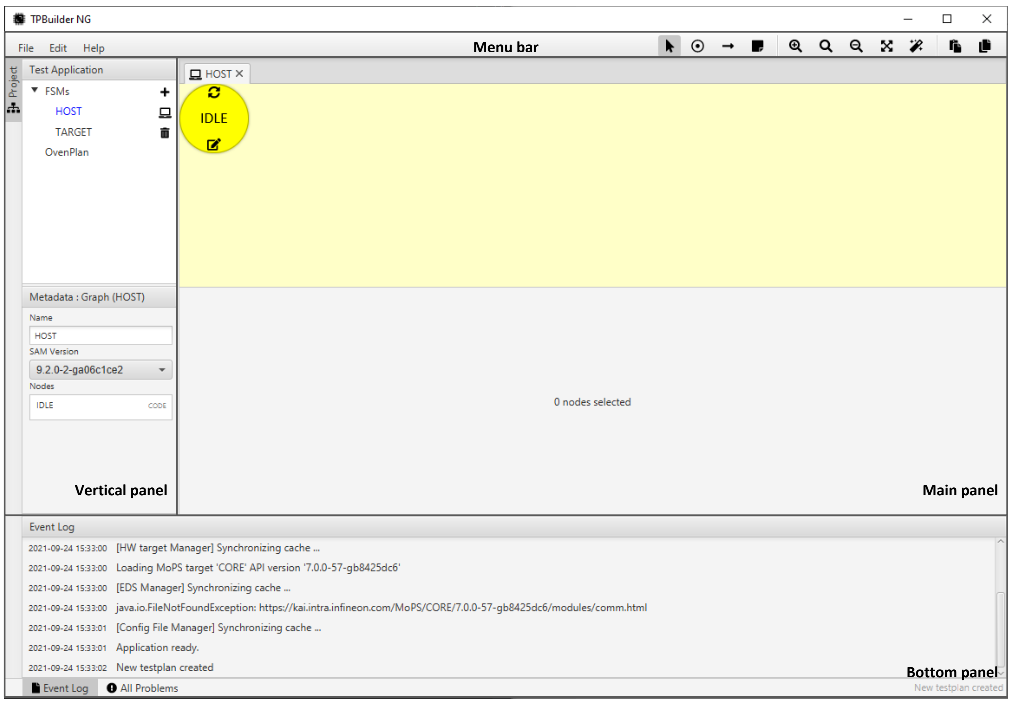
Menu bar Contains the Menu items, and a toolbar to interact with the graph area.

Vertical panel Can be shown and hidden through a specific button on the left side of the window. It is made of two fundamental components: the Test Application, which contains the project tree used to navigate among the application modules (FSMs, OvenPlan); the Metadata panel, used to show and edit the properties of the selected element, like a graph, state or transition.

Main panel It is the main area of interaction with the system, where the principal components, like graphs and the OvenPlan, are displayed.

Bottom panel It is a horizontal panel used to quickly access some functionalities like the Event log, and the Error list of the whole test-plan. The specific feature can be shown/hidden through the corresponding button located in the status bar at the bottom of the application window.

Thanks to the separation of the window in main areas, it is possible to improve the application usability by keeping all features on one single page. The bottom panel, indeed, allows access to some important functionalities, like the application log, without losing the focus from the main view, giving to the user the opportunity



```
Event Log
2021-09-24 15:33:00  [HW target Manager] Synchronizing cache ...
2021-09-24 15:33:00  Loading MoPS target 'CORE' API version '7.0.0-57-gb8425dc6'
2021-09-24 15:33:00  [EDS Manager] Synchronizing cache ...
2021-09-24 15:33:00  java.io.FileNotFoundException: https://kai.intra.infineon.com/MoPS/CORE/7.0.0-57-gb8425dc6/modules/comm.html
2021-09-24 15:33:01  [Config File Manager] Synchronizing cache ...
2021-09-24 15:33:01  Application ready.
2021-09-24 15:33:02  New testplan created
```

Figure 5.1. Main interface

to modify the system while reading the log. Furthermore, the application modularity is improved, as the programmer can easily add additional visual components to these panels with a corresponding button to enable/disable them. The secondary panels can also be collapsed to have the main component in fullscreen. In the next sections, each UI area is fully explained.

5.3.1 Vertical Panel

The Vertical panel is a core element of the system, as it contains the project structure and properties. It can be opened via the *Project* button in the left bar of the window. This panel contains two components, the Test Application and the Metadata.

The Test Application displays the project structure, organized as a *TreeView*. Each test-plan is composed of a Host FSM, one or more Targets, and an OvenPlan, and they are accessible from this component. All FSMs are grouped in the *FSM* node of the project tree, where a *plus* icon can be used to create a Target FSM: if clicked, a new, editable, item appears in the tree, where the user must insert the FSM name. Then, pressing Enter the changes are committed, otherwise everything is discarded. The inserted name must be unique among all FSMs, and a notification error is shown in case a duplicated name is entered. By clicking on an entry of the project structure, the associated component is displayed in the main panel. An example of a simple project structure where a new FSM is being inserted is shown in Figure 5.2.

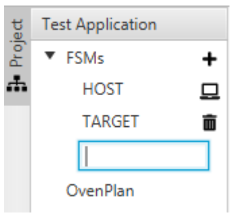


Figure 5.2. Example of a Project structure

The Metadata section is used to show and modify the properties of the active component. It improves the organization of the UI and its ease of use, as the characteristics of an element are concentrated on a single area. Each list of characteristics of a component is referred to as a Property Sheet, of which there are three types:

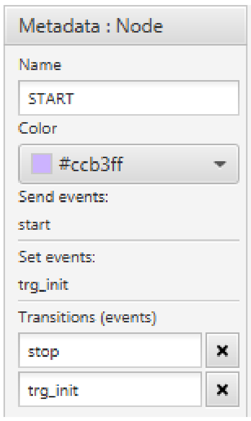
FSM Property Sheet Contains the name, SAM version (only for the Host FSM), and the list of nodes of the active FSM or sub-FSM. By clicking on a node in the list, it will be selected and centered in the graph area. It can be comfortable, indeed, to directly jump to a specific node, especially in the case of a large graph. Finally, this is the only point where it is possible to change the desired SAM API version.

Node Property Sheet Contains the name and color of the active node, as well as the list of transitions which start from the selected node. The name must be unique among all nodes of the same graph (including the children FSMs), and a notification is raised in case the user tries to rename a node using an existing name. Each transition in the list can be renamed or deleted. Furthermore, the list of events triggered by the selected state is displayed. In case the active node is of type *Comment*, the only editable property is the color. An example of a Node Property Sheet is shown in Figure 5.3.

Edge Property Sheet Shows the name of the selected edge, as well as the name of its source and target nodes.

5.3.2 Main Panel

The Main panel is the widest area of the application window which shows all the core components of the system. It is mainly used to interact with the graphs and



Metadata : Node

Name

START

Color

#ccb3ff

Send events:

start

Set events:

trg_init

Transitions (events)

stop ✕

trg_init ✕

Figure 5.3. Node Property Sheet

the OvenPlan. All the other features are accessible from the secondary panels, in the same application page, or through dialogs. Therefore, two kind of pages can be displayed in the Main panel: the one for showing the graphs, and the OvenPlan table.

The Graph Page is the area where the selected FSMs are shown. It is implemented as a list of tabbed panels (called *TabPane*), in order to provide a modular and flexible layout. When an FSM is selected from the project tree, and there are no other FSMs displayed, a *TabPane* is inserted in the Graph Page and a new *Tab* rendering the graph is added to it. If instead, at least a *TabPane* is already present, a double click on the FSM in the project tree is required to open it as a new *Tab* in the last used *TabPane*. Thanks to this strategy, a *Tab* can be moved into another *TabPane*, to freely organize the view and open all the desired FSMs at the same time. This approach increases the system usability, as the user can concurrently open all the graphs to work on. Figure 5.4 shows an example of a Graph Page with three opened graphs, shown one next to the other.

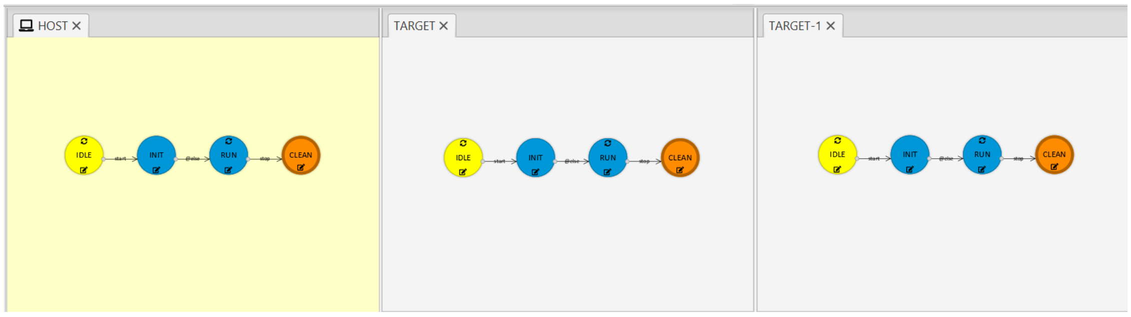


Figure 5.4. Layout organization example with three *TabPanes*

The content rendered in a *Tab* is referred to as a Graph Panel. It is the core element of an FSM, as it allows the representation and interaction with the graph. It is made of two components: the Graph area, where the graph is displayed, and the Code area, used to write the Lua code to be executed on the selected node. Before going into the details of these components, it is important to explain how the graph representation was changed. A standard node, holding Lua code, was implemented as a circular shape with the name in the center. It shows a loop icon on top of the circle, to represent the default *@else* transition pointing to itself. Furthermore, two icons indicating the status of the node can be displayed at the bottom of the circle: the first is shown if the node's code is empty, whereas the second appears if the Lua code contains some errors. An example of a node containing all icons is shown in Figure 5.5.

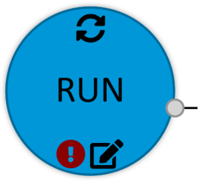


Figure 5.5. FSM node example

Another change involves the graph structure. In the MoPS system, the IDLE node is the start and the final state of the FSM, and a loop is required to obtain a valid graph. Therefore, the graph structure was changed to avoid the manual creation of a loop by introducing final states, which are distinguishable by a darker border. It is important to notice that the loop is removed only graphically, as the standard test structure must be preserved. To guarantee this representation, an *@else* transition from the final states to the IDLE is actually inserted and hidden, thus the user does not notice the presence of the loop, and the default behavior is preserved, since the *@else* edge of the final state will be automatically triggered in hardware to jump back to the first node. Figure 5.6 clearly shows the explained structure: from the visual (and the user) point of view, the CLEAN state is the final one, and it is implicitly connected to the IDLE one through an edge which is not rendered (represented in the figure as a dashed transition).

Graph Area

The Graph area, representing the first half of the Graph Panel, is a rectangular shape where the corresponding FSM is shown. The content of each *Tab* in Figure 5.4

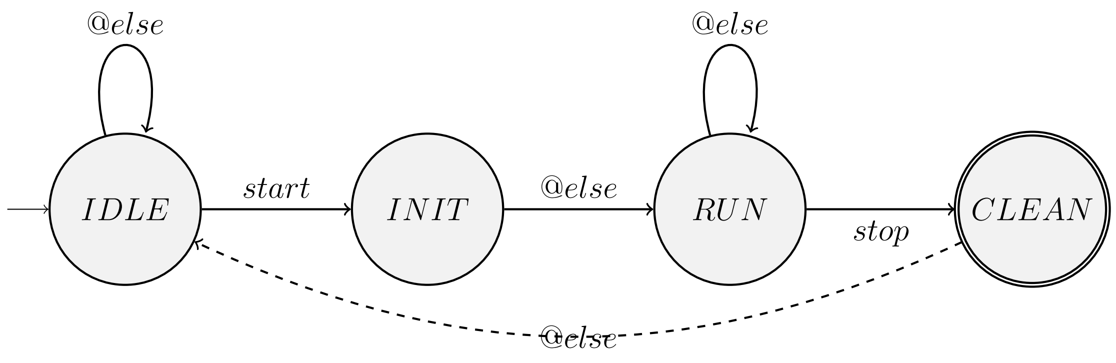


Figure 5.6. New graph structure example

is a Graph area. The element rendered inside that area is the *Canvas* generated by the modified *FXGraph* library, as described in Section 5.2. Therefore, the user can directly interact with the drawing area to visualize, draw and modify an FSM. The graph visualization is supported by simple mouse actions: it is possible to pan the area using the *right-click*, and to zoom towards the part pointed by the caret by scrolling the mouse *wheel*. The *Toolbar*, shown in Figure 5.7, provides the main actions to interact with the graph *Canvas*. Some of these actions are also accessible through shortcuts, to increase the application usability.

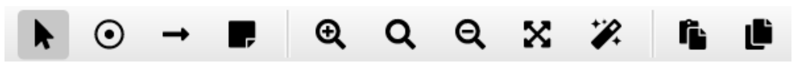


Figure 5.7. Application Toolbar

The first four items of the *Toolbar* are used to choose the type of element to be added to the graph. Only one of these items can be selected at a time. The aim of the other items, instead, is to adjust the graph visualization and to copy/paste the selected items. In the following, all graph interaction actions are fully explained.

Select nodes The item ➤ is chosen when the user wants to navigate among the graph, without doing specific actions. Furthermore, the cursor is used to draw a rectangle to select all nodes enclosed on it. Multiple items can also be selected/deselected by clicking on a node while holding the ⇧ key, in order to select the desired set of nodes.

Add a node To add a standard node, the item ⊙ can be selected. The cursor icon of all active graph areas becomes a cross, to give visual feedback to the user which indicates that an *insert* action can be done. After clicking on a point of the *Canvas*, a dialog (represented in Figure 5.8) appears to request the node properties, which are the name, the color, and a flag to indicate if the node should be a final state. The name must be unique, therefore a different value is requested each time the user inserts an existing one.

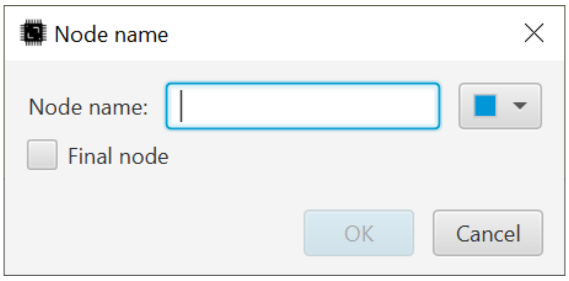


Figure 5.8. New node dialog

When the node properties are confirmed, the coordinates of the clicked point are retrieved and used to insert the node in the desired area. The same action can be triggered through the shortcut Ctrl+click on the *Canvas* area.

Add a transition The arrow button ➞ can be selected to switch to the edge drawing mode. At this point, the user can draw an arrow in the *Canvas*, and the line is shown in the meanwhile. To add a valid transition, it must be drawn between two nodes, otherwise the line is discarded and deleted from the area. This operation works thanks to the mouse listener events attached to each graphical node, and through the *LineContext* class, which stores the source and target nodes of the drawn transition. First, the *mousePressed* event associated with the clicked node is triggered. Then, if the edge drawing mode is enabled, the state is saved as a source node of the *lineContext* object. After that, if the user releases the line dragging on another node, the *mouseEntered* event of the destination state is called, and the latter is used as target node of the *lineContext*. Finally, a dialog appears to request the edge's name, which must be unique among all edges having the same source node. When confirmed, the information stored in the *lineContext* is used to insert the transition in the graph. It is also possible to draw an edge holding the Ctrl key while dragging the cursor in the graph *Canvas*. An existing transition can also be connected to a different target node by pressing on its attach point, in the source state, and dragging it to another node. Then, a dialog appears to eventually change the name of the moved edge.

Add a comment A comment is a pure graphical element used to add a text note into the graph. It can be useful to describe a specific part of the FSM, or to remember something to do. It can be added through the item of the toolbar. A comment node emulates the *sticky note* layout, and contains a *WebView* to render pure HTML text. Figure 5.9 shows an example of a comment.

Figure 5.9. Example of a comment

Adjust the zoom The zoom can be adjusted by changing the scale of the graph *Canvas*. This operation is typically done using the mouse wheel, which is more comfortable. Anyhow, the toolbar provides three buttons to increase , decrease , and reset the zoom. The latter is the most useful as it sets the scale to 1 and moves the *Canvas* to have the graph centered in the graph area, which is the initial configuration of each FSM layout. These three actions are also accessible through the shortcuts Ctrl + +, Ctrl + -, and Ctrl + 0.

Fit to space The aim of the item (Ctrl + R) is to fit the graph in the available graph area. In this way, the user can visualize the entire FSM in the current space. This functionality is implemented by computing the right scaling and shifting value of the graph *Canvas*: first, the coordinates of the lowest and the highest points in the area are computed. Then, the width of the current graph area is divided by the range between the minimum and the maximum x values of the computed points, to obtain the correct scaling value to fit the graph to the available space with respect to the x-axis. The same operation is applied considering the height and the y axis, and the smallest among the two values is used as a scale factor. Finally, the *Canvas* is aligned to the first node (the one with the smallest coordinates) in order to see the whole FSM in the available space.

FSM autolayout The autolayout is a powerful functionality which places the nodes based on a tree structure. This function is automatically used when a test-plan is imported, since there is no information about the nodes' placement, but can also be accessed through the toolbar item . Thanks to the removal of the loop, the FSM can be structured as a tree: first, the root node is visited, and each child is recursively visited to explore all nodes. The nodes in the same layer are placed in the same row (or column, depending on the tree orientation) and spaced apart.

Copy and paste Copy and paste is a core action that simplifies the drawing process. All properties of the copied nodes are cloned, whereas the name is modified appending a number, to keep the names unique. When the selected nodes are copied, through the shortcut Ctrl+C or the button ⎘, a new graph *Model* (the object used to store the graph data, as explained in Section 5.2) is created, to temporarily store the structure of the copied area. The states enclosed to the copied area are added to the *Model*, together with the edges among those nodes. The *Model* is then serialized using *Gson*, to obtain a deep copy, represented in JSON, of the structure. The serialized copy can be pasted into the same or another FSM, using the shortcut Ctrl+V or through the button 📋. After that, the *Model* represented in JSON is parsed and visited, and each time a node/edge is encountered is added to the graph. The convenience of using the serialization is that a deep copy of the *Model* is computed, and it also easily scale to a large number of nodes to be copied.

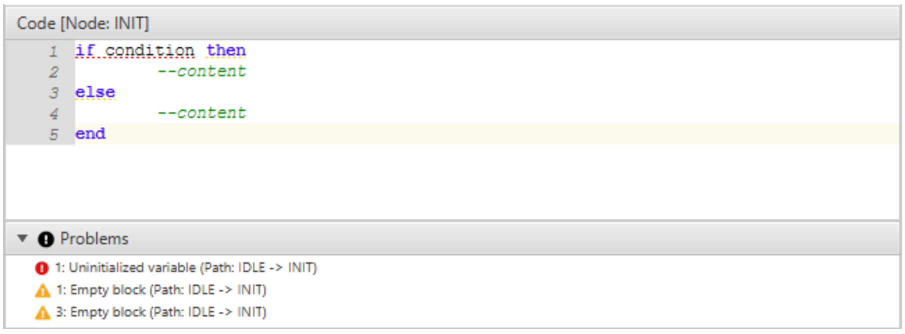
Elements deletion The last important feature related to the graph interaction is the removal of graphical elements. Selected nodes can be deleted by pressing the Del. key. First, all edges connected to these nodes are removed and only then the states are deleted from the graph. If a transition is selected, the deletion action will remove it from the graph.

Code area

The second half of the Graph Panel is referred to as Code area. It is implemented using the *RichTextFX* library which supports, among the other features, custom text styling and a line number indicator. It does not perform automatic syntax highlighting for a specific programming language, therefore it must be implemented manually by applying a custom style to a specific range of characters of the text. To achieve this goal, a set of regular expressions to match all the relevant parts, like strings, comments, data types, etc., of the code, was defined. Then, the corresponding styling rules, defining custom text colors for the recognized parts, were created and added to a stylesheet. At this point, the custom event listeners supported by the *RichTextFX* library were exploited: an event is triggered each time that the text in the area changes, but only after that 100 ms have passed from the last typed character. This strategy reduces the computational cost as it avoids continuous calls of the same operation. When the event is triggered, the text is matched against the defined regular expressions, and the corresponding styling rule is applied to each found match. Furthermore, a code autocompletion mechanism was implemented to simplify the code writing, as explained in Section 5.5.5.

The same listener also executes the *Checker* to the current FSM, to recognize the code errors defined in Section 3.6.2. A detailed explanation about how the FSM is checked can be found in Section 5.6. The recognized problems, distinguished among errors and warnings, are highlighted in the code and listed in the *Problems* section, located below the Code area. Clicking an entry of the list, the Code area jumps to the affected line and selects the error in the code.

Finally, a context menu was added to the component to support advanced text manipulation actions. Some of them are already built-in, like the copy/cut & paste, and the undo/redo functionalities. Other useful features, instead, were created to increase the usability of the coding process, like the indentation, duplication and comment of the selected text. An example of a Code area containing some errors and warnings is shown in Figure 5.10.



Code [Node: INIT]

```
1  if condition then
2          --content
3  else
4          --content
5  end
```

Problems

- 1: Uninitialized variable (Path: IDLE -> INIT)
- 1: Empty block (Path: IDLE -> INIT)
- 3: Empty block (Path: IDLE -> INIT)

Figure 5.10. Code area

OvenPlan table

The OvenPlan table is accessible from the Project structure of the Vertical panel, and opened in the Main panel of the application. Its content is rendered as an editable table which emulates the structure of the OvenPlan described in Section 3.2. Therefore, the table contains a column per each field of an OvenPlan entry. A new entry can be defined by filling the first empty row. Some fields like the *Slot* and the *DUT* are normal text fields, whereas *HW Target* and *Target FSM* are implemented as *ComboBox*, since their value is constrained to an existing value.

It is worth noticing that the OvenPlan page uses the properties supported by JavaFX, and explained in Section 2.8, to make the user interface more responsive and consistent. The list of OvenPlan entries, indeed, is stored in the application *Database* as an *ObservableArrayList*, so that it can be observed for changes and bound to the table. The interconnection among these components is shown in Figure 5.11.

This means that every change performed in the table is directly propagated to the OvenPlan list, and vice versa. This strategy also reduces the implementation effort, since it is not necessary to define custom functions to properly update the data when something changes in the table.

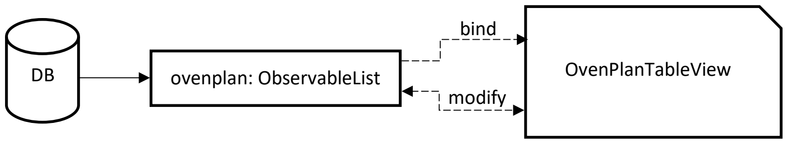
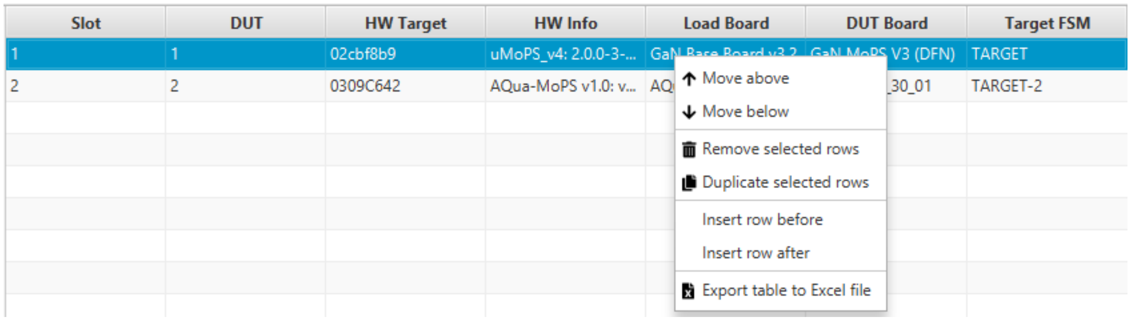


Figure 5.11. OvenPlan data binding

A context menu was also defined to interact with the selected rows, as well as export the table into an Excel file. An example of an OvenPlan page, showing also the context menu options, is represented in Figure 5.12.



Slot	DUT	HW Target	HW Info	Load Board	DUT Board	Target FSM
1	1	02cbf8b9	uMoPS_v4: 2.0.0-3-...	GaN Base Board v3.2	GaN MoPS V3 (DFN)	TARGET
2	2	0309C642	AQua-MoPS v1.0: v...	AQ	_30_01	TARGET-2

Figure 5.12. OvenPlan page

Finally, the OvenPlan structure is automatically using the existing checker described in Section 3.6.3: each time that the data changes, a new *Thread* executes the checker in background, and the found errors are listed in the general error list, accessible from the Bottom panel.

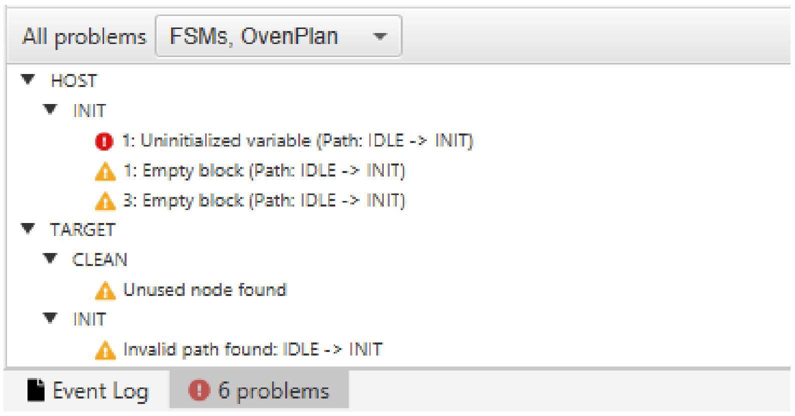
5.3.3 Bottom Panel

The Bottom panel guarantees access to important functionalities on the same application page, avoiding altering the content of the Main panel. It is implemented as a horizontal container populated with a specific component, based on the selected item in the status bar, located below the Bottom panel. The status bar contains a button for each feature that, when it is clicked, opens the Bottom panel, and the corresponding component is rendered. If the active button is clicked again, the

Bottom panel is collapsed. This panel is also implemented in a modular way, as new visual components can be added and attached to a new item in the status bar. Currently, the view contains two elements:

Event Log It shows the list of logs generated by the *Logger* (see Section 5.5.2). It can be useful to check the application status or to read the details of an exception when it occurs. Each log entry shows the time it was generated, and its description.

All Problems It shows the errors of all graphs. This component aims to provide an overview of all existing problems, giving immediate feedback to the user about the test-plan correctness, without the need to open and inspect every single FSM looking for possible errors in the nodes. It is implemented as a *TreeView*, where each root element represents a graph and shows all its problems, grouped by nodes. Furthermore, the icon of the corresponding button in the status bar, represented as a warning circle, turns red in case there are errors, and the total number of problems is indicated. Clicking on an item, the source of the problem is opened, and the corresponding error is highlighted. This is also the area where the OvenPlan errors are reported: in that case, an *OvenPlan* root item is created and its errors are listed inside. Finally, a scope area was implemented at the top of the component, to allow choosing which kinds of errors to show (FSMs only, OvenPlan only, or both). An example of the described component is shown in Figure 5.13.


All problems FSMs, OvenPlan

- HOST
  - INIT
    - 1: Uninitialized variable (Path: IDLE -> INIT)
    - 1: Empty block (Path: IDLE -> INIT)
    - 3: Empty block (Path: IDLE -> INIT)
- TARGET
  - CLEAN
    - Unused node found
  - INIT
    - Invalid path found: IDLE -> INIT

Event Log | 6 problems

Figure 5.13. All Problems panel

5.3.4 Additional features

Some additional graphical components were defined to make the system more user-friendly. Among these are:

Status Bar It is located at the bottom of the window and contains, in addition to the Bottom panel items, a progress bar, and a status label. The progress bar notifies the download status of the application data, whereas the label shows the last logged event message.

Notifications A notification system, provided by the *ControlsFX* library, was implemented. Notifications, shown in the bottom center of the application window, are used to notify errors, warnings, or success events. For example, when a drawing action violates a constraint, a notification appears to indicate the type of error, without blocking the user interaction with the application. They are also used to notify the status of a save or an export operation, to give immediate feedback to the user about the outcome.

5.4 Hierarchical FSM

The core topic of the graphical tool relates to the hierarchical design of an FSM, following the proposed solution explained in Section 4.2.3. It is represented as a block with input and output ports, which correspond to the transitions entering and leaving the sub-FSM. Every time that an edge is drawn from a state to the sub-FSM block, an input port is created in the child graph, which must be connected to one of its states. In this way, a state can be connected to another one in a deeper layer. The child graph also contains an *exit point* to link a node to another one in a parent graph. When a state is connected to the *exit point*, indeed, an output port appears in the sub-FSM block of the parent graph, that must be linked to another node.

While designing hierarchical FSMs, the standard rule of the MoPS FSM must be preserved. This means that each node, whatever its level of depth, must be reachable by the start state of the root graph, and must be connected to a final state of the latter. Furthermore, the name of each state must be unique among the entire hierarchical structure. To properly support the hierarchical design, the graph library was extended adding the following graphical components:

RectangleCell Represents the block containing a child graph. Like the standard nodes, it has a unique name, and shows an error icon in case of problems inside it (or in any of its nested layers).

AttachCell It is a small circle which represents an input port. It contains the reference to the corresponding edge in the parent graph, which has triggered the creation of the port itself. It also has a name which is directly bound to the name of the transition it is related to.

OutputPortEdge It is a point created in the sub-FSM block each time that a state of its graph is connected to the *exit point*. It stores the reference of the inner edge it is related to, and its name is bound to it. Therefore, if the name of the exiting transition is changed, the modification is reflected directly to the output port name in the parent graph. Moreover, by pressing and dragging the point, it is possible to connect it to the target node.

5.4.1 Create a sub-FSM

A sub-FSM block cannot be added from scratch, since the idea is to start grouping together a set of existing nodes. Therefore, in order to create a sub-FSM, it is necessary to select at least one state, open the context menu from one of them, and press the *Convert to sub-FSM* option. At this point, a function that tries to move the selected nodes into a nested layer is triggered. First, a dialog appears to enter the name of the sub-FSM block, which must be unique. Therefore, a function that recursively traverses all layers, looking for the entered name, is executed. Then it creates a *RectangleCell* instance, and attaches a *Graph* object to it. The latter will contain the structure (nodes and edges) of the child graph. Furthermore, the name of the sub-graph is bound to the name of the *RectangleCell*, so that it is automatically changed when the sub-FSM block is renamed.

The next step concerns the population of the sub-graph. First of all, the selected nodes are moved to the child graph, excluding the start and final states from the selection, as they cannot be part of an inner graph. Then, the edges whose source and target states belong to the added nodes are computed and moved to the child graph, to preserve the existing connections among the nodes. After that, the function computes the entering edges, that are the transitions having only the target node included in the group of nodes now moved into the sub-FSM. This means that these edges start from a node in the parent graph and enter into the sub-FSM block. For each of these edges, its destination node becomes the sub-FSM block, and an *AttachCell*, representing an input port, is created into the child graph, binding its name to the edge itself. Then the input port is connected to the node in the sub-FSM which was the original destination of the entering edge. The same operation is applied for the exiting edges, which are the ones having as target a node belonging to the parent graph. In this case, a transition is added from the source node in the child graph to the *exit point*, and this operation triggers the creation of the corresponding output port in the parent graph, which is then connected to the original target node of the exiting edge.

A sub-FSM can be opened as a new *Tab* in the Graph Page by double-clicking its block. Then, the user can freely organize the graph panels to better work with them. For example, it can be convenient to open the parent on the left side of the area, and one of its children on the right side, to efficiently operate with them. Figure 5.14 shows how a sub-FSM is graphically represented, both from the external point of view, in the parent graph, and internally.

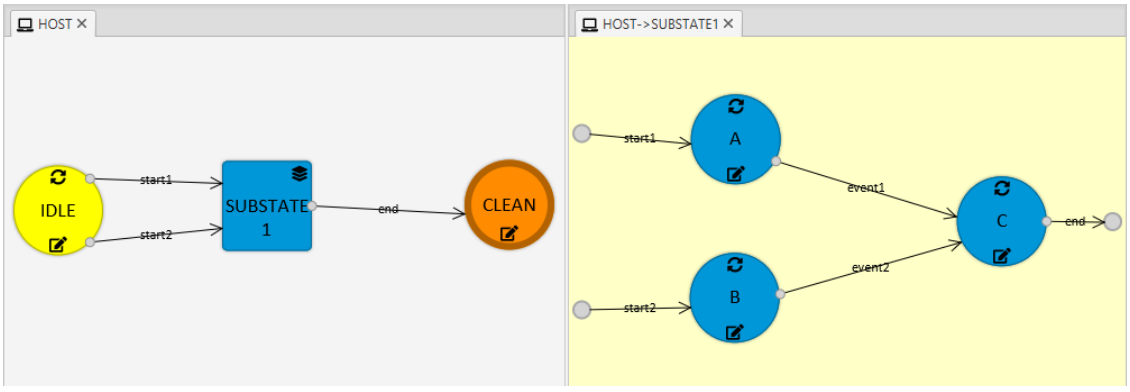


Figure 5.14. Example of a hierarchical FSM

To add an input port it is necessary to draw a transition from a state to the sub-FSM block, in the parent graph. When a node is connected to the *exit point*, instead, an output port is added to the *RectangleCell*. Each port can be connected to a node by pressing and dragging its point. Furthermore, it is possible to delete an arrow from a port to a node by double-clicking on the port's point. The input ports are always placed on the left side of the graph area and equally spaced, whereas the *exit point* is positioned on the right side. These points are fixed on the page, so that the user never loses them, even if the graph area is panned/zoomed. It is worth noticing that every time that an operation involves the sub-FSM, all the affected graphical items are consistently updated. For example, if a transition is added to the sub-FSM node, the *AttachCell* representing the input port immediately appears in the child graph. In the same way, when an edge is removed, its corresponding input/output port is automatically deleted.

The explained strategy used to create hierarchical FSMs, based on nested *Graph* objects and parent/child relationship of edges, imposes no limits either on the number of sub-FSMs per graph or on the number of nested layers. Thus, the group of nodes to be moved into a sub-graph can include a sub-FSM node, which will be then moved into a deeper layer. Furthermore, to guarantee a correct output, the following rules must be followed:

- A sub-FSM can contain neither the IDLE nor final states.
- Multiple transitions from the same external node to a sub-FSM node are allowed.

- Multiple transitions from an external node to the same internal node are not allowed, and vice versa.
- It is not possible to draw a transition having a sub-FSM node as source, as only the output ports can be used to connect it.

Finally, a specific function was implemented to safely remove a sub-FSM. It deletes all nodes in the sub-graph, and recursively calls the same function for all sub-FSM blocks among its nodes. In the end, all states and transitions from the level of the deleted sub-FSM and below are removed.

5.4.2 Flat a sub-FSM

Since the creation of a sub-FSM is represented as a grouping operation, it can be useful to have the opposite function, to ungroup a sub-FSM block. This function is accessible from the context menu of a sub-FSM node, which provides two strategies for flatting a graph: a shallow and a recursive one. The shallow flat takes all nodes of the child graph, add moves them to the parent graph, including the connections among them. Then, for each input port, its source node in the parent graph is retrieved, and an edge is created from that node to the target one, which has been moved to the parent. Finally, a similar operation is applied for the output ports, and the sub-FSM node is removed, as it became empty. The recursive flat acts like the shallow one, but it recursively calls the flat function for each sub-FSM node, in order to ungroup the entire hierarchy from the selected node.

5.5 Core Modules

The entire system works thanks to the interconnection of a set of modules, which can be either graphical or internal components. In the following, the main modules are explained, focusing on their goal and implementation.

5.5.1 Database

The *Database* class is the heart of the entire system, as it stores all relevant application properties and elements. It is defined as a Singleton class, in order to be accessed anywhere in the project. Among the other things, it contains the OvenPlan data, the list of graphs, the selected SAM version, and the list of Undo managers. It also defines the methods to create, rename, duplicate, and delete FSMs, as well as the functions to save, export and import a test-plan. The *initNewGraph*, for example, receives the FSM type and name and creates an instance of a *Graph*. Then, it is populated with the default graph structure, as explained in Figure 5.6, and saved into the Graph list. At the application startup, the *Database* is initialized and two FSMs are created, one of type *Host* and another of type *Target*.

This module is also in charge of properly updating the SAM version and the OvenPlan, to consistently update all the affected modules. Therefore, when the SAM version is changed, it reloads the autocomplete values of the Host FSM and, recursively, of all its sub-graphs. A similar operation is executed when the OvenPlan changes, updating the supported hardware APIs of the Targets.

5.5.2 Logging

A single Logger is defined and used by the application to track all important events, as well as runtime exceptions. Its goal is to let the user understand what happens, especially in case of problems. Those events are registered with a different severity level, based on their relevance. The application is set by default to the *INFO* level, which means that less serious events, like debug messages, are not handled. Anyhow, the Logger level can be changed on the Settings page.

A handler is attached to the Logger to handle all captured events. For each of them, it triggers a function which adds the log into the Event Log in the format of *Date & Time - Message*, as explained in Section 5.3.3.

5.5.3 Application settings

The application properties are handled by several configuration classes, each of which deals with a specific part of the system. The configuration is split into three modules:

Network Configuration To configure the application data downloaded from the remote repositories.

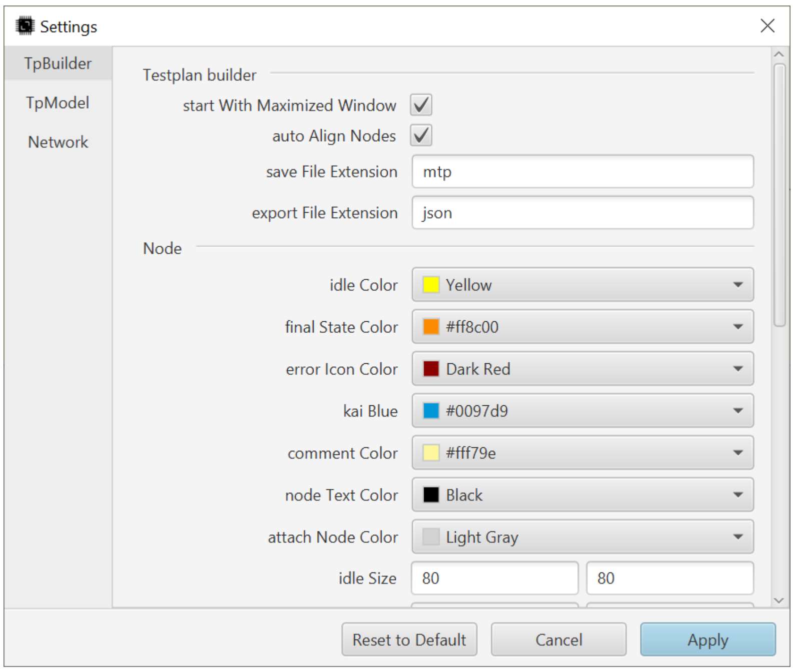
TPBuilder Configuration To customize the default graphical properties of the FSMs, like colors and sizes of nodes/edges.

TPModel Configuration To configure specific parameters of the test-plan model, like the default name of the *else* event, or the hardware event prefix.

All these settings are persistently stored into a JSON file, and mapped into the corresponding classes during the application startup. If a property does not exist in the configuration file, its default value is used. Furthermore, the output file can be altered manually, as the JSON syntax is human-readable.

Settings page

The settings can be directly customized from the visual application through a Settings dialog, accessible from the menu File › Settings, or using the shortcut Ctrl + Alt + S. The window, shown in Figure 5.15, is divided into two parts: the left bar used to select the module to configure, and the right panel to edit the related properties.



Testplan builder

start With Maximized Window	✓
auto Align Nodes	✓
save File Extension	mtp
export File Extension	json

Node

idle Color	🟨 Yellow	
final State Color	🟧 #ff8c00	
error Icon Color	🟥 Dark Red	
kai Blue	🟦 #0097d9	
comment Color	🟨 #fff79e	
node Text Color	⬛ Black	
attach Node Color	⬜ Light Gray	
idle Size	80	80

Figure 5.15. Settings window

The settings page uses a flexible approach to render and edit the application properties, relying on Java Reflection (see Section 2.3). First, the instances of the configuration classes are cloned, to avoid affecting the original ones. Then, for each class, the list of attributes is retrieved, and a specific graphical field is created, based on the attribute's data type. This field is populated with the current value of the attribute. After that, a listener is attached to each field, and the corresponding setter method is called in the cloned configuration class to update the parameter value. The setter function is retrieved from the configuration class by its name using Reflection, as functions are named prepending the word *set* to the name of the attribute. Thanks to the usage of cloned classes it is possible to discard the changes by clicking the *Cancel* button. In that case, the real classes are not manipulated. If instead, the *Apply* button is pressed, the cloned classes are assigned to the real ones, and the application is restarted to guarantee that the new settings are properly loaded. The code snippet shown in Listing 5.2 explains how the setter method associated with a field is retrieved.

Listing 5.2: Settings through Reflection

```java
//object is the instance of a configuration class
Field[] fields = object.getClass().getDeclaredFields();

for (int i = 0; i < fields.length - 1; i++) {
   Field field = object.getClass().getDeclaredField(fields[i].getName());
   //...
   //build the setter name
   String methodName = "set"
        + field.getName().substring(0, 1).toUpperCase()
        + field.getName().substring(1);
   //get the method with the computed name
   Method m = object.getClass().getMethod(methodName, String.class);
   //invoke the object passing the new value as argument
   m.invoke(object, "newValue");
}
```

5.5.4 Application data

The application data is retrieved from a web server and stored in a related JSON file. During the program startup, several threads are executed in parallel to download and cache the application APIs and the list of EDS of the devices. Each thread retrieves the cached versions of a specific module (i.e. the SAM documentation) from the corresponding JSON file, and stores them into a *HashMap*, where the *key* is the version and the *value* is the related documentation. Then, it contacts a remote uniform resource locator (URL) to get the actual versions, downloading the ones that do not exist locally and removing the local versions that are no longer supported online. This works by checking, for each remote version, if it exists in the *HashMap*, and vice versa. For each API version to be downloaded, its data is retrieved and saved into a proper object using the *Jsoup* HTML parser. Finally, the cached file is replaced with the new data, serializing it in JSON.

During the data caching process, a progress bar is shown in the Status bar to notify the user that the application is still loading the required data. When all threads finish their execution, the progress bar is hidden. Each thread is encapsulated into a specific class implemented using the Singleton pattern, which also represents the access point to the cached documentation.

Loading of local documentation

To allow the testers to use APIs not yet published on the remote server, a mechanism to load the documentation from a local source was implemented. It can be enabled from the Settings page, ticking the option *load local documentation*, in the *Network* section. Then, the user has to specify the paths of the local documentation to be loaded. Three directories are possible: one for the SAM versions, another for the Hardware APIs, and the last one for the EDSs, but it is not necessary to insert all of them. The directory three of each path emulates the corresponding remote one. For example, the root folder of the local SAM API must contain the list of versions wrapped in a folder named with the API version. Regarding the Hardware APIs, instead, the versions must be grouped by their family type and encapsulated in a folder with the name of the family, as shown in the following:

```
/
├── CORE
│   ├── 6.0.0
│   │   ├── index.html
│   │   └── modules
│   └── 7.0.0
│       └── ...
└── SoM
    └── 1.0.0
        ├── index.html
        └── modules
```

When the application is started, each thread checks if the *load local documentation* option is enabled and the path related to the type of documentation it is in charge of is specified. If not, it proceeds by downloading the data from the server; otherwise, it explores all folders in the specified local directory, and uses the same function of the remote data caching, connecting the HTML parser to the local files. Using this strategy, the data is loaded but not cached, as it is meant for testing purposes only.

5.5.5 Code Autocompletion

The code autocompletion is one of the main components of a visual code editor, as it improves the user experience and simplifies the code writing process. In the *Test-plan Builder*, it is particularly useful to suggest the MoPS-CORE and SAM APIs, offering also the documentation of each module/function. As explained in Section 4.2.1, an autocomplete component was developed from scratch. It is composed of two panels, wrapped together into a container: the left one, implemented as a *ListView*, which contains the list of suggestions, and the right one, which uses a *WebView* to show the documentation of the selected entry. Among the other things, the most important elements to be suggested are API modules and functions, as they have a proper implementation and description. Therefore, a specific class hierarchy was defined to represent an autocomplete entity:

CodeCompletion Represents a general entity, like a Lua keyword, or a shortcut to insert a block of code, like a condition or a loop. The *input* field indicates the name that should match the written word in the Code area to suggest the element. The *replacement* string, instead, is the block of code that must be inserted if the suggestion is confirmed. If empty, the confirmation of the suggestion will just complete the written word using the *input* string.

ModuleCompletion It is a child of the *CodeCompletion* class and represents an API module/class. Its fields contain the proper data to show the corresponding documentation in the right panel.

FunctionCompletion Defined as a child of *CodeCompletion*, represents the specialized class for an API function. Therefore, its fields define the set of arguments, the return type, etc.

The *ListView* of the left panel accepts instances of the *CodeCompletion* class.

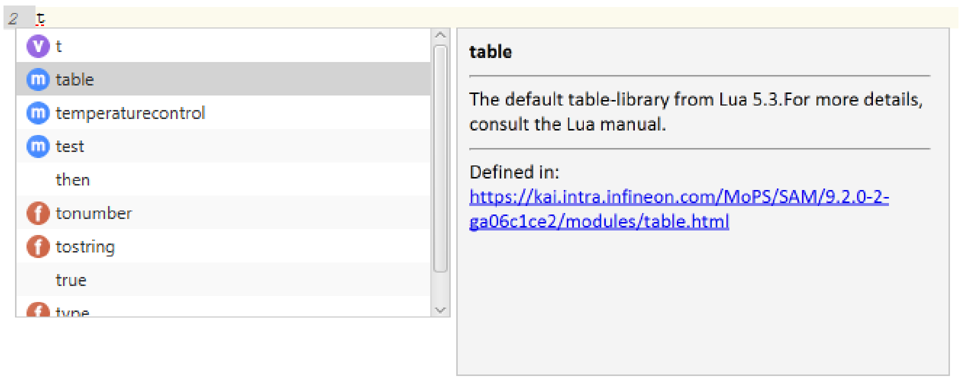
When a Graph panel is created for a specific FSM, an autocomplete component is attached to it. Then, Lua keywords and APIs are added to the suggestions list. In particular, if the FSM is of type Host, the SAM modules are retrieved from the cached data and added to the list, mapping them into *ModuleCompletion* instances. After that, all methods related to each module are explored and inserted as *FunctionCompletion* objects. If instead, the FSM is of type Target, all its EDSs (related to the Hardware targets linked to the FSM) are retrieved, and the intersection among the modules supported by each EDS is computed, in order to obtain a list of common modules working in all linked boards. At this point, the computed modules and functions are loaded as with the Host graph. Every time that a *ModuleCompletion* or *FunctionCompletion* instance is created, its fields are properly filled and formatted in HTML, so that the documentation can be directly rendered in the description panel. Since the suggested APIs depend on the selected SAM version and the OvenPlan, the suggestions list is reloaded every time that this information changes.

In addition to the API elements, a dynamic suggestion of user-defined variables, tables, functions, etc., was implemented. This feature represents a big improvement of the code autocompletion with respect to the previous system. To achieve this goal, only the nodes that can reach the selected one should be considered, otherwise the risk is to suggest elements declared in subsequent nodes. Furthermore, to guarantee a correct suggestion, it is necessary to take into account the code scope where the user is writing. Therefore, a recursive function was implemented to get all nodes that can reach the current one, starting from the latter and visiting the source nodes of the transitions directed to it. The operation is repeated for each encountered state, and a list of visited edges is kept to avoid loops. The code of the previous nodes is then merged and analyzed to find all global declarations, since local scopes are not visible by others, whereas the code of the selected node is considered from the beginning to the current position, where the user is currently writing, since the subsequent declarations cannot be reached. The local code is then analyzed separately to take only the data visible by the current scope.

LuaJ parser is used to analyze the code by implementing a custom *Visitor* named *VariableVisitor*. The *Visitor* visits variable assignments and function declarations, storing them into a list which is then returned to the autocomplete component. After that, each entry is mapped into a *CodeCompletion* and added to the suggestion list. In order to consider the current scope, it is important to highlight how the declaration works in Lua. In particular, it is necessary to prepend the *local* keyword to declare something locally, otherwise the declaration is intended as global, and the element is reachable even outside its scope.

Regarding the code of the previous nodes, only the not-local declarations are considered. For analyzing the local code, instead, the following strategy was implemented to consider the scope. First, the *Visitor* visits each block, which corresponds to a scope, and adds it to a list of scopes. Then, global and local declarations are caught and stored in the list of results. For each scope object, LuaJ stores the list of variables declared inside it, and a pointer to its parent scope. Thus, given a variable and a scope, it is possible to know if it is visible by that scope checking if the variable is contained into the scope's variable list or in one of its ancestors. At this point, it is just necessary to find the block being modified by the user and filter the list of declarations keeping only the ones reachable by it. To accomplish this task, a trick is used to get the index of the current scope, based on the fact that the parser generates a *ParseException* if the code is not syntactically correct. Since the local code is cut, its scopes are not closed properly, and an error is thrown every time it is parsed. Thus, when the exception is caught, an *end* keyword is added to close a scope, and a *scopeIndex* variable is incremented. This operation is repeated until there are no more opened blocks. At this point, the code is correctly parsed, and the current scope in the *VariableVisitor*'s scope list corresponds to the one at position *scopeIndex*. Once the scope is found, all elements visible inside it are added to the list of suggestions.

When the text in the Code area changes, an event is triggered, and the last inserted word is computed, which is used to filter all suggestions which start with that string. In case there are some results, the autocomplete panel is automatically shown and placed below the row where the user is writing to. Furthermore, if a module name plus a *dot* is written, its methods are shown. Additionally, in case of a function suggestion is confirmed, the required parameters are also inserted, and the first one is selected. To let the user understand the meaning, each row in the list can contain a symbol to indicate its type (i.e. *m* indicates a module, *v* a variable, etc.). The panel can also be manually opened through the Ctrl + Space shortcut. Moreover, its location is automatically re-adapted to correctly fit it in the window. Figure 5.16 shows an example of an autocomplete panel.



table

The default table-library from Lua 5.3.For more details, consult the Lua manual.

Defined in:
https://kai.intra.infineon.com/MoPS/SAM/9.2.0-2-ga06c1ce2/modules/table.html

Figure 5.16. Code Autocompletion example

The autocomplete component can also be filled with custom values to use it in different contexts. For example, every time that a *setEvent* function is inserted, the autocomplete suggests all possible event names. Finally, when the mouse pointer is over an API function or an error, the description panel appears as a popup to show the description of the pointed item.

5.5.6 Search function

Searching a portion of code among the whole test-plan is an important feature to be implemented, as finding a specific string may be difficult and time-consuming, especially in the case of large tests. Therefore, a search and replace system was developed to accomplish this goal. It can be opened through the Menu bar or using the shortcut Ctrl + F, and it is graphically represented as a dialog which does not block the main window, so that the user can continue using the application while searching the data through the dialog. The component contains two text areas, one to define to text to be searched, and the other to additionally specify the replacement text. Furthermore, the following searching options were specified to refine the search:

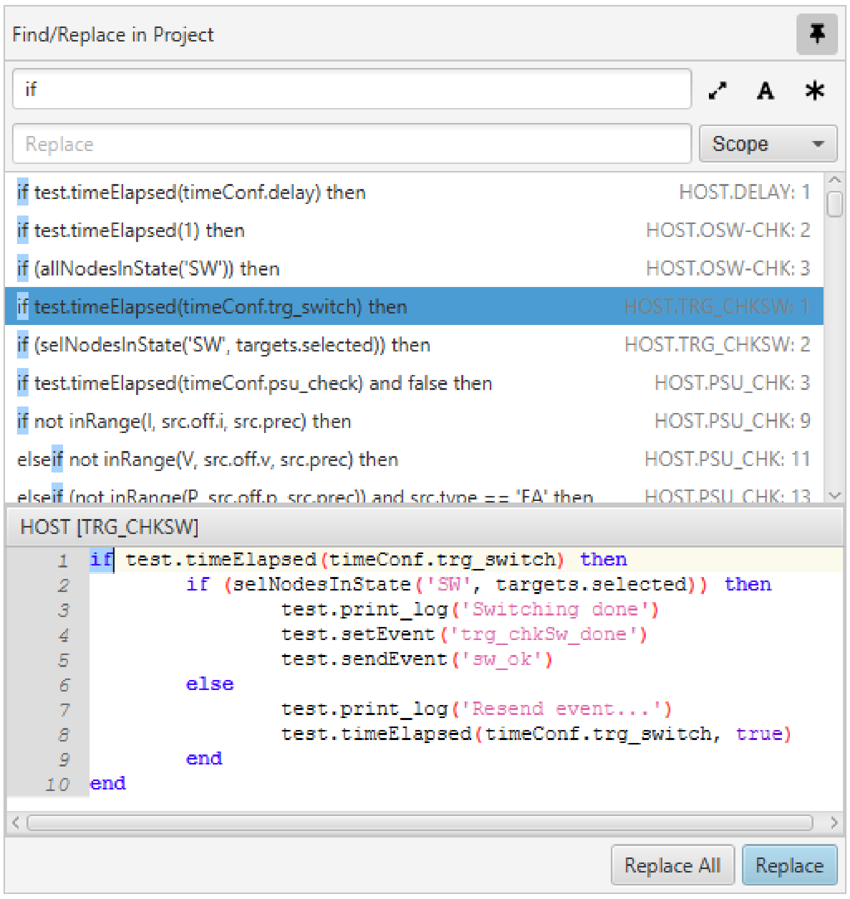
Match case Used to make the search case sensitive.

Whole word To match whole words only.

Regular expression To find all matches of a custom-defined regular expression.

Scope To define the set of FSMs to search on.

The search is automatically executed, each time that the text in the search area changes, or the search options are modified. In that case, the right pattern, based on the search parameters, is computed and searched among all nodes of the specified graphs. The occurrences are searched using the Java *Matcher* class, which finds all matches to a given regular expression. Therefore, the search pattern is always specified as a regular expression, and it is properly defined to match the exact text in case the *regex* option is not selected. All results are shown into a *ListView*, where each entry reports the line of code related to the found match, and the information about its location (in the format *<FSM>.<Node>: <lineNumber>*). Once an entry is selected, its code is directly shown in a Code area in the search dialog itself, to offer the possibility to immediately modify the node's code. The content of this text area is bound to the Code area of the Graph panel of the FSM the selected occurrence belongs to. This increases the consistency of the application, as the code modified from the local area in the search dialog is directly propagated to the Code area of the corresponding Graph panel, if visible. Furthermore, when an entry is double-clicked, it jumps to the corresponding node in the Main panel, and the affected code is selected in the Code area. The last important feature of the search component is the replacement function: the *Replace* and *Replace All* buttons are placed at the bottom of the dialog to affect respectively the selected entry, or all occurrences. Figure 5.17 shows an example of a Search dialog containing some results.



if

if test.timeElapsed(timeConf.delay) then	HOST.DELAY: 1
if test.timeElapsed(1) then	HOST.OSW-CHK: 2
if (allNodesInState('SW')) then	HOST.OSW-CHK: 3
if test.timeElapsed(timeConf.trg_switch) then	HOST.TRG_CHKSW: 1
if (selNodesInState('SW', targets.selected)) then	HOST.TRG_CHKSW: 2
if test.timeElapsed(timeConf.psu_check) and false then	HOST.PSU_CHK: 3
if not inRange(I, src.off.i, src.prec) then	HOST.PSU_CHK: 9
elseif not inRange(V, src.off.v, src.prec) then	HOST.PSU_CHK: 11
elseif (not inRange(P, src.off.p, src.prec)) and src.type == 'FA' then	HOST.PSU_CHK: 13

HOST [TRG_CHKSW]

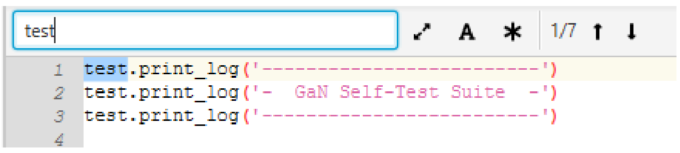
```
if test.timeElapsed(timeConf.trg_switch) then
        if (selNodesInState('SW', targets.selected)) then
                test.print_log('Switching done')
                test.setEvent('trg_chkSw_done')
                test.sendEvent('sw_ok')
        else
                test.print_log('Resend event...')
                test.timeElapsed(timeConf.trg_switch, true)
        end
end
```

Figure 5.17. Search dialog example

It is worth noticing that the dialog is automatically closed when the user clicks outside it, or double-click on an entry. To prevent the default behavior, it is possible to pin the window, keeping it always on top of the main application, toggling the *pin* button in the top-right corner of the dialog.

Local search area

The search dialog is intended to search something in the entire test-plan, but it is not suitable for performing a local search on the current Code area. For this reason, a local search component was implemented in the Graph panel. It can be opened using the shortcut Ctrl + F inside a Code area, and it is rendered on top of it. The search options are the same as the search dialog, but the occurrences are searched only in the current code. Each found occurrence is highlighted in the text, and it is possible to navigate through the other results using the arrow buttons defined in the component, or pressing *enter*. Figure 5.18 shows an example of this component.



```
test.print_log('-------------------------')
test.print_log('-  GaN Self-Test Suite  -')
test.print_log('-------------------------')
```

Figure 5.18. Local search area example

5.5.7 Undo Manager

The history of changes is a really useful functionality that every application should have. Making a mistake during the FSM definition is something common, and reverting the changes would make the life of a user easier. This goal was achieved by implementing a general undo/redo system, applicable in different contexts. As described in Section 4.2.2, a local history of changes must be kept for each FSM and for the OvenPlan, to avoid affecting the whole system during a undo or redo. First, an *UndoManager* class was implemented, which defines the logic to navigate through the history of changes. It keeps a list of changes, and an index that indicates the last applied change. In this way, every time that an *undo* is performed, the index is decremented and the component is reverted to the change in that position. Likewise, the index is incremented if a *redo* is requested.

An *Operation* is an object which describes a change, storing the state of the component before and after the modification. Since the *UndoManager* should work with every component regardless of its implementation and data type, a *UndoOperation* interface representing an *Operation* was defined, so that the manager can hold in the history list every object implementing that interface. Then, an abstract class defining the common properties of an *Operation* was defined. It stores the state of the object before and after the edit, a description of the operation and the creation date. The only function which is component-dependent is how the undo or redo is executed. Furthermore, the functions to set the component's state can be overridden to customize the way the state is saved. Thus, a specialized class was defined for a Graph and the Ovenplan, overriding the *execute* method of the interface, in order to define the specific strategy to change the state of a component to a previous or subsequent one. Both classes also reimplement the *setUndo* and *setRedo* methods to serialize the state of the object in JSON, so that it can be easily rebuilt. Figure 5.19 provides an overview of the whole system. This modular approach guarantees that the *UndoManager* can be used with any kind of object, as the only required action is to create a class for the desired component which inherits the *Operation* class and defines the execution logic of the undo/redo operation.

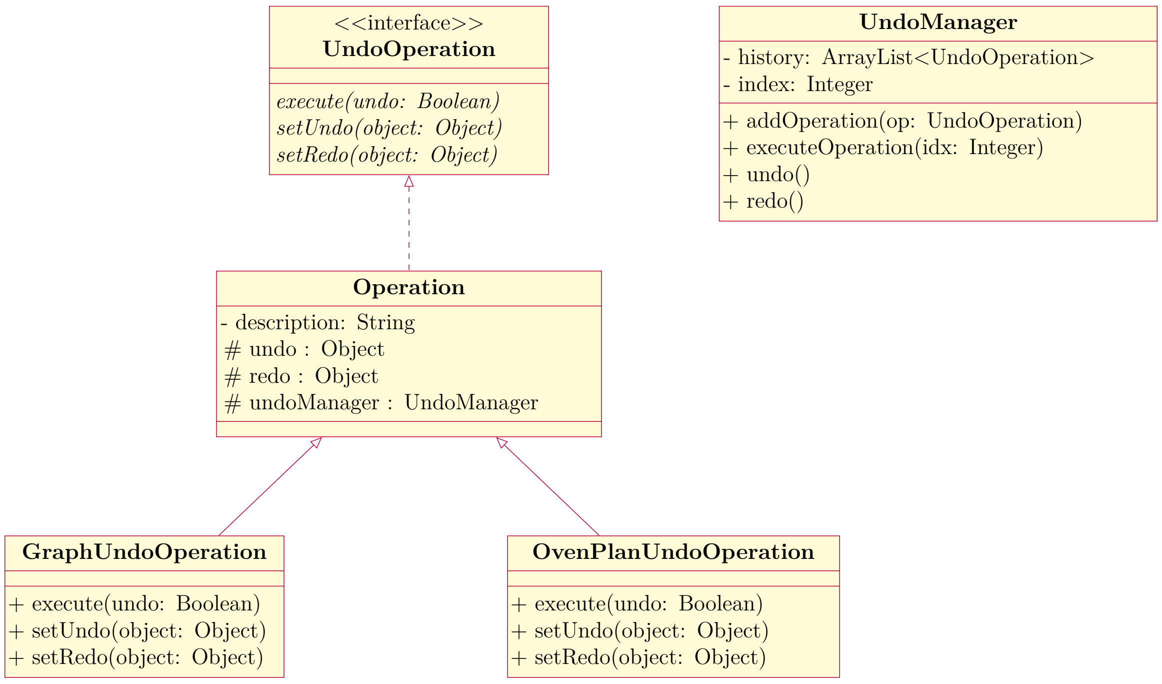
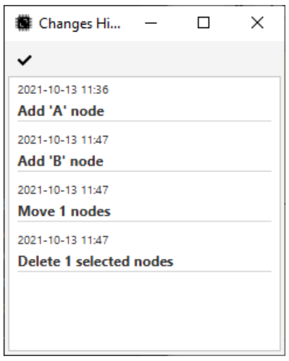


Figure 5.19. UML class diagram - Undo Manager

When an FSM is created, an *UndoManager* is attached to it. Then, every time that the graph is modified, a *GraphUndoOperation* is created and added into the manager's history. In particular, the instance of the transaction is created before editing the graph, and the *setUndo* method of the operation is called, passing the graph object as an argument, to store the snapshot of the graph before the modification. After the FSM has been edited, the *setRedo* function is called to serialize the new version of the graph. Despite these operations might seem complex, they do not affect the performance and responsiveness of the system, as the object serialization in JSON is very fast. Furthermore, the object is serialized twice only for the first added operation, as the *redo* of the subsequent ones is equal to the *undo* of the previous operation, therefore it is copied from it. In the case of a graph containing sub-FSMs, the *UndoManager* is created only in the root, since the serialization of the root graph hierarchically involves all children. A similar behavior was implemented for the OvenPlan. In that case, when the table is modified, a *OvenPlanUndoOperation* is inserted into its *UndoManager*, after storing the serialization of the entries before and after the edit.

An undo/redo action can be triggered through the *Edit* menu, or using the shortcuts Ctrl+Z / Ctrl+Y. Then, the *undo* or *redo* function is called from the *UndoManager* of the focused item, according to the requested action. These functions update the index value, take the corresponding operation, and call the *execute* method on it, that will invoke, at runtime, the right concrete method. Furthermore, a dialog to show the history of the current *UndoManger*, offering also the possibility to restore a specific snapshot of the object, was created. It contains the list of operations with their description and creation date, which is directly bound to the history list of the manager. In this way, every time that something is modified, the corresponding operation immediately appears in that list. Moreover, by pressing the ✔ button, it is possible to restore the object to the snapshot of the selected item. It triggers the *executeOperation* function of the manager, which is capable of executing an arbitrary operation, corresponding to the received index. The history of changes, shown in Figure 5.20, can be accessed using the menu item Edit › Show History, or through the shortcut Ctrl+H. Finally, the managers are used to check if the project was modified, in order to ask the user to save the changes before closing it.



2021-10-13 11:36
Add 'A' node
2021-10-13 11:47
Add 'B' node
2021-10-13 11:47
Move 1 nodes
2021-10-13 11:47
Delete 1 selected nodes

Figure 5.20. History of changes example

5.5.8 Main actions

The application provides all useful functionalities to create, save, export, etc., a test-plan. The implementation of these actions, accessible from the File menu, is explained in the following sections.

New Project

This action, also accessible through the shortcut Ctrl+N, creates a blank project. If the current project was modified, the user is asked to save or discard the changes before proceeding. Whenever this function is triggered, the *Database* instance is reset, and two FSMs (a Host and a Target) are created. Then, the SAM version is set with the latest, and the documentation is subsequently loaded to the appropriate FSMs. Finally, the content of the Project Tree is reloaded, and the Host FSM is automatically selected. This function is also invoked at the program startup, to create an empty project.

Save

A project can be saved either from the menu File〉Save, or using the shortcut Ctrl+S. The path of the current project is stored into the *Database* object, which is not set in case the project has never been saved before. In that case, a dialog appears to select the directory where to save the file, and that path is stored in the application. If instead, a project file already exists, it is just replaced with the new one. Anyhow, it is always possible to save the project as a new filename through the *Save As* option. Saving a project is different from exporting it, as the file format is application dependent and it is meant to be reopened and modified, preserving the whole structure of the test-plan. The project is saved, after having been serialized in JSON, in a file with the extension *.mtp*. The output contains the OvenPlan entries, the SAM version, and the FSMs. For each FSM, the lists of edges and nodes are stored, together with their styling attributes, like the color, position, whether the node is a start or final state, etc. If a node is a sub-FSM, its graph is added as an attribute of the node itself, preserving the hierarchical structure of the FSM.

The save function uses Gson to serialize the most relevant *Database* data. Since a custom output should be created, to avoid storing useless information and to insert the required data to rebuild the hierarchical layers, some custom adapters (see Listing 2.1) were implemented to define how edges and nodes are serialized. The most important ones are:

EdgeAdapter defines the output format of an edge. It contains the name of the source and the target node, plus a piece of information that tells if these states are in the current FSM or the parent one. If the latter condition is true, it means that the edge comes from/is directed to the parent FSM. Such that information will be useful to recreate the graphical FSM from the opened project file.

RectangleCellAdapter specifies how a sub-FSM node should be saved. In particular, it creates an entry to store its graph, serialized using the same rules as the root one.

Export Project

This command, accessible from the menu File › Export project or with the shortcut Ctrl + E, generates a valid test-plan compliant to the standard MoPS format (see Section 3.2), and stores it in a JSON file. Since the generated test must be robust, all checks mentioned in Section 3.6 are executed before exporting the project. Thus, in addition to what is automatically checked, other verifications regarding the FSM structure and the code are performed. In case of errors, the output is not generated, and a notification appears to notify the failure.

The export procedure creates a *Testplan* object, whose structure emulates the MoPS test format (see Figure 3.2). Then, each field (i.e. SAM version, OvenPlan, list of FSMs) is populated with the corresponding content. The hierarchical representation of FSMs is not supported by the MoPS system, therefore, each graph is flattened to remove all nested layers. It means that all states, regardless of their depth level, are inserted into a single, plain, graph. This flatting procedure is also used by the live checker, and a better explanation of how it works can be found in Section 5.6.1.

Open

An existing project, saved in *.mtp* extension, can be reopened through the menu item File › Open, or with the shortcut Ctrl + O. When this action is triggered, a dialog to choose a file appears. Then, the content of the selected file is parsed into a JSON Object and manually analyzed to graphically build the FSM. First, the SAM version and the OvenPlan entries are retrieved and stored in appropriate data structures. Subsequently, the FSMs are rebuilt: it starts traversing the list of nodes, creating and adding the corresponding objects to the *Graph*, based on their type. After that, the edges are analyzed and added between their source and target states, which are retrieved from the added nodes by their name. If the encountered node is a sub-FSM, the procedure is repeated recursively to visit it. To correctly create the hierarchical structure, it is necessary to entirely build a layer of FSMs before proceeding with the subsequent layers, since the transitions entering or leaving a sub-FSM are linked to the corresponding ones in the parent graph, which must already exist. This is not guaranteed in general, as a sub-FSM node may appear before the list of edges of a graph and, therefore, it is visited before finishing to build its parent. To ensure the correct behavior, each time that a sub-FSM node is visited, the execution of the recursive call for visiting the sub-graph is delegated to a thread. The threads are then started only when the parent graph was completely built, ensuring an ordered reconstruction of the hierarchical layers.

Import Project

An exported test-plan can also be imported and reconstructed by the system through the menu item File » Import Testplan, or using the shortcut Ctrl + I. It can be useful when, for example, the project *.mtp* file is not available or its format is malformed, or to visually check the correctness of the test structure. When this action is triggered on a test-plan JSON file, its content is parsed into a JSON Object and visited. The graph creation process is simpler than the one used to open a project, as the file does not contain hierarchical FSMs. Therefore, for each visited state, a graphical node is created and styled using the default values defined in the application settings. When an edge is found, instead, the two instances of the source and target states are retrieved by their name and passed to the created *Edge* object. Since there is no graphical information about the nodes' placement, the autolayout feature (see section 5.3.2, FSM autolayout) is used to obtain a readable graph.

5.6 Test-plan check

The *Test-plan Checker* is an independent module able to verify the correctness of the entire test-plan, as explained in Section 3.6. It is used to automatically check the correctness of the code and the FSMs' structure, and during the export phase, where further checks are executed. To perform the verification, the test-plan must be converted into a specific format supported by the *Checker*, before being analyzed. During the live check, this conversion is performed by the *FSMChecker* class, which represents a bridge between the *Builder* and the *Checker* modules. An instance of this class is attached to each *Graph* object, and it is run every time that its FSM structure is modified, triggering the corresponding *update* method. First, this function rewrites the FSM into the proper format supported by the *Checker*. Then, each possible path from the IDLE node to itself is computed, the code of its nodes is merged, parsed and analyzed by the LuaJ visitors, looking for errors. For each error, a specific *token* object is created and stored into a list. The main information provided by the *token* is the error description, the affected node, and its location in the code, in case of a syntax error. Furthermore, if a path does not reach the final state, it is considered invalid, and an error is generated. The latter check is required to verify if the FSM meets the requirement of the MoPS system. Since the verification mechanism requires a lot of resources and, therefore, it is not immediate, the *update* function is executed in a secondary thread. At the end of the process, all nodes present in the list of errors are marked as wrong, and the error icon is shown on them. When a node is clicked, the list of errors is filtered, looking for the tokens related to the selected state, which are then listed in the *Problems* list, as shown in Figure 5.10.

The following sections describe how the *Checker* was adapted to support the verification of hierarchical FSMs, as well as the implemented optimizations and extensions to improve the *Test-plan Checker* module.

5.6.1 Hierarchical FSM verification

With the implementation of the hierarchical design of FSMs, a new verification mechanism, able to support multiple layers, was required. A solution could be to check the sub-graph independently, but it is not viable since its paths are not independent, but are connected to the start state of the root graph. Furthermore, the modification of the sub-FSM also affects the ancestors (and vice versa), which should be accordingly rechecked every time. To overcome these problems, and to avoid running too many threads (one per sub-FSM) at every modification, the *FSMChecker* was modified to merge all hierarchical layers into a single graph. Therefore, every time that the graph (or a sub-graph) is modified, the entire hierarchy is flattened, and the resulting FSM is checked. Thus, the *FSMChecker* receives always the root graph, and uses a recursive function to visit all sub-FSM nodes of each layer. For each encountered node that is connected to a state located in a different layer, it retraces the path to get the state to which it is connected to, and adds a direct link among them. For example, considering the hierarchical structure described in Figure 5.21 (above), where node A is connected to node B, which is linked to node *CLEAN*, the following actions are applied. When it visits node A, it traverses the nested layers to find the destination node, by following the *@else* transition in the subsequent layers. The traversal ends when node B is reached, which is added to the flat FSM and linked to node A. The process continues visiting recursively the sub-FMSs, applying the same mechanism to the encountered nodes. As soon as node B is visited, indeed, its *stop* transition is traversed among the different layers, until the destination node is found. The latter, which corresponds to the *CLEAN* node in the root graph, is added and linked to node B. The result of the flatting procedure is shown in Figure 5.21 (image below).

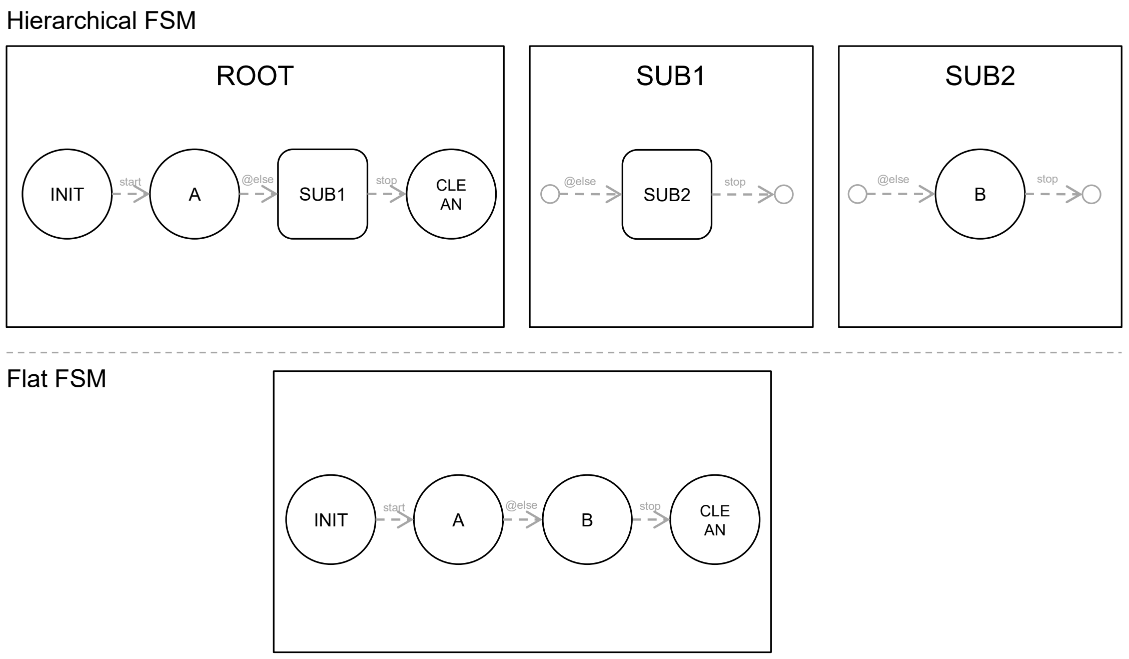


Figure 5.21. Hierarchical FSM (above) and its corresponding flat structure (below)

Furthermore, for each sub-FSM block, the function checks if all input/output ports are connected, throwing an error otherwise. These errors are visible either clicking on the sub-FSM node, or through the *All Problems* panel.

5.6.2 Checker optimization

The live verification of the test-plan is a heavy process, as it visits each possible path multiple times, looking for wrong paths, unused nodes, and code errors. This operation can require a relatively long time, especially in the case of large graphs, reducing the responsiveness and the usability of the system. After analyzing the checking process, it turned out that the slowest operation is the visit of the Lua code, and the execution of several *Visitors*, one after the other, to the same code, negatively affects the performances. Since each code visit is independent of the others, the mechanism was improved by executing all *Visitors* of a path in parallel, using a Thread pool system implemented through the Java *ExecutorService* (see Section 2.6). In particular, for each path, a thread pool with a number of threads equal to the number of *Visitors* is created. Then, each *Visitor* is submitted to the *ExecutorService*, and its *Future* object, immediately returned by the service to catch the return value of the executed thread, is stored in a list. After submitting all threads, the result of their computation, which corresponds to the list of found errors in the visited code, is obtained through the *get* method of the *Future* object. Finally, the thread pool is closed, and the list of tokens is returned. The source code of the explained operation is shown in Listing 5.3.

Listing 5.3: Parallel execution of the LuaJ Visitors using an ExecutorService

```java
//create thread pool
ExecutorService executorService = Executors
        .newFixedThreadPool(visitors.size());
//create list of futures
List<Future<List<GraphErrorToken>>> futures = new ArrayList<>();
try{
    for (MyVisitor v : visitors) {
        v.reset();
        //submit the visitor to the thread pool
        futures.add(executorService
            .submit(() -> LuaChecker.parse(v, mergedLua)));
    }
    //store the visitors' return value
    for (Future<List<GraphErrorToken>> f : futures){
        tokens.addAll(f.get());
    }
    executorService.shutdown();
}catch (InterruptedException | ExecutionException e){
    e.printStackTrace();
}
```

This strategy, which also improves the export phase, as it uses the same mechanism to check the Lua code, has drastically reduced the verification time, as explained in Section 6.2.

5.6.3 Checker extension

As mentioned in Section 4.2.2, the *Test-plan Checker* was extended to cover other possible errors, in order to guarantee an output that was as robust as possible. The previous version, for example, was not able to check if the triggered events existed or not. Therefore, it could recognize a test-plan as valid, even if it was not, violating the robustness property of the system. For this reason, a mechanism to validate the used events was implemented and exploited both in the live checking and during the export phase. In particular, the following situations must be checked:

- In case a node triggers an event belonging to its graph, through the *setEvent* method, it must verify whether the written event name exists or not.
- If instead, an event is triggered to one or more Target FSMs, using the *sendEvent* function, it is necessary to check if the event is present in all the specified graphs.

To accomplish this task, the Lua code is further analyzed by the *TriggeredEventsVisitor*. It visits all function calls of the parsed code, looking for a function used to trigger an event. If a *setEvent* is found, it extracts the first argument, which corresponds to the event name, and checks if it is included in the list of events of the FSM the node that triggered the event belongs to, generating an error otherwise. If the *Visitor* encounters a *sendEvent* function in which the second argument is not specified (and, therefore, the event is triggered in all Targets), it verifies that the event name exists in all Target FSMs, reporting the ones where it is not present. It is also possible to trigger an event only to the desired Targets, specifying a list of their physical addresses to the *sendEvent* function. In this case, the *Visitor* obtains the FSM linked to each address, through the OvenPlan, and checks if the event exists in that graph. Furthermore, it notifies an error if a physical address is not mapped to any FSMs.

Chapter 6

Evaluations

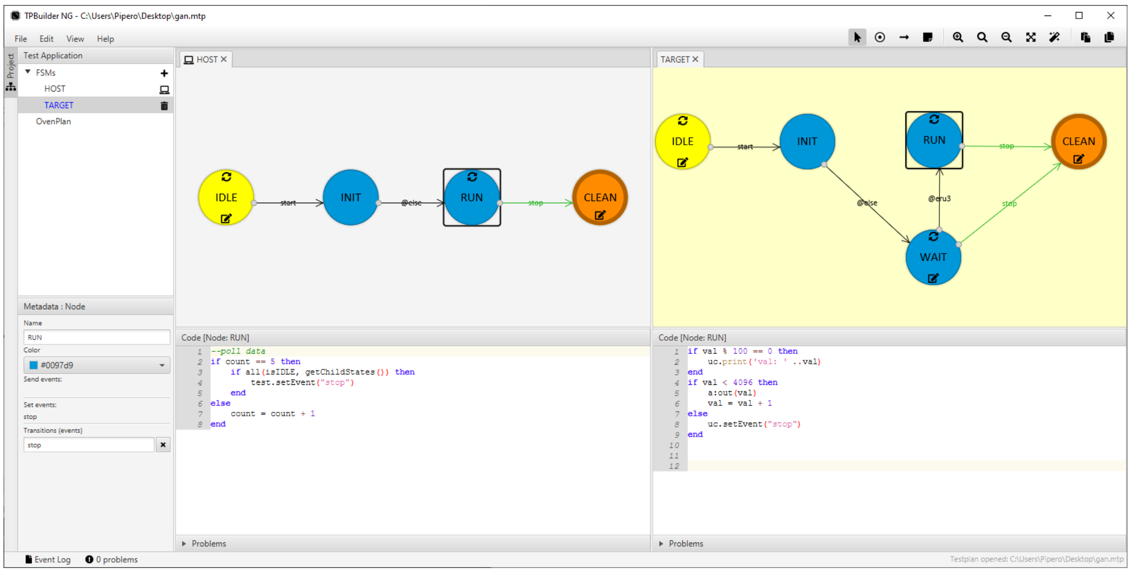
This chapter covers the evaluation of the implementation of the system. First, the results obtained by the new GUI of the *Test-plan Builder* are presented. The second part, instead, focuses on the improvements of the *Test-plan Checker*, comparing the performances with the previous system.

6.1 Test-plan Builder

The new user interface adopted for the system turned out to be more intuitive and user-friendly than the previous system. The implementation of a single-page application increased the simplicity and convenience of use, as the user can access all application components in the same window, without losing the focus from the main panel, where the test-plan is defined. Furthermore, the main interface, shown in Figure 6.1, is divided into specific areas, as described in Section 5.3.

On the vertical side, there is a panel used to access the project structure. From there, the user can easily manage the FSMs of the test-plan, and select the ones to be shown in the main area. On the bottom side, instead, the user can quickly check the application's log and errors, to get immediate feedback about the test-plan correctness. These side panels can be hidden to give more space to the main area, which can be comfortable in case a large graph needs to be represented.

The main area is the widest panel used to interact with the graphs and the OvenPlan. While in the previous system only two FSMs can be shown at the



Metadata : Node
Name: RUN
Color: #0097d9
Send events:
Set events: stop
Transitions (events): stop

Code [Node: RUN]

```
--poll data
if count == 5 then
    if all(isIDLE, getChildStates()) then
        test.setEvent("stop")
    end
else
    count = count + 1
end
```

Code [Node: RUN]

```
if val % 100 == 0 then
    uc.print('val: ' ..val)
end
if val < 4096 then
    a:out(val)
    val = val + 1
else
    uc.setEvent("stop")
end
```

Testplan opened: C:\Users\Pipero\Desktop\gan.mtp

Figure 6.1. Test-plan Builder

same time, the Host and one Target, the new *Test-plan Builder* is more modular, as all the desired graphs can be opened concurrently, increasing the usability of the application. For example, the user can close the Host FSM, and open more Targets, one after the other. Finally, it is worth noticing that, thanks to the usage of the *Binding* properties of *JavaFX* (see Section 2.8), the graphical components are always updated automatically, reflecting the data changes.

The interaction with the FSMs resulted much faster and smoother than before, especially in the case of pan and zoom operations. Drawing and editing the graph has also been simplified: using final states, for example, the user can immediately complete a path, without the need to connect the last state to the IDLE one. Moreover, it is possible to change the destination of an existing edge dragging it to another node, avoiding deleting it and creating a new one to accomplish the task. Finally, the sub-FSM definition is really intuitive, as the entire process is graphical and based on interconnections of input/output ports through mouse actions.

Another remarkable improvement regards Lua code writing. Thanks to a smarter autocompletion mechanism (see Section 5.5.5), the user automatically receives the possible suggestions based on the inserted text. Those suggestions include variables, functions, as well as API classes/modules. Moreover, only the elements reachable by the scope where the user is writing are suggested. In this way, the user can also receive immediate feedback about the variables/functions that can be used on a specific part of the code.

6.2 Test-plan Checker

The modifications of the *Checker* module have resulted in a more error-resistant test-plan, improving the overall performances of the validation process. The introduction of a check to validate the triggered events (see Section 5.6.3) increased the robustness of the system, which is one of the main requirements of the application. The previous system, indeed, was not able to verify the correctness of the triggered events. For example, the Host FSM could trigger an event which did not exist in one or more of the specified Targets, without recognizing the error and generating, therefore, a wrong test-plan. Another improvement regards the live checking: the new event check, together with the OvenPlan verification, is also executed during the automatic check. Thus, the user does not have to export the project to realize the presence of wrong events, or OvenPlan related errors.

The most relevant result obtained by the improvement of the *Test-plan Checker* is the significant increase of the validation speed. The general optimization of the checking function, the removal of repeated paths, and the parallelization of the source code's visit operations, have drastically reduced the time to check the entire test-plan, especially in the case of large graphs. Some tests were performed to measure the validation time during the export phase, the command line check, and the real-time check of a single FSM. Each measurement was repeated five times, and the average was considered. The tests were executed either with the new and the previous version of the system, considering both a small and a large graph. The detailed results are presented in Table 6.1.

Test-plan	Operation	Old time (ms)	New time (ms)	Improvement (x)
large	Export	4849.8	897.8	5.4
large	FSM check	2912.4	395.8	7.4
large	Cmd check	7437.4	1915.8	3.9
small	Export	234.4	105.4	2.2
small	FSM check	105.4	56	1.9
small	Cmd check	1153.4	457.6	2.5

Table 6.1. Comparison of the performances

It turned out that, in case of a live check of a large graph, the old system takes few seconds, whereas the new *Checker* needs less than half a second, becoming more than 7 times faster. Similar results were achieved for the export phase, and the command line check. Even the tests conducted in the small graph show significant improvements, as the new system is up to 2.5 times faster. Anyhow, the speed difference is lower in small test-plans due to the delay introduced by the creation of the threads.

Finally, the extension of the validation mechanism to support the hierarchical checking turned out to be fast and efficient. This check is executed only in the root graph, considering the hierarchical structure as a unique, flat, FSM. Thanks to this strategy, explained in Section 5.6.1, the original validation process was preserved, avoiding the introduction of an overhead due to multiple checking for every nested layer.

Chapter 7

Conclusion & Outlook

7.1 Conclusion

A desktop application has been developed during this master thesis project, which is related to the improvement of an existing system. It is made of two modules: the *Test-plan Builder*, a graphical tool that was recreated from scratch; the *Test-plan Checker*, which was modified to optimize the overall performances. All problems stated in Section 1.3 have been answered and solved during the implementation, following the solution proposed in Chapter 4.

The user interface was created using a more modern and efficient framework, which guarantees a better and smoother layout. The *Test-plan Builder* provides a user-friendly GUI, made of well-defined sections which increase the ease of use of the system. The creation of an FSM is really intuitive, as the user can draw it inserting states and connecting them through transitions. Additionally, the render quality of the graph is higher than in the previous system, and the overall design process is much smoother, even in the case of big graphs.

The goal of implementing the hierarchical design of FSMs was achieved successfully, keeping the process simple. A sub-FSM, indeed, is opened, rendered and populated as a normal graph, so that a user does not need much effort to familiarize with it. Furthermore, hierarchical FSMs were implemented as general as possible, avoiding constraints both in the number of sub-FSMs per layer, and in the depth of the structure. Therefore, this functionality allows the compactness of the representation, as well as the creation of more complex graphs.

The overall user experience in the code writing process has been improved successfully, making the environment more professional and similar to an IDE. The autosuggestion component, totally created from scratch, implements a smart suggestion strategy, as it is able to recognize which elements (i.e. APIs, custom variables/functions) can be inserted on a specific part of the code, based on what is visible on that scope. This system involved a complex mechanism which takes only the valid code for each selected state, and exploits the Lua parser to get the declared items.

Finally, the improvement of the test-plan validation to increase the global performances of the checking process, obtained very good results. This operation involved the upgrade of the original LuaJ library, a separate work, independent of the *Test-plan Builder*, usable in different projects. The optimization of the process, which removed redundant paths and parallelized the checking operation, drastically reduced the validation time. This achievement made the overall test-plan design better and smoother, as the live check, executed at every modification, became much faster. Additionally, more sanity checks were created to cover more errors and improve the robustness of the system.

7.2 Outlook

The implemented system works very well and all goals were met successfully. Anyhow, there is always space for improvements. A first improvement concerns the *Test-plan Checker*, as more sophisticated checks could be added to cover other errors. For example, an inspection regarding the correctness of the data type of the arguments passed to an API function could be implemented. Moreover, the *Checker* can be optimized, or recreated from scratch, to further increase the validation speed. Another improvement is related to the graph drawing, introducing a grid layout and *cornered* edges, to avoid overlaps and optimize the FSM visualization.

Additionally, the coding environment can be improved, increasing the link between the code and the IDE. This feature can be useful, for example, to directly jump to a variable/function declaration, or to find all usages of it. In the same way, it could recognize more bugs and code smells, both general and project-specific, proposing also an auto-fix. Last but not least, the overall user interface, already modern and dynamic, can be optimized organizing the layout more freely, giving, for example, the possibility to move the side panels in other areas of the window.

Acronyms

µC Micro Controller

API Application Programming Interface

DUT Device Under Test

EDS Electronic Data Sheet

FSM Finite-State Machine

GUI Graphical User Interface

HTML HyperText Markup Language

HW Hardware

IDE Integrated Development Environment

JSON JavaScript Object Notation

KAI Kompetenzzentrum Automobil- und Industrie-Elektronik

MoPS Modular Power Stress

SAM Software Architecture for MoPS

UI User Interface

URL Uniform Resource Locator

XML eXtensible Markup Language

Bibliography

[1] B. Steinwender, S. Einspieler, M. Glavanovics, and W. Elmenreich, «Distributed power semiconductor stress test & measurement architecture», English, in *Proceedings of the 11th IEEE International Conference on Industrial Informatics*, Bochum, Germany, Jul. 2013, pp. 129–134. DOI: 10.1109/INDIN.2013.6622870.

[2] B. Steinwender, «A Distributed Controller Network for Modular Power Stress Tests», Ph.D. dissertation, Alpen-Adria-Universität Klagenfurt, Jun. 2016.

[3] B. Steinwender, M. Glavanovics, and W. Elmenreich, «Executable Test Definition for a State Machine Driven Embedded Test Controller Module», in *Proceedings of the 13th IEEE International Conference on Industrial Informatics*, 2015. DOI: 10.1109/INDIN.2015.7281729.

[4] K. Plankensteiner, «Test Plan Generation and Verification for a Modular Power Stress Test System», M.S. thesis, Graz University of Technology, 2015.

[5] *The JSON Data Interchange Format*, ECMA International, 2013. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (visited on 06/09/2021).

[6] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, «Comparison of JSON and XML Data Interchange Formats: A Case Study.», in *Proceedings of the ISCA 22nd International Conference on Computer Applications in Industry and Engineering, CAINE*, vol. 2009, 2009, pp. 157–162, ISBN: 978-1-880843-73-4.

[7] J. Friesen, «Parsing and creating json objects with gson», in *Java XML and JSON: Document Processing for Java SE*. Berkeley, CA: Apress, 2019, pp. 243–298, ISBN: 978-1-4842-4330-5. DOI: 10.1007/978-1-4842-4330-5_9. [Online]. Available: https://doi.org/10.1007/978-1-4842-4330-5_9.

[8] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, «Lua—an extensible extension language», *Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996, ISSN: 1097-024X. DOI: 10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.

[9] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, «The evolution of lua», in *HOPL III Proceedings of the 3rd ACM SIGPLAN conference on History of programming languages*, 2007. DOI: `10.1145/1238844.1238846`.

[10] R. Ierusalimschy, «Integers in Lua 5.3», in *Lua Workshop 2014*, Moscow, Russia, Sep. 2014.

[11] I. R. Forman and N. Forman, *Java reflection in action (in action series)*. Manning Publications Co., 2004.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994, p. 416, ISBN: 978-0-201-63361-0.

[13] R.-G. Urma, M. Fusco, and A. Mycroft, *Java 8 in action*. Manning publications, 2014.

[14] J. Friesen and S. Pal, *Java Threads and the Concurrency Utilities*. Springer, 2015.

[15] J. Palsberg and C. B. Jay, «The essence of the visitor pattern», in *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*, IEEE, 1998, pp. 9–15.

[16] S. Chin, J. Vos, and J. Weaver, *The Definitive Guide to Modern Java Clients with JavaFX*. Springer, 2019.

[17] R. Ierusalimschy, «Programming in lua», in 4th ed., Lua.org, 2016, ch. 21.