

Test Plan Generation and Verification for a Modular Power Stress Test System

Master's thesis
submitted by

Klaus Plankensteiner

supervised by:

Univ.–Prof. Dipl.–Ing. Dr.techn. Horst Bischof

Institute for Computer Graphics and Vision
Graz University of Technology

and:

Univ.–Prof. Dipl.–Ing. Dr.techn. Wilfried Elmenreich

Institute of Networked and Embedded Systems
Alpen-Adria-Universität Klagenfurt

and industrially supervised by:

Benjamin Steinwender, BSc. MSc.

Kompetenzzentrum Automobil- u. Industrieelektronik GmbH

September, 2015



Affidavit

I hereby declare in lieu of an oath that

- the submitted academic work is entirely my own work and that no auxiliary materials have been used other than those indicated,
- I have fully disclosed all assistance received from third parties during the process of writing the work, including any significant advice from supervisors,
- any contents taken from the works of third parties or my own works that have been included either literally or in spirit have been appropriately marked and the respective source of the information has been clearly identified with precise bibliographical references (e.g. in footnotes),
- to date, I have not submitted this work to an examining authority either in Austria or abroad and that
- the digital version of the work submitted for the purpose of plagiarism assessment is fully consistent with the printed version. I am aware that a declaration contrary to the facts will have legal consequences.

(Signature)

(Place, date)

Abstract

The heart of a new modular power stress test system approach is a test plan which describes the whole system and test procedure. This test plan has to be created in a user friendly way and should be verified. The verification includes, among others, Finite-State Machine (FSM) and Lua script verification.

A Graphical User Interface (GUI) was implemented in Java to ease the task of the test plan creation. This GUI contains a graphical system to create FSMs with Lua code inserted per state. An auto completer and code highlighting system was implemented for the Lua code input to further improve the user experience. Several additional enhancements were added – like a history system, different layouts and visual error indication. Additionally, an automatic build and deployment system was added to ease the deployment of future versions.

The *Test-plan Checker* represents a static checker and was implemented as an extra project which is used by the *Test-plan Builder*. Several FSM conditions had to be checked for this task. The hardest part was the verification of the Lua code. This code is distributed on several FSM states and can be executed in arbitrary order depending on the FSM transitions. The resulting implementation of the Lua checker turned out to be usable in other Lua projects too. Additionally, a simple, experimental simulator was implemented which represents the first steps to add a dynamic checker. The *Test-plan Checker* is also provided as a stand alone command line tool which will be used by the test system for a final validity check before starting.

Zusammenfassung

Das Herz eines neuen modularen Power Stress Tests Systems ist ein Testplan, welcher das ganze System und den Ablauf beschreibt. Dieser Testplan muss benutzerfreundlich erzeugt und auf Fehler überprüft werden. Die Überprüfung beinhaltet unter anderem FSM und Lua Script Überprüfungen.

Eine GUI wurde in Java implementiert um das erstellen zu erleichtern. Diese GUI enthält ein grafisches System zum erstellen von FSMs welche in jedem Zustand Lua code enthalten. Eine automatisch Vervollständigung und ein Code highlighting System wurde für die Lua Eingabe implementiert um die Benutzer Erfahrung zu verbessern. Zusätzlich wurden einige Verbesserungen hinzugefügt – wie ein History System, verschiedene Layouts und visuelle Fehler Kennzeichnung. Es wurde auch ein automatisches Bau- und Verteilungssystem der Applikation erstellt um die zukünftige Entwicklung so einfach wie möglich zu gestalten.

Der *Test-plan Checker* stellt eine statische Fehlerüberprüfung dar und wurde als zusätzliches Projekt implementiert welches vom *Test-plan Builder* verwendet wird. Mehrere FSM Bedingungen mussten für diese Aufgabe überprüft werden. Der schwerste Teil war das überprüfen des Lua Codes. Dieser Code ist auf mehreren FSM Zuständen aufgeteilt und kann in beliebiger Reihenfolge, abhängig von den Übergängen, ausgeführt werden. Es stellt sich heraus, dass die resultierende Implementierung der Fehlerüberprüfung auch für andere Lua Projekte verwendet werden kann. Zusätzlich wurde ein einfacher, experimenteller Simulator implementiert welcher den ersten Schritt in Richtung dynamische Überprüfung darstellt. Der *Test-plan Checker* ist auch als Kommandozeilen Programm zur Verfügung gestellt um dies für eine finale Überprüfung vor dem Teststart zu verwenden.

Acknowledgments

I would like to thank some people who have kindly supported the successful completion of my computer science master studies – Thank you!

Horst Bischof & Wilfried Elmenreich who gave me the opportunity to write this master thesis.

Josef Fugger & Michael Glavanovics who gave me the opportunity to develop the resulting application and write this master thesis under fantastic conditions at KAI.

Benjamin Steinwender who was the best industrial supervisor someone could think of.

Sascha Einspieler who gave me important inputs for the usability of the resulting application.

My family who gave me the financial support and motivation to finish my studies without distractions.

My friends who were responsible for some needed distractions and endless fun in times of boredom.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Goals	3
1.3	Problem definition	4
1.4	Outline of the Thesis	4
2	Basic concepts	7
2.1	Test-driven programming	7
2.2	Reflection	8
2.3	Generics	9
2.4	JSON with Gson	10
2.5	The MoPS testplan	11
2.6	The MoPS FSM	13
2.7	Lua verification	13
2.8	Electronic Data Sheet	15
3	Test plan generation	17
3.1	MoPS FSM input	17
3.2	Lua Input	20
3.3	Ovenplan	23
3.4	Network data	24
3.4.1	Configuration file data	25
3.4.2	MoPS and SAM data	26
3.4.3	EDS data	27
3.5	Loadable parameter	28
3.6	Additional features	29
4	Test plan verification	33
4.1	FSM	33
4.1.1	Dead end nodes	34
4.1.2	Lonely nodes	34
4.1.3	Hardware event	34
4.1.4	Software event	35

4.2	Lua	35
4.2.1	Script generation	35
4.2.2	Code verification framework	36
4.2.3	Code verification	37
4.2.4	General Lua errors	39
4.2.5	Project specific Lua errors	46
4.3	Oven plan	48
4.4	Simple simulator	50
4.5	Standalone checker	51
5	Evaluation	53
5.1	Test-plan Builder	53
5.2	Test-plan Checker	55
5.2.1	MoPS FSM data	55
5.2.2	Lua data	56
5.2.3	Oven plan data	56
5.2.4	Experimental script simulator	56
6	Conclusion	59
	Index of abbreviations	61
	Bibliography	62
A	Sample source code	67

Chapter 1

Introduction

In semiconductor industry, it is very important to know the reliability of devices to avoid malfunction in dangerous situations. To achieve this knowledge for semiconductors, long and accurate stress tests are performed with automatic test systems. Tests which put the devices to their limits until they are destroyed are very common in this industry to gather data of high quality.

Such a test system in a modular approach (software and hardware) controlled by micro controllers and a host PC is currently developed at Kompetenzzentrum Automobil- u. Industrieelektronik GmbH (KAI). It is called Modular Power Stress [1] (MoPS) test system. In this test system, there are 3 dynamically controlled hierarchy levels which provide test management, control of each Device Under Test (DUT), and measurements for diagnosis as shown in Figure 1.1. The following list explains each of these layers:

Test Layer Responsible for test management. This actor handles several node actors which are lower in the hierarchy level and is located on a host computer.

Node Layer Dedicated interface to the μ Controller to enable communication and provide a GUI. One actor handles exactly one μ Controller. The actor is located on the same host computer as the test actor.

μ Controller Executes the test, drives and monitors a DUT. It is located near the DUT in the test system hardware to simplify wiring and is accessed via Ethernet from the node actor.

Each of these levels is working individually based on a test plan in JavaScript Object Notation (JSON) format [2]. Such a test plan consists of one FSM [3] created for each level. The states of the FSM execute Lua code [4] defined by the user through the test plan. The code in the FSMs are able to trigger transitions for the FSMs which are in the same or lower hierarchy.

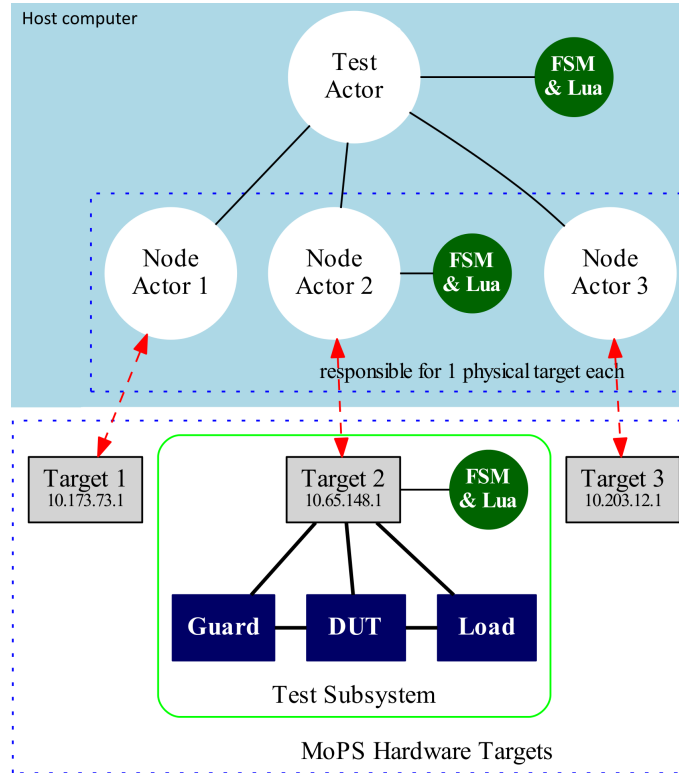


Figure 1.1: Software Architecture for MoPS (Host Software) (Image created by Benjamin Steinwender and modified by Klaus Plankensteiner)

1.1 Motivation

Until now, the test plans were written and designed in a text editor. For complex test plans, this is very hard and error prone, which calls for a more comprehensive tool with a graphical user interface. There is no existing software which could solve this task due to the fact that the test plan is very project specific. It is actually the first project in which Lua code is combined with a FSM. Therefore, an own implementation is needed which is described in Chapter 3.

Additionally, the generated test plan should be as error resistant as possible. This resulted in the implementation of a test plan checker which is described in Chapter 4. The Lua check could have been a part which is already available because the Lua scripting language is very common and available since 1993. But research (summarized in Section 2.7) resulted in a terrible insight – there is no flawless Lua verification tool available to the public.

1.2 Goals

There is a need for verifying the test plan (especially the contained Lua code) since it is not verified in the current software version. The key goal of the master thesis is to simplify the creation of the test plan and apply verification to it. Basically, an evaluation to which extent the test plan can be validated is performed. This includes Lua script and FSM structure. Static methods are applied to detect the following errors:

- Testplan file
 - JSON format errors
- FSM (Section 4.1)
 - Deadlocks (e.g. Node which has no outgoing edges)
 - FSM structure errors
 - Unreachable nodes
 - Possible transitions
 - Empty IDLE node
- General Lua (subsection 4.2.4)
 - Always the same or unreachable code (e.g. if a \neq a then end)
 - Empty code blocks (e.g. if/else block)
 - Infinite loops (e.g. while true do end)
 - Unused variables
 - Use of uninitialized variables
 - Lua syntax errors (checked with parser)
- Project specific Lua (subsection 4.2.5)
 - Used MoPS-CORE Application Programming Interface (API) with hardware description
 - Used Software Architecture for MoPS (SAM) API

Thus, the first step is to provide an intuitive application to generate the test plan. This application includes an intelligent auto-complete system for the Lua code to support the input for lab engineers which are not used writing in the script language Lua. The second step is to verify the generated test plan by checking the FSM, Lua code and parameters. This drastically improves the quality of the used test plans and reduces the risk of life test crashes. To further increase the maintainability,

these two steps are separated and the test plan generation application will use the verification tool as a separate module.

Parsing of the Lua code is not required since there are already existing tools which are able to handle this task. Therefore, this task is not part of this master thesis.

Since this project is written in Java and verifies Lua code, all code listings and explanations are focused on these two languages.

1.3 Problem definition

This thesis is concerned with the user friendly creation of the test plan file and error detection in this file for the MoPS system. Therefore, the following questions arise:

- How to visualize the creation process to provide an intuitive user experience?
 - How to build the FSMs?
 - How to easily provide Lua code input to non computer experts?
 - How to provide the MoPS-CORE and SAM API in the Lua code?
- How to detect as much errors as possible in the test plan?
 - How to verify the FSMs?
 - How to verify the Lua code?
 - How to generate all possible Lua part combinations from the FSMs for verification?
 - How to verify the in Lua code used API?
 - How to verify the machine is able to use the API?
 - How to verify the oven plan?

1.4 Outline of the Thesis

This sections concerns with the outline of this thesis and will give some reading advice.

The introduction is found in Chapter 1 and describes the motivation, goals and problems of this thesis. It is highly interesting and should not be skipped by anyone who want to get more insight into this thesis.

Basic concepts are explained in Chapter 2. This includes basic software concepts and concepts of the parent project. If you do not already know the parent project, you should definitely read about the MoPS test plan in Section 2.5, the MoPS FSM in Section 2.6 and the Electronic Data Sheet (EDS) file in Section 2.8. This knowledge is mandatory to understand certain aspects of this thesis.

The creation of the test plan is described in Chapter 3. That section concerns mainly with the GUI. If you are not interested in GUI design and user experience enchantments, just have a look at the network data section (see Section 3.4) and the loadable parameter section (see Section 3.5).

The test plan verification is described in Chapter 4. It concerns with the static verification of FSM, Lua and oven plan. Additionally, a dynamic approach for the Lua verification is described in Section 4.4.

An evaluation of the implementation is done in Chapter 5 and the conclusion follows in Chapter 6.

Chapter 2

Basic concepts

For better understanding of this thesis, some basic concepts are explained in this chapter. These basic concepts contain computer science topics and some important concepts of the parent project, the doctoral dissertation of Benjamin Steinwender [5]. Additionally the usage of these concepts in this thesis is explained too.

2.1 Test-driven programming

Test-driven programming [6] is a simple but powerful software development process. The idea behind it is to design test procedures before implementing the system. This makes the developer focus on the requirements before writing new source code which is a small but powerful difference. Additionally, the code is already tested which increases quality and maintainability a lot. The only downside of this concept is the time overhead which is needed to write test cases before implementing small features. This process consists of 4 steps as seen in Figure 2.1.

Define requirements The first step is to define all requirements and exception conditions. It can be accomplished, for example, through use cases and user stories. This step is one of the most important ones since all test cases and the final implementation will be written to meet these requirements. Changing the requirements afterwards results in time consuming re-factoring of test cases and the function. In the worst case this also influences other implementations.

Implement test The second step are the tests. These will ensure that the function will meet the requirements and are the backbone of this development process. It does not matter in which test framework they are written, but the more precise these tests are, the better. It is also important to test all corner cases.

Implement/Re-factor function The third step is the implementation of the actual function.

Test passed? The last step is to test the implementation with the implemented test. If it passes, the development process of this function is over, otherwise the process continues in step three by re-factoring the function.

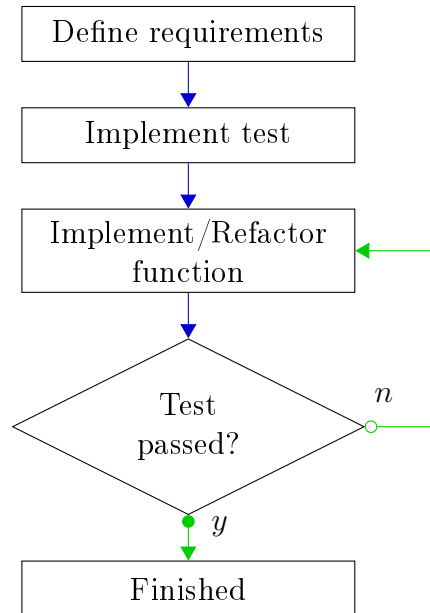


Figure 2.1: Test-driven programming workflow

This development process turned out to be very useful during implementation of the Lua code checker (see Section 4.2). First, the errors which had to be recognized were defined by test cases with the JUnit framework [7] and only then the check for a specific code attribute was implemented.

2.2 Reflection

Reflection [8] is the ability of a computer program to observe and modify its own structure. In the beginning, computer programs were created in their native machine language. These languages were inherently reflective since you could define instructions as data and use self-modifying code. Over time, higher-level languages like C were developed to ease the process of programming and therefore the ability of reflection got lost. In 1982, Brian Cantwell Smith introduced the idea of reflection in higher-level languages in his doctoral dissertation [9].

A big drawback of reflection is performance. In most cases, it is significantly slower than its static counterpart because of string comparisons and additional needed validity checks. Furthermore, the compiler is not able to optimize the code as it would do for static code. Nevertheless, it is a great way to increase runtime flexibility on parts of the application where performance is not necessary.

Among other things it is possible to read out all members of an object as shown in Listing 2.1. This easy way to inspect and modify an object in addition to generics (see Section 2.3) is heavily used in this thesis to alter configuration Bean [10] classes with a user defined configuration file (see Section 3.5).

Listing 2.1: reflection.java

```
1 public void reflectionExample(Object o){
2     for (Field f : o.getClass().getDeclaredFields()) {
3         String fieldName = f.getName();
4         Object fieldValue = f.get(o);
5     }
6 }
```

2.3 Generics

Generics [11] is the ability of a programming language to use parametrized objects. The type of the object is specified when the function or class is used. This is needed to avoid duplicate functions or classes in type safe languages. A very simple example would be a container which holds one arbitrary object. With generics it is possible to define one single class which changes the data type of the object depending on the usage. In the example shown in Listing 2.2 the object in the container is an Integer.

Listing 2.2: GenericContainer.java

```
1 class Container<T>{
2     T object;
3 }
4
5 Container<Integer> c = new Container<Integer>();
```

The *List* structure in Java is implemented with Generics. A big advantage of Java unlike other languages is the possibility to use the extend keyword with generic types. This allows e.g. to define a type as *Comparable* which forces all objects used with this generic to extend the defined class. Therefore, it is possible to implement a parametrized *List* data structure which provides a *sort* function.

This mechanic is used in this thesis for loading configuration Beans with default parameters. The reason why generics is used instead of polymorphism is, that this forces both function parameters to be the same type and therefore have the same fields. A simplified version of overwriting null values by default values is shown in Listing 2.3. Since this code snippet uses generics and reflection (see Section 2.2) it is highly flexible and can be used with arbitrary classes.

Listing 2.3: generic.java

```

1 <T> void overrideNullValues(T toOverride, T values) {
2   for (Field f : toOverride.getClass().getDeclaredFields())
3     {
4     if (f.get(toOverride) == null){
5       f.set(toOverride, f.get(values));
6     }
7   }
}

```

2.4 JSON with Gson

JSON [2] is a very popular and compact data format which is used in many applications for data exchange. The two most important advantages of this format is the simple usage and the human readable format. Furthermore, the encoding and decoding of this format is implemented in almost every programming language¹. This enables e.g. system environments to encode data on the server, send it to the client, and decode it there to use the data in a structured way.

Google developed the application Gson [12] on top of JSON. Gson is able to take a JSON string in addition to an arbitrary class and tries to parse the data into an object of this class with the help of reflection Section 2.2. Therefore, it is only needed to define a class which represents the JSON object to parse the data into an useful data structure. A simple example of Gson in action is given in Listing 2.4.

Listing 2.4: Simple Gson example

```

1 public class Person {
2   public String name;
3   public Integer age;
4 }
5 String jsonString = "{\"name\":\"Max\",\"age\":20}";
6 Person p = new Gson().fromJson(jsonString, Person.class);

```

Lines one to four define the class. This class is represented by the JSON string in line five. Line six converts the string into an actual object.

Gson is very simple to use as shown above, but it also has its downsides. It is not possible to define standard values for a non present value in the string. If the value is not in the string, the member gets the *null* value assigned. Another issue is object referencing. If there are two members which refer to the same object, the object is written twice and duplicated on reloading the string. This is the only behavior which prevents the usage of Gson on complex objects.

¹Available JSON parser (September 2015): <http://json.org/>

node actors. There has to be exactly one defined FSM with the same name and of type *test*.

FSM Multiple FSMs are possible. Defines the behavior of an actor in form of a FSM and Lua code. Since one actor uses one FSM, there are maximal as much FSMs as actors.

type There are 3 possible actor types: *test*, *node*, *uc*. Actors are only allowed to use FSMs of the same type. This field is only needed for the *Test-plan Checker* to be able to generate all errors at once. If this field would not be present, the checker would not know what to check on an unused FSM. For example, the MoPS-CORE API is only allowed on the μ C actor.

name Name of the FSM. Used for assigning a FSM to the corresponding actor. This name has to be unique for a distinct assignment.

Node Represents one state in the FSM.

name Name of the node in a FSM. Used for assigning nodes in transitions. This name has to be unique in the FSM for a distinct assignment.

code Lua code which is invoked on state transitions. Transitions in own FSM or FSMs of child actors can be triggered here with the transition name (event). Furthermore, modules defined in MoPS-CORE can be called per layer, e.g. the MoPS-CORE API is available on the μ C actor FSM. The scripts on each node are called in the same Lua instance. This provides the possibility to e.g. define variables in one state and use them in another state.

Transition Possible transitions between the nodes. The code in the new node will be invoked on transition.

currentNode Name of the current node. A transition is only able to occur if the current state represents the current node.

event The name of the transition. According to convention, hardware events (triggered by the hardware) have the prefix *@* and software events (triggered by the software through the Lua code) do not have this prefix. Additionally, the transition name *@else* is available which indicates a transition that occurs after the end of the Lua code execution of one state. The combination of *currentNode* and *event* has to be distinct per FSM to guarantee an unambiguous use of transitions.

nextNode The name of the new node (state) after a transition.

Ovenplan The oven plan describes the required hardware for this test and maps the hardware to the corresponding FSMs.

slot Location of the DUT in the machine.

- dut** Name of the DUT.
- node** IP addresses of the μ C node. This is needed because the communication between node and μ C actor is performed over Ethernet.
- ucTarget** Name of the hardware target. Defines for example the μ C hardware which is then checked against the used hardware in the machine.
- applicationModule** Name of the test application. Defines the application hardware which is then checked against the used hardware in the machine.
- nodeFsm** Name of the FSM running on the node actor responsible for this DUT. This FSM controls the corresponding μ C FSM defined through the *uCFsm* entry. There has to be exactly one defined FSM with the same name and of actor type *node*.
- uCFsm** Name of the FSM running on the μ C actor responsible for this DUT. There has to be exactly one defined FSM with the same name and of actor type *uc*.

2.6 The MoPS FSM

The MoPS FSM is an FSM with Lua code in the nodes and a modified transition mechanic. This mechanic is indicated in Figure 2.3. If the MoPS system is in a state, the Lua code of the corresponding node is executed. At the end of the execution a check for events is performed. If an event and transition name matches, the FSM changes its state. If there is no matching transition, either the *@else* transition is performed if present, or the same node is executed again.

There are two different types of events.

- Hardware events are triggered in the hardware and do have a naming convention: prepended @.
- Software events are triggered in the software through the MoPS-CORE and SAM API in the Lua code.

2.7 Lua verification

Lua[4] is a small and simple scripting language which is very widespread². It is also used in the test plan of the MoPS test system to describe the behavior of a test run. Since this thesis concerns with the verification of the test plan, Lua script verification has to be done. This section will discuss the need of an own verification implementation and different solution approaches.

²Where Lua is used (September 2015): <https://sites.google.com/site/marbux/home/where-lua-is-used>

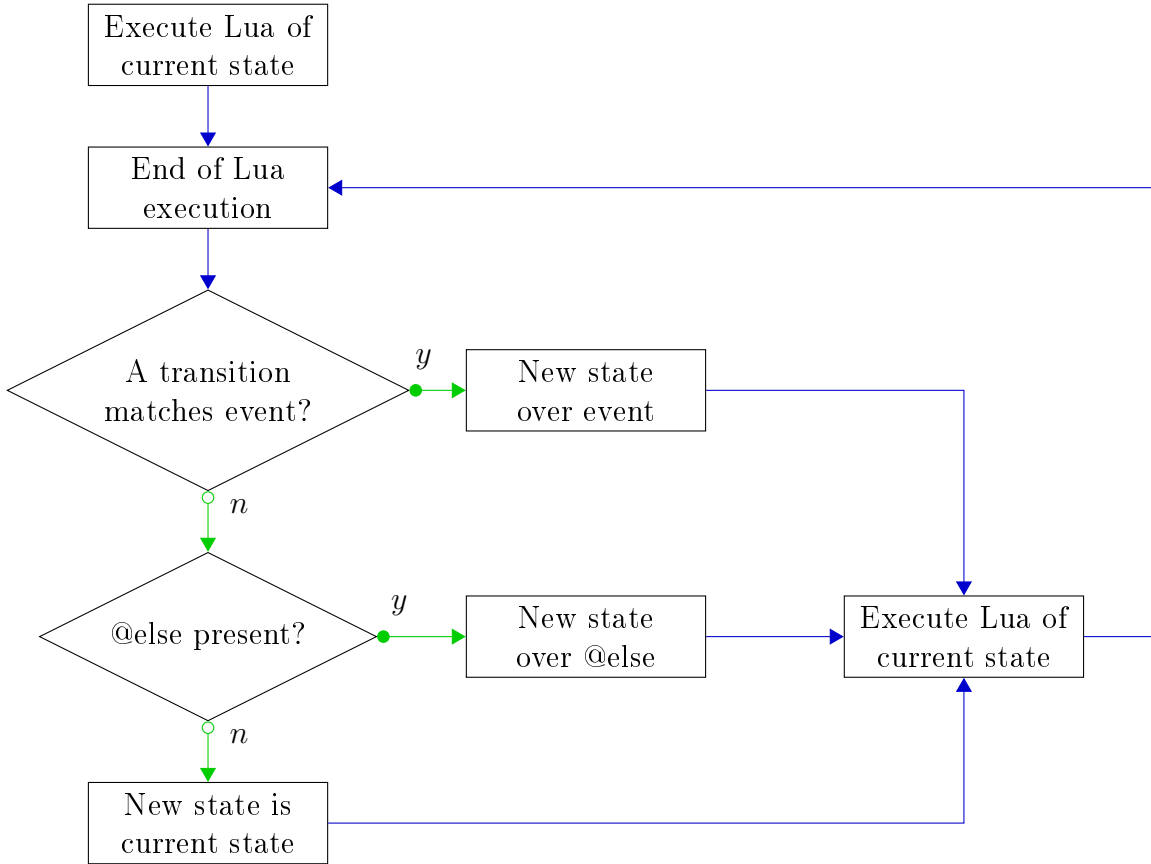


Figure 2.3: MoPS FSM state transition

The search for and evaluation of existing applications, which are able to verify Lua scripts, resulted in a terrible insight. There is currently no satisfying application for this task. All of them are in preliminary state, not developed anymore and does not fulfill the required error detection rate. Following most promising applications were found, evaluated and rejected:

LuaLint³ It does only static analysis of global variable usage in Lua source code. It does not provide any possibility to travel Lua code in a convenient way or create an Abstract Syntax Tree (AST). It is not moderated since 2004.

lua-checker⁴ It does only implement checks on variables. It has no possibility to travel Lua source code. It is not moderated since 2013.

LuaInspect⁵ It does not find uninitialized variables. It has no possibility to travel Lua source code. It is not moderated since 2011.

³<https://github.com/philips/lualint>

⁴<https://code.google.com/p/lua-checker/>

⁵<https://github.com/davidm/lua-inspect>

LuaFish⁶ The source analysis part is superseded by LuaInspect. Different from LuaInspect, it is possible to travel Lua code in a convenient way. Therefore, this would have been the most promising application. It is not moderated since 2011.

The next idea to perform the error detection is to transform the script into another language and check it in this language. This would be feasible, but a big problem is the retrieving of the error location in the original language and occurrence of side effects. A similar approach is described in a paper [14] from 2014, which tries to translate a Lua program to ANSI-C and verify it in this language.

The need of some project specific Lua verification and the above evaluations resulted in an own implementation described in Section 2.7.

2.8 Electronic Data Sheet

Electronic data sheets have already been proposed for sensors in distributed monitoring and configuration architectures [15]. In this work, EDS documents are files which are located on every μC in the MoPS test system. These files describe the hardware of the corresponding μC . Each of them are also available on a web server to be fetched by the applications developed in this thesis.

The content of such a file is crucial for the test plan verification part in this thesis. A list, which describes some of the information contained in an EDS file, is following. This information is used by the test plan checker as described in Chapter 4.

MoPS-CORE version This defines the MoPS-CORE API version the μC is capable of. It describes the functionality of the Lua modules available on corresponding μC . The test plan checker uses this to know the available functions per MoPS-CORE module.

IP Since the EDS files are loaded from a web server, there is no actually connection to the μC on which this file is located on. This is fixed with this field. Every μC has an IP address and therefore it is possible to match a μC to the corresponding EDS file over the oven plan located in the test plan (Section 2.5). The test plan checker uses this to map EDS files to the FSMs specified in the test plan.

Events This represents a list of possible hardware events. The test plan checker uses this to perform advanced verification on the FSM structure.

Modules This represents a list of available modules and module instance names. Only the intersecting modules defined in this list and in the MoPS-CORE API documentation are accessible in the Lua scripts. Therefore, the test plan

⁶<https://github.com/davidm/lua-fish>

checker uses this information to verify the right usage of the available modules in the Lua scripts.

Chapter 3

Test plan generation

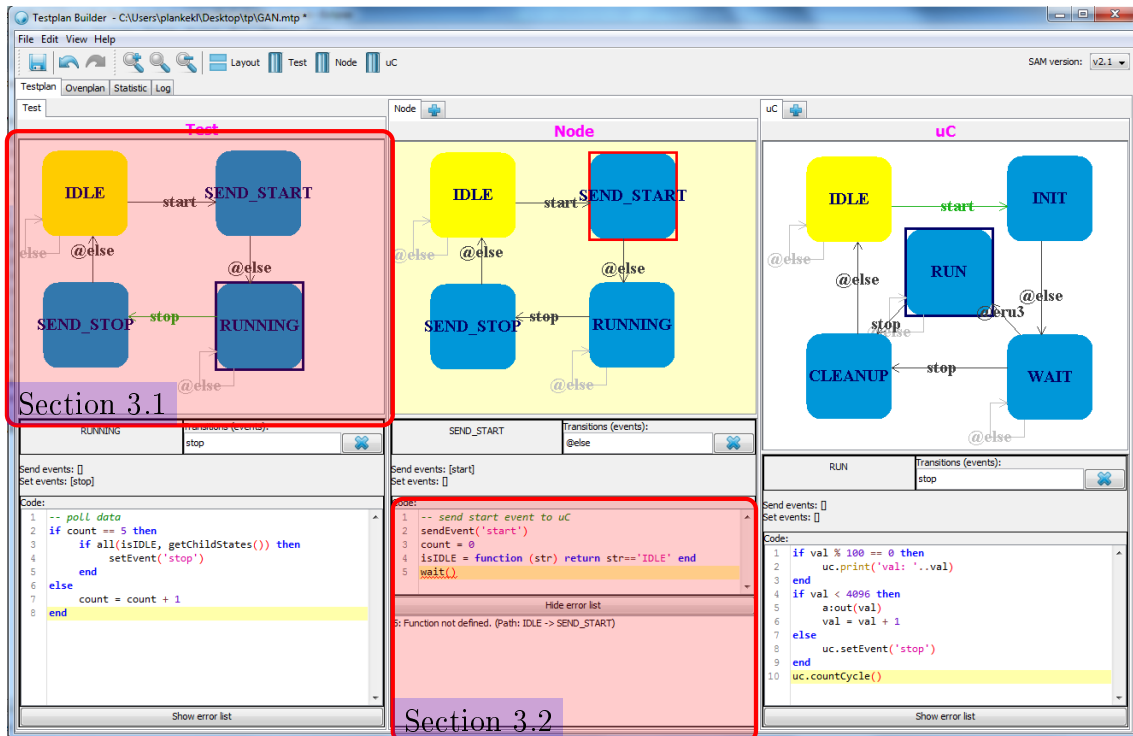
A graphical user interface application called *Test-plan Builder* was developed to generate test plans in the most convenient way. This application is able to create a test plan and afterwards export this data to a test plan file (see Section 2.5). Additionally, it uses the *Test-plan Checker* (see Chapter 4) to visualize errors in the test plan. Only valid test plans can be exported. Therefore the development process of a test plan is a lot easier, faster and error resistant than writing the test plan file by hand.

The result is visible in Figure 3.1. Marked areas in this figure refer to a section with implementation details of the corresponding area. These marks include the FSM generation part of Section 2.6 and the Lua input part of Section 3.2. Another important area in the *Test-plan Builder* is the oven plan area. This is hidden in that figure but is explained in detail in Section 3.3. The general work flow of the application to create a test plan is following:

- Draw the needed FSMs. This will represent the different states of the MoPS test machine and forces the user to think about the test procedure. The names of the nodes should represent the activity during this state. See Section 3.1 to get more details on how to do this and how it is implemented.
- Fill the FSM nodes with Lua code. This code is controlling the hardware at this state and is responsible for the whole test procedure. See Section 3.2 to get more details on how to do this and how it is implemented.
- Apply the FSMs to the corresponding hardware by creating the oven plan. See Section 3.3 to get more details on how to do this and how it is implemented.

3.1 MoPS FSM input

This section will focus on the MoPS FSM input. Refer to Section 2.6 if the MoPS FSM concept is not already known. The creation process is a very easy and intuitive one. A drawing space is provided where nodes can be drawn and connected.

Figure 3.1: *Test-plan Builder*

The draw process of a node can either be managed with *right click* → *New Node* (see Figure 3.3) or with *CTRL + left click drag* on a free space. The second option has the advantage that it is very fast and it is possible to adjust the size of the node during creation. The node name is checked for all current nodes in the same FSM on creation. A new name is asked for in case the user enters a duplicate name. It is also possible to adjust the size of a node after creation by click and drag one of the edges. Re-positioning is also possible by just click and drag the node. Modifying the color and name of a node is done by *right click* → *Edit Node* as seen in Figure 3.2. Deletion of a node is possible by selecting and *right click* → *Delete* or pressing *DEL* on the keyboard. Additionally it is possible to copy and paste a node by the usual *CTRL - C* and *CTRL - V*. All of these functions are also accessible with the context menu or as shortcuts as seen in Figure 3.16. Furthermore, most of them are possible to apply on multiple nodes at the same time. Additionally, it is possible to copy nodes between different FSM areas or share nodes with any social application since the copy - paste mechanic serializes the node into the clipboard. E.g. copy a node, paste it into an e-mail (it will paste a string), copy the string from the e-mail and paste it into the *Test-plan Builder*.

A system to define the current active FSM is needed since most modifications can also be performed from the global *Test-plan Builder* menu bar or by shortcuts.

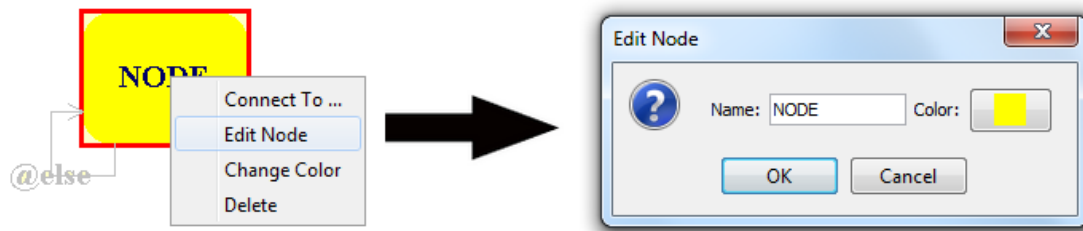


Figure 3.2: Edit node

The recently active FSM panel is remembered in a database. This is updated if any changes happens to the FSM view, e.g. re-sizing it, changing data inside or clicking into it. The background of the current active FSM has a different color than the others. This indicates the current active FSM to the user and, therefore, the user will always know which FSM the action will be performed on.

The second important draw process is to connect nodes to represent transitions in the FSM. This is done either with *right click* → *Connect To...* or *CTRL + left click drag* from one node to another. The later option has the advantage that it is very fast. The transition name is checked for all current transitions leaving the same node on creation. A new name is asked for if this name does already exist. The **+** sign above the FSM area is able to create an additional FSM area for this

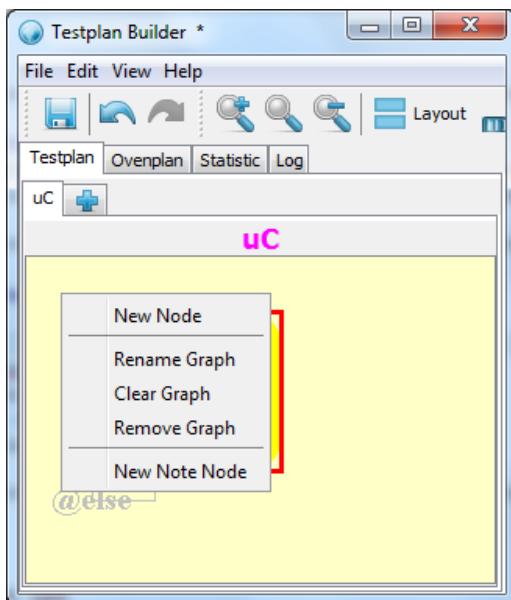


Figure 3.3: Free space right click

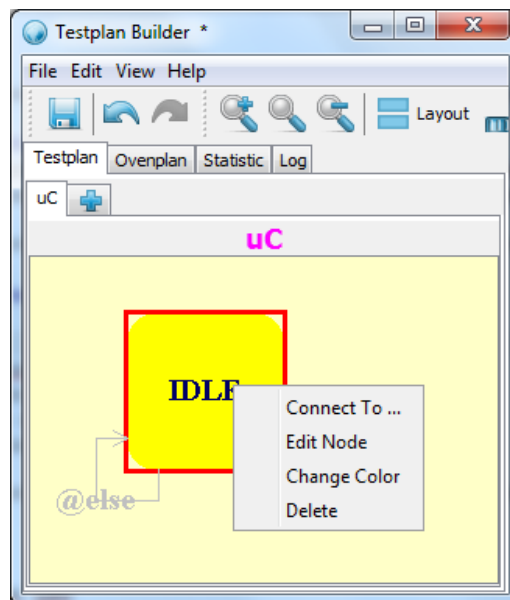


Figure 3.4: Node right click

actor type since multiple different FSMs are possible on one actor type. There are option for removing, renaming or clearing the FSM in the context menu as seen in Figure 3.3.

Sometimes FSMs are getting really big. Therefore, it is important to provide enough space for the designing part. This is the reason why a layout and zoom mechanic was implemented. It is possible to hide FSM areas with shortcuts (refer Figure 3.16) and zoom in or out of them as seen in Figure 3.5. It is also possible to place graphical notes which are also present in this figure. Although these notes will not be in the generated test plan file, they do hugely improve usability because they enable grouping of nodes in a visual way.

Hierarchical FSMs (represent a whole FSM with one state) are not supported by the MoPS system. Nevertheless, it would be possible to provide this functionality in the graphical user interface and convert the FSM for the test plan. This is not implemented in the current state but will be in the future. There is only one downside to this, sub FSMs are lost if the user imports the test plan instead of the *Test-plan Builder* save file.

Figure 3.6 shows all FSM view improvements at a glance.

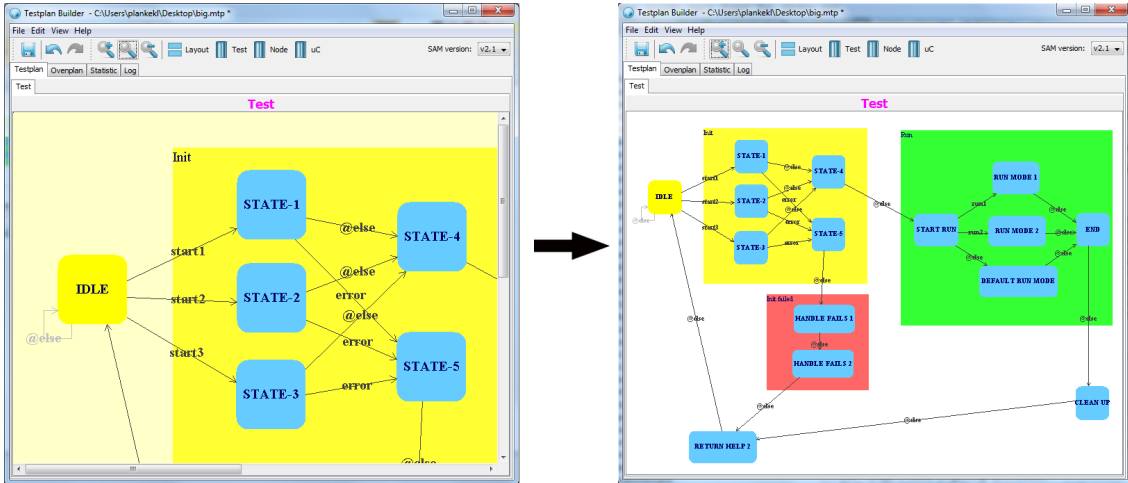


Figure 3.5: Zoomed big test plan with graphical notes

All of this was implemented from scratch by extending a JComponent. This method was inspired by the graphpanel project from John Matthews [16].

3.2 Lua Input

A code panel shown in Figure 3.7 is provided per FSM. This panel contains the Lua code of the current selected FSM state. Code input would not be convenient if two important components are missing.

The first is the code highlighting. This one is added by using code from a Free and Open Source Software (FOSS) [17] project called RSyntaxTextArea [18]. It provides a class which implements code highlighting and code folding. Since it is inherited from a JTextArea it is very convenient to use and to integrate. Furthermore, it

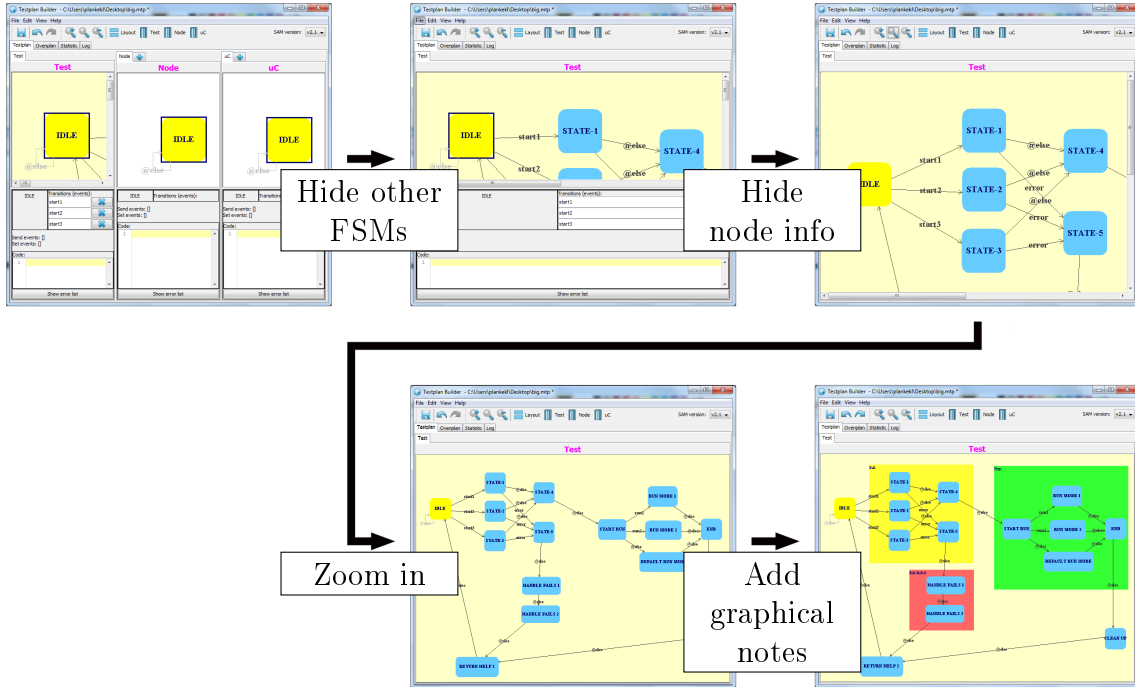


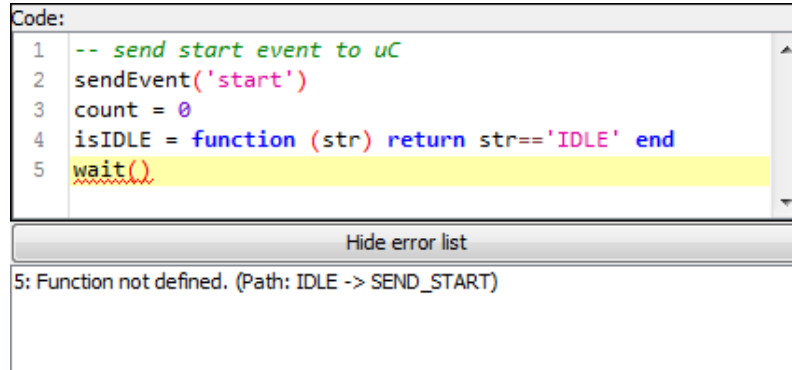
Figure 3.6: FSM view improvements at a glance

natively supports highlighting of Lua source code. Another highlighting library called JSyntaxPane [19] was found but this project is not maintained any more and does not support Lua.

The second important component is the auto-completer. A project called AutoComplete [20] is the chosen one to implement this task. It is maintained by the same people as the RSyntaxTextArea and is also freely available. AutoComplete has especially good compatibility and features for RSyntaxTextArea because both projects are a result from the same parent project. There is only one drawback to this project, it only implements the visual part and does not gather the correct content like available functions or variables. Thus, there is another project needed for this – fortunately the creators of RSyntaxTextArea also have such a project called RSTALanguageSupport [21]. Unfortunately it does not support Lua.

Thus, a simple dedicated implementation is needed. This implementation only takes Lua keywords, functions and modules of the MoPS-CORE API into account which are parsed and cached from the API documentation located on a Transport Layer Security (TLS) secured web server. This API documentation provides every detail needed for the auto completer as you can see in Figure 3.8 and Figure 3.9.

An error indication is located on the bottom of the text area in addition to the Lua input. This error indication is a list which shows all detected errors by the *Testplan Checker*. A mapping between error script lines and state lines is created as shown in Figure 3.10 to convert script lines of the error to state lines. This is needed

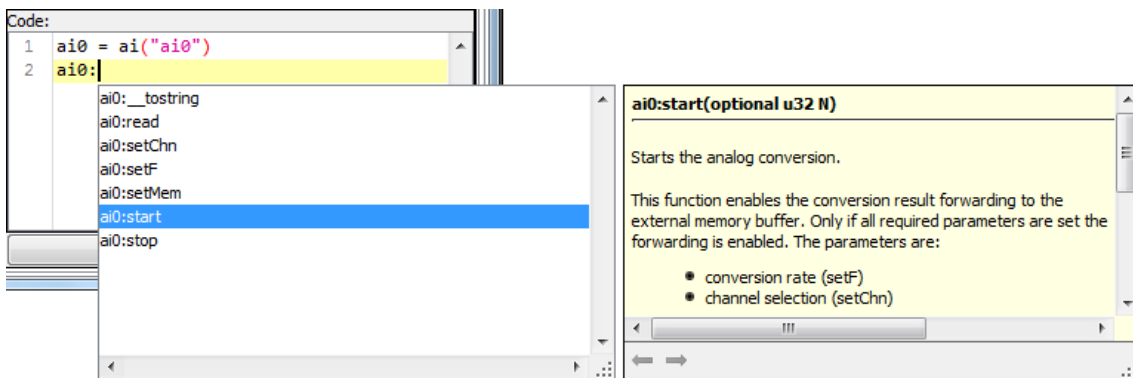


```
Code:
1  -- send start event to uC
2  sendEvent('start')
3  count = 0
4  isIDLE = function (str) return str=='IDLE' end
5  wait()
```

Hide error list

5: Function not defined. (Path: IDLE -> SEND_START)

Figure 3.7: Lua input



```
Code:
1  ai0 = ai("ai0")
2  ai0:
```

- ai0: __tostring
- ai0: read
- ai0: setChn
- ai0: setF
- ai0: setMem
- ai0: start
- ai0: stop

ai0:start(optional u32 N)

Starts the analog conversion.

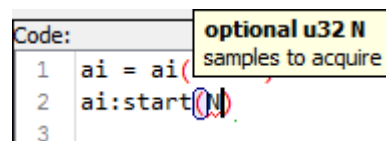
This function enables the conversion result forwarding to the external memory buffer. Only if all required parameters are set the forwarding is enabled. The parameters are:

- conversion rate (setF)
- channel selection (setChn)

Figure 3.8: AutoComplete

because the *Test-plan Checker* takes only valid full scripts, and the Lua code of a state is only a part of it. This enables the *Test-plan Builder* to visually underline the errors because the exact location is known. Additionally, it is possible to jump to the corresponding Lua code location with a double click on an error in the list.

An important factor in user interface design is to be always responsible. This means an user interface should never be blocked by a long running function. Therefore, all heavy working processes (like the checker) have to be outsourced from the GUI thread to a worker thread. An extra checker thread is implemented which is triggered on data change. After this thread is finished, it updates the GUI to indicate found errors. An advanced trigger mechanic was developed since a simple implementation would trigger a lot of checks. In this advanced mechanic, a flag is



```
Code:
1  ai = ai(
2  ai:start(N)
3
```

optional u32 N
samples to acquire

Figure 3.9: AutoComplete - Parameter Completion

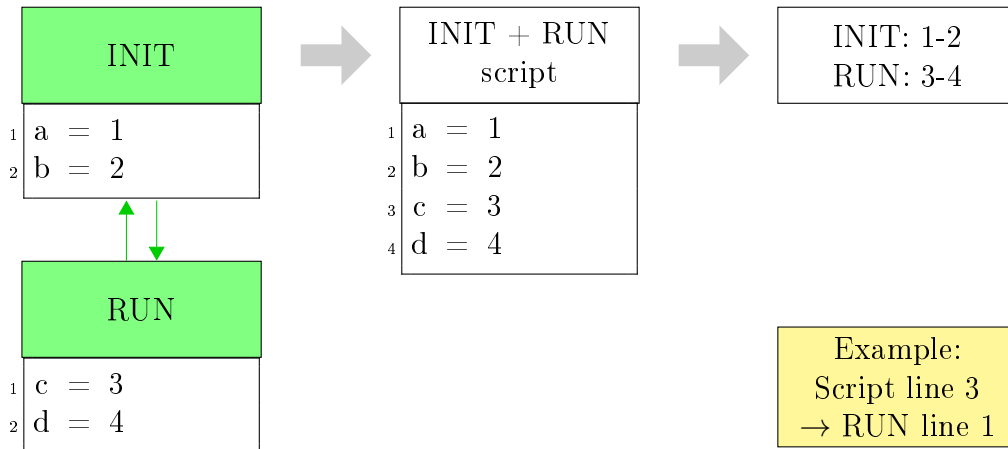


Figure 3.10: Script to state line mapping

set to 1 on data change and the checker thread is started if not already running. The checker thread sets the flag to 0 at start and reruns at the end if the flag has the value 1. This mechanic ensures three things:

1. Only one checker thread is running (and only if needed).
2. Always the latest data state is checked.
3. Some data states are skipped if the checker is not fast enough.

Therefore, the delay between data change and error visualization is defined as:

$\Omega(t)$ and $\mathcal{O}(2t)$, where t is the time of one complete check.

3.3 Ovenplan

The oven plan is a mapping between multiple components. The available components with explanations are listed in Section 2.5.

An input has to be provided which maps each DUT to the corresponding values. Therefore, the input is represented as a table as seen in Figure 3.11.

A usability feature is the last line. It is always empty to provide the space for new entries. A new empty line is automatically displayed if the user inserts a value into the last line. Multiple selected lines can be deleted or duplicated with the context menu.

There are different input methods depending on the column. The *Slot* and *DUT* columns are normal text fields because there is no information available about these strings. The remaining fields are combo-boxes with a forced auto completer. This means it is possible to write in the field but it does only accept values from a predefined list as seen in Figure 3.12.

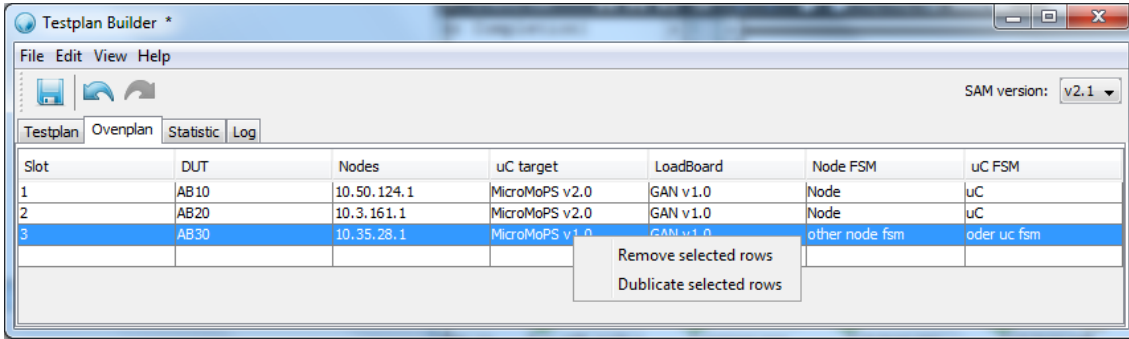


Figure 3.11: Oven plan input

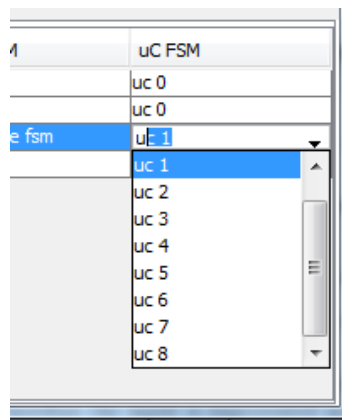


Figure 3.12: Forced auto completer

This list is either generated by the *Test-plan Builder* itself (node FSM and μ C FSM) or it is loaded from a web server (see Section 3.4) and locally cached for the other fields.

3.4 Network data

The MoPS-CORE and SAM API is still in development and will be changed over time due to user feedback and hardware changes. This is the reason why hardware description files and configuration files have to be provided to the *Test-plan Builder* and *Test-plan Checker*. Therefore, a system to provide external data to the application has to be implemented. The following three sections will explain how this is achieved. The general idea is to load the external data from a web server over a TLS secured connection, parse it into a usable data structure and cache it for offline sessions.

3.4.1 Configuration file data

The configuration files include the *uC-HW-targets.json* and *application-modules.json* file. Both files contain a JSON string as indicated by the file name with version, date and other fields. It is possible to create a base configuration class with version and date since these two fields exist in every configuration file. This enables the base class to compare configuration file versions and, therefore, the application knows when a new version was downloaded. Furthermore, the base class is able to provide a *loadFromCache()* and *loadFromNetwork()* function since all child classes have to implement a *loadFromJson()* method as shown in Figure 3.13.

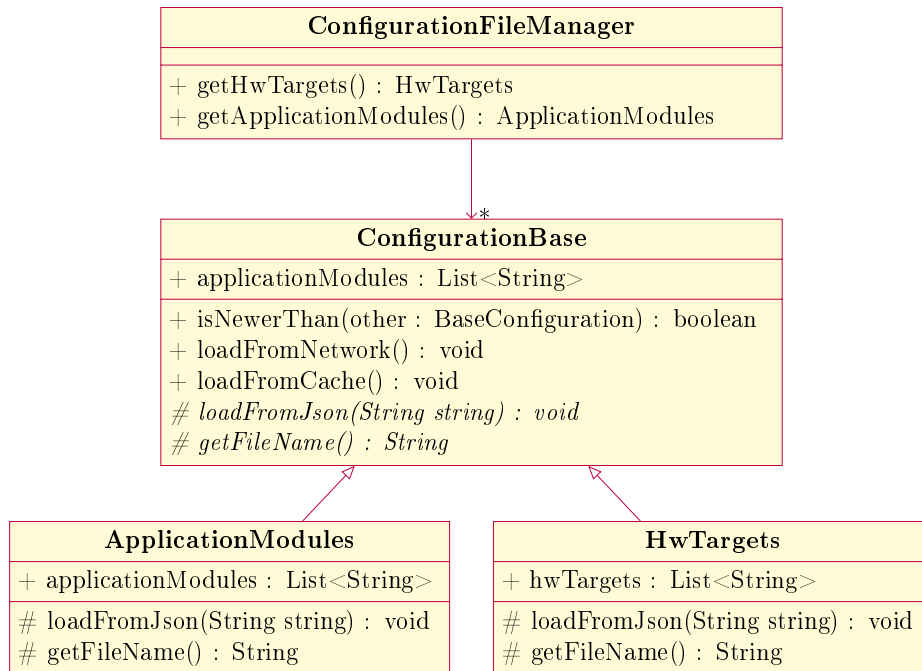


Figure 3.13: Configuration file management UML diagram

A configuration file manager class is able to handle download, update and caching to make this system as extendable as possible. This manager is a singleton [22] containing all configuration file classes in an array. The update is done with a little trick. For each configuration file object in the array a clone is produced with reflection to keep the object type. Afterwards, the values from the first object is loaded from cache and the values from the second object is loaded from network with the provided functions defined in the base class. Finally, a check for the newer version is performed with the *isNewerThan* function.

The implementation allows to add new configuration files in a simple way. This is done by inheriting a new configuration file class from the base class and by adding this to the array in the manager class.

The application is always started with the configuration file data cache, updates the caches in another thread and notifies the user if the cache was updated to

reduce waiting times. The update thread is exposed by the manager. This enables the application to wait until the update is complete before exiting. The cache represents a JSON file from the actual object. This enables experts to inspect the cache since a JSON string is a clear text string. Furthermore, the readable cache has already proven worthy a lot of times during development.

3.4.2 MoPS and SAM data

The MoPS-CORE and SAM API documentation is available on a web server. Since these two documentations do have the same format, they use the same system in the *Test-plan Builder* and *Test-plan Checker*. The documentation is generated with LDoc [23] and, therefore, changes in the API can easily applied to the documentation. This documentation is the perfect location where to get the API specification since it is always up to date. There is one directory on the web server containing all documentation versions. This ensures to not break older test plans which are written for an older API version. Since LDoc puts useful HTML tags into the generated documentation it is very easy to parse it and therefore there is no other representation of the API needed.

The implemented parser needs to know the URL to the directory with the documentation versions. From there on it deletes local version which are missing on the server or adds new local versions which are new on the server. A documentation version is parsed into a suitable data structure which contains the actual version and a list of all found classes and modules in the documentation. The parsing process is indicted in Figure 3.14. One parsed class or module includes everything useful from the documentation to not only be able to provide data to the *Test-plan Checker* but also to provide the data to the auto completer. Such data includes names or usage info – just everything which is available in the documentation. Therefore, the user typically does not need to access the API documentation on the web server at all since every information from the API documentation is also provided through the auto completer.

The data is managed by a MoPS-CORE and SAM manager which is implemented with the singleton pattern. These managers handle download, update and caching of the API documentations.

The application is always started with the API documentation cache, updates the caches in another thread and notifies the user if the cache was updated to reduce waiting times. The update threads are exposed by the manager in the same way as in the configuration file section. This enables the application to wait until the updates are complete before exiting. Since the plain text caching in a JSON file has proven very useful as stated in the previous section, it is also used here to cache the actual object.

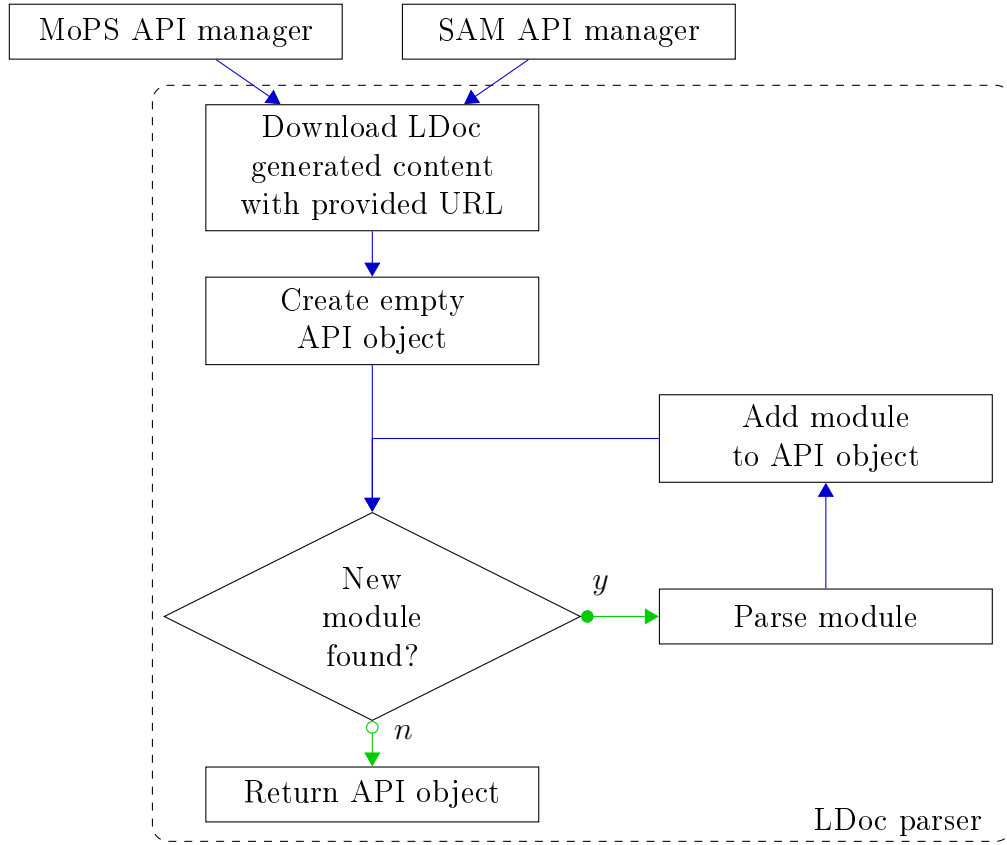


Figure 3.14: MoPS-CORE and SAM API parser

3.4.3 EDS data

An EDS file describes the details of the μC hardware on which the generated test plan is executed. Therefore, knowledge of this data sheet is helpful for developing a test plan and should be provided to the user. A detailed explanation of this file is available in Section 2.8. The current section concerns only with the retrieving part and not with the content of such a file.

All EDS files are available in a directory accessible over a web server. An EDS manager class implemented with the singleton pattern is responsible for the download, update and caching of these files. A PHP script is located on the web server and returns all EDS file names and the corresponding content hash of the file. This enables the manager to find updated EDS files without downloading all of them. Additionally the manager removes locally cached EDS files which are no longer on the server.

The caching is performed similar to the other network data. The application is always started with the EDS files cache, updates the caches in another thread and notifies the user if the cache was updated to reduce waiting times. The update thread is exposed by the manager in the same way as mentioned in the previous sections. As

already stated, the caching in a plain text format is very useful. Therefore this cache is also implemented with an JSON file to enable easy debugging or local changes by experts.

3.5 Loadable parameter

It is important to provide advanced users the possibility to alter the behavior of the application. This mechanic is implemented via loadable parameters defined in a JSON file.

The user provides a JSON file (Listing 3.2) in the root directory of the application. This file is loaded and processed by every configuration Bean class via the *getLoaded()* function (Listing 3.1). In this example the variable name is overwritten with "p2" and the other values stay at default.

Listing 3.1: PersonConfig.java

```

1 public class PersonConfig {
2     public String name = "p1";
3     public Integer age = 20;
4     public Float size = 20.0f;
5
6     static private PersonConfig defaultValues = new
7         PersonConfig();
8     static public PersonConfig getLoaded() {
9         return LoadableConfig.getConfig(defaultValues);
10    }

```

Listing 3.2: config.json

```

1 {
2     "name"    : "p2" ,
3     "age"    : null
4 }

```

static public <T> T getConfig(T defaultValue) is implemented with reflection (Section 2.2) and generics (Section 2.3) to be able to use it with arbitrary classes.

Gson (Section 2.4) takes the class of the default values Bean and the JSON file as shown in Figure 3.15. Depending on loading success, either the remaining values (null values) are replaced by default values, or all default values are taken. The configuration Bean is only loaded once and managed by the *LoadableConfig* class to improve performance.

Once implemented, this system is very easy and comfortable to use. Additionally, it is easily possible to upgrade old configuration files since the only difference to the old source code is the *getLoaded()* function.

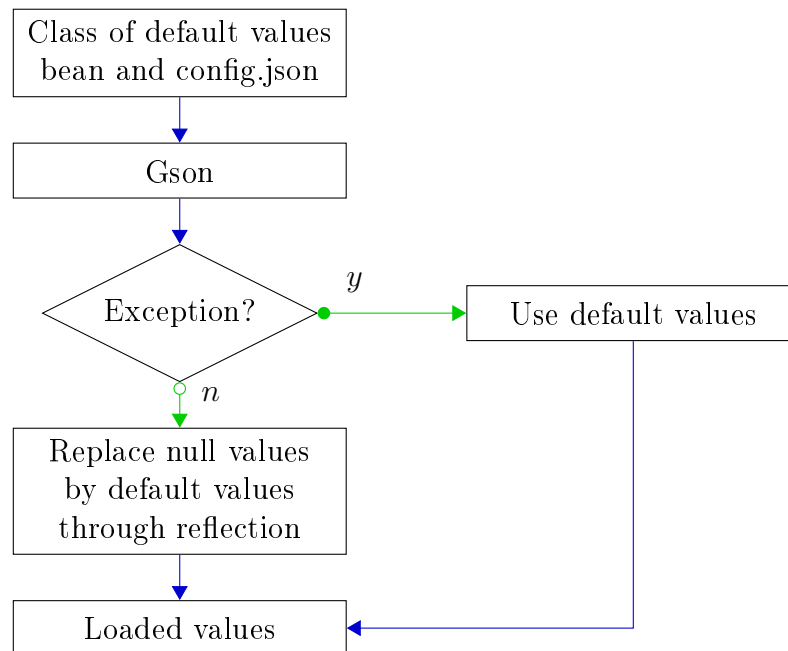


Figure 3.15: Load arbitrary configuration Bean from JSON file with Gson and default values

3.6 Additional features

This section shortly describes additionally implemented features. Although these do not represent a big part of the project, they are very important for the user experience and should be mentioned. Therefore, they are not explained in full detail. The following features are mainly concerned with usability.

History system This system tracks every modification of the test plan to enable the user to step back and forward in the modification history. A snapshot of the whole database is taken on each modification. The snapshot is created by serializing and deserializing the whole database to remove the need of a clone method. All snapshots are stored in an ordered list and the current entry index represents the current state.

The graphical user interface implements a reload method with the composite pattern [24]. This means every object will reload itself and call a reload method on all of its children. The reload method will build the corresponding object according to the database content.

Therefore, it is possible to easily walk backward and forward in the history by loading the corresponding snapshot and successively calling the reload function on the root graphical user interface object.

Different layouts Due to different layout requirements caused by the different

combinations of the displayed areas, the user is able to switch between a horizontal and vertical layout as shown in Figure 5.4 and Figure 5.5 of Section 5.1.

Shortcuts Shortcuts are an important feature of graphical user interfaces since they increase productivity. Furthermore, all popular graphical user interfaces are providing shortcuts. Therefore, shortcuts have to be provided to improve intuitive and fast handling by the user. All available shortcuts are visualized in the menu as shown in Figure 3.16.

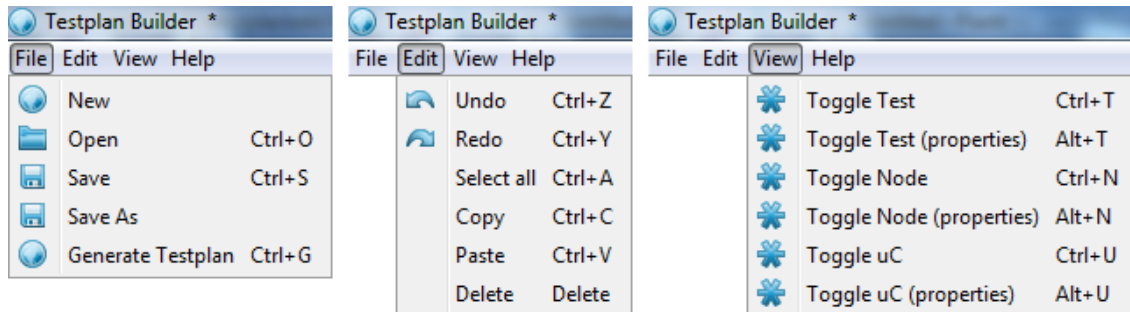


Figure 3.16: Shortcuts

FSM transition indication Possible transitions triggered by the selected states are indicated as shown in Figure 3.17. The *sendEvent('start')* function of the selected state triggers all *start* transitions on the child FSM. Therefore, these transitions are highlighted to improve usability.

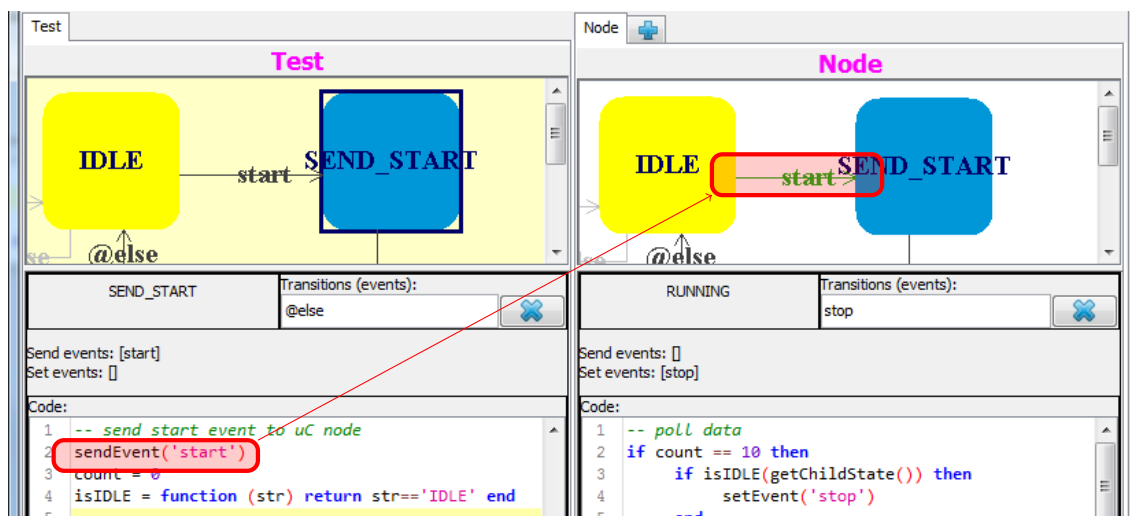


Figure 3.17: Transition indication

Error to location jump A double click on an error in the error list under the Lua input area results in a jump to the corresponding line. This increases the speed of debugging significantly since the user does not need to scroll through the Lua code anymore.

Mouse wheel navigation The mouse wheel is able to trigger some navigation features.

- Mouse wheel scrolls vertically in the FSM
- *SHIFT* + mouse wheel scrolls horizontally in the FSM
- *STRG* + mouse wheel zooms in or out of the FSM

Automatic build and deployment An easy way to build and deploy the application was requested. Therefore a build system is required which is able to build the whole project from the command line. Gradle [25] was chosen to be this build system after a comparison between Ant [26], Maven [27] and Gradle, which represent the most popular systems. After providing a build script, it is possible to build the application and automatically upload it to a web server available to all users.

Git version exposure It is important to expose the Git version of the application to the user for error reports and debugging. This is done with an information dialog which appears if the user opens the *About* menu. This version is generated by a Gradle [25] script which creates a resource file with this string as single content. Gradle was chosen because it is already used for the automatic build process and therefore this way of version exposure fits very well into the current build process. The generated resource file is deployed within the application binary. The string is read from this file during run time to provide the exact Git version to the application. Thus, it is possible to add this resource file to the Git ignore list and therefore not change the Git version in the moment the version is provided to the project.

Chapter 4

Test plan verification

Errors will occur in the test plan since it is generated by humans. Therefore, the verification of the test plan represents a very important part of the MoPS work flow. The test plan is either available as file (see Section 2.5) if used from the standalone checker (see Section 4.5), or already in a suitable data structure if used from the *Test-plan Builder* (see Chapter 3). In the first case the file has to be parsed to a suitable data structure to verify the right format of the test plan file before checking. This is done with Gson (see Section 2.4). After availability of the check able data structure the verification is performed as seen in Figure 4.1.

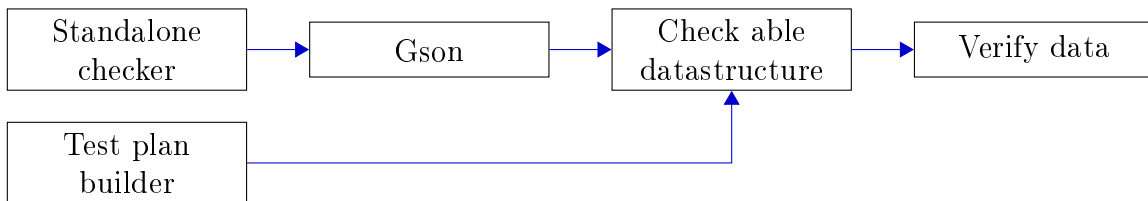
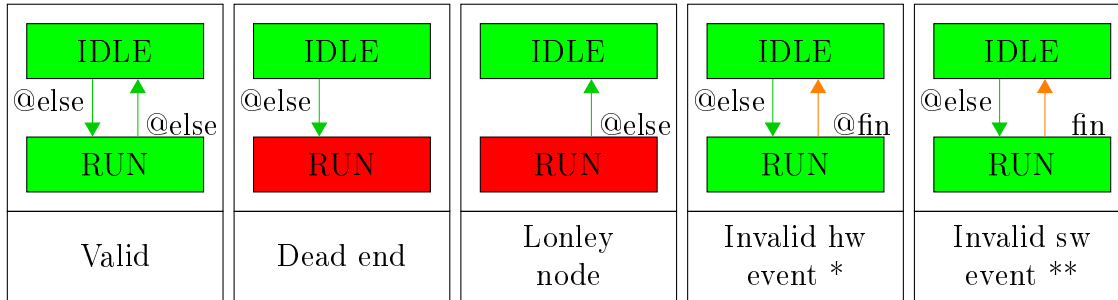


Figure 4.1: Data transformation for checker

Multiple checks are performed on the data. Each of the following sections will describe a check in detail.

4.1 FSM

The test plan contains FSMs as already known from Section 2.5. For detailed information about them please refer to Section 2.6. Four errors are able to occur in FSMs as shown in Figure 4.2.



* If not defined in EDS file

** If not used by Lua code

Figure 4.2: FSM errors

4.1.1 Dead end nodes

These are nodes that do not have an exit transition. According to the MoPS specification, a path is only valid if it is able to end in the initial state *IDLE*. Therefore, nodes which are not able to reach the *IDLE* state are erroneous nodes called “dead end nodes”.

The simplest case is shown in Figure 4.2. It is enough to check for outgoing transitions to detect this case. For nodes which have transitions but do not reach the *IDLE* node again (e.g. *IDLE* -> A, A -> B, B -> A) an algorithm is needed. Basically, this algorithm is a breadth-first search algorithm which is common in trees. Since this is a graph with possible loops there is one little difference – nodes do not need to be visited twice during the search. The “dead end” error is detected if the *IDLE* node is not visited.

4.1.2 Lonely nodes

Lonely nodes are nodes with no visitors. If a node does not get any visitors this state is impossible to occur. This is either a design failure of the test plan or a transition is missing. It is very easy to detect this error, just a check for incoming transitions is needed. It is detected in the case of no incoming transition.

4.1.3 Hardware event

Hardware events are events triggered by the hardware and have the prefix *@* by convention. There is a list of possible events defined in the EDS file. The oven plan is used to get the corresponding EDS file for a FSM. Additional to the defined events there is a special event called *@else*. It is triggered after the Lua code of a node is finished. After retrieving the valid hardware events it is possible to check these against the used hardware events.

An error is detected if a hardware event is not on this list of valid events.

4.1.4 Software event

Software events are events triggered in the Lua code through function calls. There are two possible functions to trigger software events:

setEvent(name) Triggers a software event in the own FSM. Since this triggers in the own FSM, the node in which this is used has to have a corresponding transition.

sendEvent(name) Triggers a software event in a child FSM. Since a child FSM is needed usage is only possible in a *test* or *node* FSM.

All valid software events have to be gathered to verify the software events in the FSM. This is done by getting all triggered software events from the corresponding parent FSM which is defined through the oven plan. There are more valid events per transition since there is the *setEvent(name)* which is able to trigger an event in the own FSM. The Lua code of the origin node from the transition has to be parsed for it. Found triggered events are added to the valid list. This list can now be checked against the transition software event and if it is not present an error is detected.

4.2 Lua

It is necessary to have valid full scripts to be able to check the Lua code. Therefore, it is not possible to just check the Lua code of every node on its own since variables can be declared in node *INIT* and used in node *RUN* as shown in Figure 4.3. In this case the code in node *RUN* alone would not be valid since the used variable has not been initialized. The whole script (*INIT* + *RUN*) would be valid.

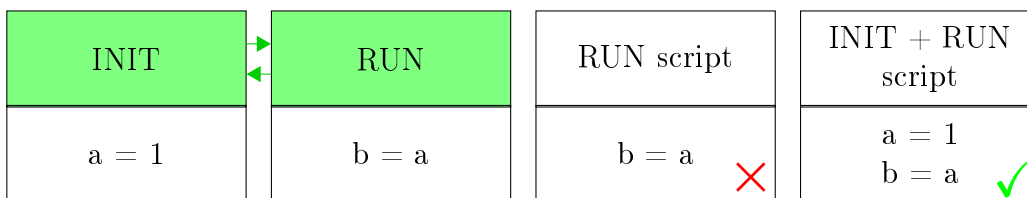


Figure 4.3: Script combination

4.2.1 Script generation

This consideration results in the need of a generation method of all possible scripts. In other words: Generate all possible paths in the FSM with start and end in node *INIT* (which is the start node). This is not a trivial problem because loops generate an infinite amount of paths in the graph. Therefore two questions arise.

How often should loops be executed and how to ensure there is no error in the $n+1$ time. Luckily we do not need to consider such problems since redefinition in Lua is valid and only a static analysis is applied. Because of the static analysis values of variables are not considered and therefore a second run of a code part does not change anything for the verification.

This results in a new definition of the path generation problem: Generate all possible paths in the FSM with start and end in node *INIT* by visiting each node only once per path. This redefinition eases the problem significantly and results in a simple algorithm. If this algorithm should not meet the requirements in the future, it is easily replaced since it is possible to provide an own generator class to the *Test-plan Checker*.

After knowing the path it is easy to generate all scripts by just appending the content of the nodes of one path. But that is not all. Some additional code is necessary before the actual script is verified since the Lua code has access to the MoPS-CORE and SAM API. The API definitions are loaded from a web server as mentioned in Section 3.4 and added as compatible Lua code in front of the checked script. Only empty dummy functions without return values are needed since Lua is not a type safe language and this is only a static verification.

4.2.2 Code verification framework

The reason why a proprietary implementation is needed and different solution approaches are discussed in Section 2.7. For better maintainability and re-usability, the verification is split in project specific checks and general Lua checks. This has the advantage that the general Lua checking part can be easily replaced or extended by future Lua checkers which verify more than this implementation. Checked scripts are generated as general valid Lua scripts as described above for further advantages of this idea. Additionally many project specific checks are performed as general Lua checks by adding additional source code to the scripts. e.g. the MoPS-CORE and SAM API as described in the above *script generation* part.

The implementation of the verification is split into several layers to ensure maintainability and future improvement of the verification. As shown in Table 4.1 the first step is to find errors. These errors are converted to *LuaErrorTokens* which contain the error and location in the script. Afterwards the *LuaErrorTokens* are converted to *FSMErrorTokens* via line mapping which contain the FSM path, the node and location in this node. These updated tokens are exposed and used by the *Test-plan Builder* (see Chapter 3) and standalone checker (see Section 4.5).

It is only necessary to generate *LuaErrorTokens* due to this layered implementation and the rest is done by the implemented framework. As stated above it is easily possible to use an external Lua script checker. The only additional implementation is the conversation of the new Lua checker errors to *LuaErrorTokens* of this framework. The next section will describe how to find errors.

<i>Test-plan Builder</i>	Standalone checker
FSMErrorToken	
LuaErrorToken	
Error	

Table 4.1: Code verification layers

4.2.3 Code verification

The first error type to handle in every language are syntax errors. For this a parser is needed. This parser will be also used for other error detection like uninitialized variables. Therefore, this parser should provide a simple way to travel the source code or to generate an AST. An AST also provides a convenient way to travel the source code to find errors and respected programs like Lint [28] use the same approach. Optionally it would be helpful to execute Lua with this application to implement a simple simulator which should be usable in Java. Additionally the application has to return the location of the parsing error. Thus the application has to meet following requirements:

- Usable in Java
- Travel code or generate an AST
- Location of parsing error
- Maintained software
- Support of latest Lua version
- Execute Lua code (optional)

Only Java applications are considered since the usage in Java is a requirement within this project. The following applications are available:

Kahlua2[29] It is a virtual machine for Lua and does not provide any possibility to travel the Lua code or generate an AST by itself. LuaJ has to be provided, to enable this feature. Since LuaJ is on the list of considered applications Kahlua2 is not suitable to solve the required tasks. Furthermore, it has not been maintained since 2013.

Mochalua[30] It is a virtual machine like kahlua2. It also provides no ability to travel the code or build a AST. Furthermore, it has not been maintained since 2008. Therefore, this is also no suitable application for the needed tasks.

LuaJ [31] Maintained interpreter which provides a convenient way to travel the Lua source code. It does not only know the exact location of the traveled

tokens, but also returns the exact location of parsing errors (in most cases). Furthermore it supports the latest Lua version and is still getting updated for future versions of Lua. Additional to the necessary requirements, it provides the optional requirements. LuaJ implements a virtual machine in Java which is able to execute Lua code.

JNLua[32] It is a virtual machine and requires the installation of the JNLua Native Library on the running operating system. This is unreasonable for a productive tool. People want to get the Java application and be able to run it without any further setup. Moreover, there is no system provided for code traveling.

LuaJ was chosen to do the job after comparison of available applications because it meets all needed and optional requirements. It provides an useful way to travel the code and implements the possibility to execute Lua code. Thus, a simple simulator (see Section 4.4) has also been implemented to perform additional checks.

LuaJ provides a very convenient way to travel the source code, such as traveling an AST without building an AST first. It implements a visitor framework. To use this, it is needed to create an inherited class, implement a visit method for every needed syntax element and pass this class to the parser. Visited syntax elements are shown in Table 4.2. A visitor is implemented for every error which needs to be detected. Listing 4.1 is a sample visitor implementation to detect empty code blocks.

Listing 4.1: EmptyCodeBlock.java

```

1 public class EmptyCodeBlock extends Visitor {
2     @Override
3     public void visit(Block block) {
4         super.visit(block);
5         if (block.stats.size() == 0) {
6             // Empty code block detected!
7         }
8     }
9 }

```

The EmptyCodeBlock class is inherited from the Visitor class to be able to use it with the parser. Every occurrence of a Block will trigger this visit method. An example for a Block is the content of *if* or *while*. The call of the *super.visit(block)* method is very important since the parser would stop if it is missing.

All Lua specific errors are recognized with this system. One visitor is implemented for each error to recognize. Errors and how they are found are described in the next two sections. The programming strategy at this part of the project is test driven programming. As mentioned in Section 2.1 there is one test class for every implemented visitor. These test classes define the valid and invalid code for every visitor. Thus, the valid code structures are defined by the test cases and the visitors are implemented correctly if all test cases pass.

Chunk	Block	Stat.Assign	Stat.Break
Stat.FuncCallStat	Stat.FuncDef	Stat.GenericFor	Stat.IfThenElse
Stat.LocalAssign	Stat.LocalFuncDef	Stat.NumericFor	Stat.RepeatUntil
Stat.Return	Stat.WhileDo	FuncBody	FuncArgs
TableField	Exp.AnonFuncDef	Exp.BinopExp	Exp.Constant
Exp.FieldExp	Exp.FuncCall	Exp.IndexExp	Exp.MethodCall
Exp.NameExp	Exp.ParensExp	Exp.UnopExp	Exp.VarargsExp
ParList	TableConstructor	Name	String
NameScope	Stat.Goto	Stat.Label	

Table 4.2: Syntax elements visited by LuaJ

4.2.4 General Lua errors

This section describes the detection of general Lua errors. Some of the listed error detection methods are also used to find project specific errors described in the next section. This is because the focus was on finding project specific Lua errors in the most general way. Every listed error will provide an example of the *Test-plan Builder*. This example uses already the *Test-plan Checker* and the described method.

- Always the same (and unreachable code)

Statements shown in Table 4.3 are considered as always the same. In addition to the listed statements, a statement is also detected if a is the same constant for all a in one statement (e.g. $a == a$ is the same as $2 == 2$). Since these statements result in the same values every time this also results in unreachable code at e.g. *if* statements.

The detection of these errors is very simple. The *visit(Exp.BinopExp exp)* method has to be overridden. This method visits all statements shown in Table 4.3. At this point there is only one step left: The check if *Exp.BinopExp exp* matches any of the listed statements.

Figure 4.4 shows some examples in action.

- Bad coding practice

Bad coding practice is considered code which is allowed by the language, but should not be used since most of these practices result in error prone source code or maintainability problems. One of these practices is the usage of the *goto* statement. Therefore, it is prohibited in the test plan. To recognize this statement it is enough to override the *visit(Stat.Goto gotostat)* method. No further implementation is needed for this.

Figure 4.5 shows an example of this error in action. The important line 2 is skipped because of the *goto* statement. Therefore a would not be initialized.

$a + 0$	always a
$0 + a$	always a
$a - 0$	always a
$a == a$	always true
$a < a$	always false
$a > a$	always false
$a >= a$	always true
$a <= a$	always true
$a \neq a$	always false

$a / 0$	always infinity
$a / 1$	always a
$0 / a$	always 0
$a * 0$	always 0
$a * 1$	always a
$0 * a$	always 0
$1 * a$	always a

Table 4.3: Always the same

```

Code:
1 a = 1
2 if a == a then a = a * 0 end
3 if a >= a then a = 1 * a end
4 if a == a then a = a + 0 end
5 if 2 == 2 then a = a / 0 end
6

```

Hide error list

2: Always the same (Path: IDLE -> ERROR)
2: Always the same (Path: IDLE -> ERROR)
3: Always the same (Path: IDLE -> ERROR)
3: Always the same (Path: IDLE -> ERROR)
4: Always the same (Path: IDLE -> ERROR)
4: Always the same (Path: IDLE -> ERROR)
5: Always the same (Path: IDLE -> ERROR)
5: Always the same (Path: IDLE -> ERROR)

Figure 4.4: Always the same

- Empty code block

This recognition is similar to the bad coding practice. It indicates source code the user forgot to write. As shown in Listing 4.1 it is only needed to override the *visit(Block block)* method. Furthermore a check for the number of statements in the block is required to recognize empty code blocks. Examples for empty code blocks would be empty loops, *if*-blocks or functions.

Figure 4.6 shows an example of this error in action. In line 2 an empty block is detected because someone forgot to implement this function.

- Undefined functions

Uninitialized functions are a little bit more complicated to detect. The detection is split into two parts since a function has to be defined and afterwards

```
Code:
1 goto done
2 a = 2
3 ::done::
4 b = a
5
```

Hide error list

1: GOTO statement is not allowed. (Path: IDLE -> ERROR)

Figure 4.5: Bad coding practice

```
Code:
1 function myFunction(),
2
3 end
4 myFunction()
5
```

Hide error list

1: Empty block (Path: IDLE -> ERROR)

Figure 4.6: Empty block

used in the right scope.

The first part is the function definition. Multiple visitor methods have to be used for this to add available functions to a function list. This function list will be accessed later to verify the usage of a function.

visit(Stat.FuncDef stat) This visits a normal global function definition which is available in every scope. An example would be *function test() end*. If this occurs a new function with name and parameter count is added to the possible function list.

visit(Stat.LocalFuncDef stat) This visits a normal local function definition which is available only in this scope. An example would be *local function test() end*. If this occurs a new function with name, parameter count and corresponding scope is added to the possible function list.

visit(Stat.Assign stat) In Lua it is possible to define functions with an assignment. An example would be *test = function() end*. This visit method does not necessarily visit a function assignment. Thus only a possible function is added if a function is assigned. The function is added with name and parameter count to the possible function list since this is a global assignment.

visit(Stat.LocalAssign stat) This is the local counter part to the *visit(Stat.Assign stat)* method. An example would be *local test = function() end*. The function is added with name, parameter count and scope to the possible function list.

The *Exp.AnonFuncDef* visitor is missing on purpose because it visits the function definition in an assignment. There would be no name and scope information if the *Exp.AnonFuncDef* visitor is used instead of the assignment visitors.

The second part is the check if used functions are defined. There is not a lot of work left since a list of possible functions is already generated in part one. The method *visit(Exp.FuncCall exp)* has to be overridden to detect used functions. The function is checked for valid usage depending on the *Exp.FuncCall*. Only a function in the form of *test()* is checked since other types (e.g. *a.test()*) are handled in the next checker. A function is valid if it matches an entry in the possible function list with name, parameter count and scope.

It is important to mention that part one and two do not work in sequence. Since the methods are implemented in the same visitor a function has to be defined before usage. If a function is defined after usage the visitor will detect an error.

Figure 4.7 shows an example of this error in action. In line 4 a typo causes a call to an undefined function.

```
Code:
1 function myFunction(a)
2     return a * 3
3 end
4 myfunction(2)
5
```

Hide error list

4: Function not defined. (Path: IDLE -> ERROR)

Figure 4.7: Undefined functions

- Undefined classes & methods

It is not possible to write Object-oriented programming (OOP) code natively in Lua. But with some workarounds and table tricks it can be simulated. To be able to check classes and methods a valid syntax has to be created. For this purpose a class function shown in Listing A.1 is defined. With this trick it is now possible to define classes with *ai = class(function(a, name)end)* for the usage like *obj = ai('name')*. Furthermore it is possible to define methods with *function ai:print() end* and use them like *obj:print()*. This is now the only

valid OOP syntax in the test plan which is checked and it is defined in this way because the MoPS-CORE API is accessible with this syntax. To verify the right usage there are four steps required.

- The first one is the availability of the actual class. The visitors for local and global assignments are overridden. The implementation adds a new class to the valid class list if a class assignment is detected.
- The second step is the availability of the class methods. The method *visit(Stat.FuncDef stat)* is overridden. The implementation checks if this is a method definition. If it is a method definition it will either add a possible function with name and parameter count to the class, or it will generate an error token if the class is not already in the valid class list.
- Variables are mapped to classes in the third step. On every assignment a variable is cleared from the class mapped to it. If the assigned value is the simulated constructor function, either the class is mapped to it, or an error token is generated if the class is not in the valid class list.
- The last step verifies the right usage of an object. For this the *visit(Exp.MethodCall exp)* is overridden. The implementation checks if the variable has a class mapped to it. An error token is generated if no class was found. Otherwise the method is compared to the method definitions in this class.

Additional to the usage check it also checks for double assignment of a class. It is not possible e.g. to assign something to C after creating the class C .

It is important to mention that these parts do not work in sequence. Since the methods are implemented in the same visitor the class and methods have to be defined before usage. If a class or method is defined after usage the visitor will detect an error.

Figure 4.8 shows an example of this error in action. It is not possible to add a method to a not defined class in line 1. In line 4 the assignment fails because the class does not exist. Line 5 tries to call a method, but the variable is not a class object.

- Undefined tables & table functions

A table is defined with $t = \{\}$ and function are added with *function t.clear() end*. It is possible to use the function like *t.clear()*. This is now the only valid table function syntax in the test plan which is checked and it is defined in this way because the MoPS-CORE API is accessible with this syntax. To verify the right usage there are three steps required.

- The first one is the availability of the actual table. The visitors for local and global assignments are overridden. The implementation adds a new table to the valid table list if the assignment of $\{\}$ is detected.

```

Code:
1  function Aclass:myFunction()
2  return 1
3  end
4  aclass = Aclass()
5  aclass:myFunction()
6

```

Hide error list

```

1: Table was not defined previously. (Path: IDLE -> ERROR)
4: Function not defined. (Path: IDLE -> ERROR)
5: Variable is no class object. (Path: IDLE -> ERROR)

```

Figure 4.8: Undefined classes & methods

- The second step is the availability of the table functions. The method *visit(Stat.FuncDef stat)* is overridden. The implementation checks if this is a function definition. If it is a function definition it will either add a possible function with name and parameter count to the table, or it will generate an error token if the table is not already in the valid table list.
- The last step verifies the right usage of the table. For this the *visit(Exp.FuncCall exp)* is overridden. The implementation compares the function to the function definitions in this table.

Additional to the usage check it also checks for double assignment of a table. It is not possible e.g. to assign something to *t* after creating the table *t*.

It is important to mention that these parts do not work in sequence. Since the methods are implemented in the same visitor the table and functions have to be defined before usage. If a table or function is defined after usage the visitor will detect an error.

Figure 4.9 shows an example of this error in action. In line 3 a typo tries to add a function to a not defined table and line 4 causes a call to an undefined function on the table *t*.

- Infinite loops

Infinite loops are detected by inspecting the statement in the condition. For this the *visit(Stat.RepeatUntil stat)* and *visit(Stat.WhileDo stat)* methods have to be overridden. Both result in the same function which accepts condition statements and body statements. There are two cases of infinity loops.

The first one occurs if the condition is a *true* constant. Conditions which always evaluate to *true* are already recognized by another check.

The second case occurs if no variable of the condition is modified within the body. Therefore the body is searched for an assignment to one of the variables in the condition. If none is found an error token is generated.

```
Code:
1 t = {}
2 function t.getOne() return 1 end
3 function g.getTwo() return 2 end
4 t.myFunction()
5 t.getOne()
6
```

Hide error list

3: Table was not defined previously. (Path: IDLE -> ERROR)
4: Function not defined. (Path: IDLE -> ERROR)

Figure 4.9: Undefined tables & table functions

Figure 4.10 shows an example of this error in action. There was a typo (*i* assignment) in line 4 and therefore *i* is never modified in the while loop.

```
Code:
1 local i = 1
2 while i < 5 do
3     print(i)
4     j = i + 1
5 end
6
```

Hide error list

2: Infinite loop (not all condition vars are modified in the block) (Path: IDLE -> ERROR)

Figure 4.10: Infinity loop

- Uninitialized variables

Variables should be accessed after initialization only. The fast implementation would be to override the *visit(Exp.NameExp stat)*, resolve the variable for it and handle it depending on read or write access. But an implementation is needed which is able to distinguish between read or write access since there is no distinction between these two access types in this method. For this the verification is divided into two parts. The first is the variable initialization part and the second is the check on usage part.

Two methods has to be overridden to perform part one: *visit(Stat.LocalAssign stat)* and *visit(Stat.Assign stat)*. Luckily variables are already handled nicely in Luaj. There is a distinction between local and global variables in the parser. Therefore it is only needed to add an *isAssigned* member to the variable class and set it to true.

The second part is as easy as the first one but needs a little bit more implementation work. Every visitor method in which read access is able to occur

is overridden. The read access statements are checked for variables in the implementation . If there are variables found, they are checked for the new *isAssigned* member.

Figure 4.11 shows an example of this error in action. Variable *a* on line 5 was never assigned. Therefore it is uninitialized. This example showcases, that the recognition is scope sensitive. Additionally to scopes it also considers loop variables.

```
Code:
1 function double(b)
2     local a = 2 * b
3     return a
4 end
5 b = a
6
```

Hide error list

5: Uninitialized variable (Path: IDLE -> ERROR)

Figure 4.11: Uninitialized variables

4.2.5 Project specific Lua errors

Project specific errors are only able to occur in relation to the MoPS project. Nevertheless the focus of the following detection methods is on finding these errors in the most general way. Therefore the following methods will partly use the previously implemented detection methods. This results in the fact, that the successful detection of project specific errors does highly depend on the successful detection of general Lua errors.

- MoPS-CORE API access

It is possible to access the MoPS-CORE API within Lua in the μC FSM. There are two access types:

Modules MoPS-CORE modules are comparable to singleton classes with static access. They are defined in the MoPS-CORE documentation and updated in the *Test-plan Builder* and *Test-plan Checker* by parsing the documentation available on a web server. The API is implemented in *C* in MoPS-CORE and the documentation is generated with LDoc [23].

Examples for this API are *uc.clear()* and *time.print()*.

Classes Classes are normal classes. They are defined and updated like modules. Other than modules, class instances have to be created in the Lua script.

An examples for this API is *scan0 = ai("scan0"); scan0:setChn(0, 1)*.

Prepended code for the tested Lua scripts is generated to verify the API access in the most general way. Therefore dummy Lua code is created with the same syntax defined in the previous visitors. It is important that this generated code is not only valid for this checker implementation but also for general Lua. This provides the possibility to add external Lua checkers in the future if available.

The used API content is generated from EDS files explained in Section 2.8 since the available APIs are able to evolve over time. Furthermore not every hardware can use all modules/classes defined in the corresponding API documentation. The available modules/classes are defined in the EDS file. Therefore it is necessary to generate a common API content (intersection) for every hardware using the same Lua script. The prepended code for the Lua script is generated from the common API content. Afterwards, the MoPS-CORE API errors are detected by the *Undefined classes & methods* and *Undefined tables & table functions* checker.

Figure 4.12 shows an example of this error in action. The MoPS-CORE *ai* class is assigned to the variable *a* on line 1. Afterwards various method accesses are tried. Most of them fail because they are not defined. Afterwards the MoPS-CORE *time* module is accessed on line 6 and 7. The second one fails because this function is not defined in this module.

```
Code:
1 a = ai("ai0")
2 a:read2(2)
3 a:read()
4 a:read(2)
5 a:read(2, 2)
6 time.print()
7 time.print2()
8
```

Hide error list

```
2: Method not defined. (Path: IDLE -> ERROR)
3: Method not defined. (Path: IDLE -> ERROR)
5: Method not defined. (Path: IDLE -> ERROR)
7: Function not defined. (Path: IDLE -> ERROR)
```

Figure 4.12: MoPS-CORE API access

- SAM API access

It is possible to access the SAM API within Lua. This API is implemented in LabVIEW and the documentation is provided in the same way like the MoPS-CORE API. The only different to the MoPS-CORE API is, that the

SAM API provides functions for the *test* and *node* FSM instead of classes and modules. Therefore the only access type is:

Global functions They are predefined functions which are accessible via Lua. They are defined in the SAM documentation and updated in the *Test-plan Builder* and *Test-plan Checker* by parsing the documentation available on a web server. The API is implemented in *LabVIEW* on SAM and the documentation is generated with LDoc [23].

Examples for this API are *sendEvent(event)* and *print(text)*.

Prepended code is added to the tested Lua scripts as for the MoPS-CORE API check. It is important, that this code represents valid Lua code to enable the checker to verify the API in the most general way. Afterwards, the SAM API errors are detected by the *Undefined functions* checker. This provides the possibility to add external Lua checkers in the future if available.

The used API version, which determine all additional valid functions, is indicated by the SAM version field in the test plan.

Figure 4.13 shows an example of this error in action. A function defined by the SAM API is called on line 1. The function on line 2 does no exist.

```
Code:
1 state = currState()
2 state = currState2()
3
Hide error list
2: Function not defined. (Path: IDLE -> ERROR)
```

Figure 4.13: SAM API access

- Code in *IDLE* node

Source code other than comments is not allowed in the *IDLE* node. Only the content of the *IDLE* node is considered to recognize this. Every *visit* function has to be overridden and an error token is generated if any of them are triggered. This is the safest way to implement this detection since a regular expression check for comment tags is very error prone.

4.3 Oven plan

The oven plan is a mapping between multiple components. The available components with explanations are listed in Section 2.5.

There should be one oven plan entry for every DUT in the test machine. Unfortunately the *Test-plan Checker* does not know the DUTs used in the test machine since it has no physical access to them. This is because the test plan is built and checked before the machine is set up to run this test. Therefore this has to be checked by the test machine itself.

On the other side it is possible to check every existing entry in the oven plan. Every entry has to contain exactly one of each component described above and depending on the chosen components, other components are valid or not. This is why multiple checks are needed to verify the oven plan.

Missing values Every entry is checked for missing components. As mentioned above it is mandatory for each entry to contain a value for every component.

Double values The components *slot*, *dut* and *node* have to be unique in the oven plan. This is because *slot* is the location of a DUT and it is not possible to put multiple DUTs into the same location. *dut* maps a DUT to this entry and if there would be more DUTs with the same name it would not be possible to assign an entry unambiguously. *node* is the IP address of the μ C and only one unique IP address is allowed in a network as commonly known.

μ CTarget and application module values Multiple configuration files are loaded from the network as described in Section 3.4. One of these files contains a list with possible μ CTarget and application module values. Therefore this has to be checked against this list. Further checks regarding these values are performed by the test machine since it has physical access to the used hardware and the *Test-plan Checker* has not.

Node values (IP) Multiple files are loaded from the network as already mentioned. These files also contain EDS files (see Section 2.8) which contain useful data for the oven plan check. To understand this check it is important to know, that EDS files are the description of the μ C. It includes among others the IP address of the μ C, which is used here, and the MoPS software running on it, which is used in the Lua check (see Section 4.2). To verify the node value it is enough to find an EDS file which contains the IP address. The check if this is the right μ C has to be performed by the test machine since it has physical access to the used hardware and the *Test-plan Checker* has not.

FSM usage The components *nodeFSM* and *ucFSM* are defining the used FSMs for the corresponding DUT. A value is valid if the corresponding FSM exists and has the right type. *nodeFSM* has to have the type *Node* and *ucFSM* has to have the type *uC*. This is also the place where the *testFSM* from the test plan is checked for the right type, which has to be *Test*.

Additionally the *Test-plan Checker* searches for unused FSMs which also indicates a wrong oven plan.

4.4 Simple simulator

The simple simulator is an experimental feature of the *Test-plan Checker*. Therefore, it has to be enabled by the user with an configuration file. The idea is to improve the *Test-plan Checker* by extending the static checking behavior by a simple dynamic check. This would enable the checker to reliable detect run time errors like infinite loops or *null* table access.

It is enough to execute the tested Lua script and observe a successful stop of the script in the simplest case. LuaJ is used to perform this task. It implements a virtual machine in Java which returns the exact error location with detailed information if an error occurs.

Sadly the implementation of a working simulator is not that easy since the Lua script has access to the MoPS-CORE and SAM API. This means that some methods and functions will return values depending on the hardware. It would go beyond the scope of this master thesis to cover all of these values or implement a full simulator.

Nevertheless, a simple implementation was performed. Dummy code which simulates the API is prepended to the script as already mentioned in subsection 4.2.1. This implementation assumes that the API methods and functions return the same values all the time. Therefore the scripts will have a slightly different behavior on the simulator compared to the real system. An obvious example is the following valid script which results in an infinite loop on the simulator.

```

1  ai1 = ai("ai1")
2  while ai1:read(2) < 5 do
3  end

```

The read function from line two will always return the same value on the simulator, therefore it will never leave the loop. But on the real system, this method returns different values all the time.

Another problem arises in the form of the halting problem. This problem concerns with the question if an application has a finite run time or not. It can not be solved as proven by Alan Turing [33] in 1936. Another resource to this problem is the dissertation of Peter Puschner [34] in which he presents a method for the analysis of program execution time. Raimund Kirner follows another approach by developing a whole programming language which is designed for execution time analysis in his master thesis [35]. Therefore, a practical solution has to be implemented to bypass this problem. This implementation assumes that the application will not finish anymore if it has not finished during a specified timeout. This timeout has to be specified by the user within a configuration file. The Lua script will stop and generate an error token with the current executed line if the timeout occurs. At this point the user has to evaluate if the generated error token is a real error or a false-positive.

This dynamic check is disabled by default since the simple simulator produces false-positives and the handling of this simulator needs additional knowledge and

```

1 $ java -jar checker.jar -V -s tp.json
2 v1.0-5-gc5e16fe
3 [INFO] Cache update mode: SERIAL
4 [MoPS Module Manager] Cache is getting synchronized.
5 [EDS Manager] Cache is getting synchronized.
6 [Config File Manager] Cache is getting synchronized.
7 [INFO] Loading testplan
8 [INFO] Checking testplan
9 ##### TOKENS #####
10 [GRAPH] BUG NODE (uC):WARNING:0:0:0:0:Invalid path:Invalid path found:
    IDLE -> RUN -> STOP -> BUG NODE
11 [GRAPH] NOT USED NODE (uC):WARNING:0:0:0:0:Lonely node:Unused node
    found
12 [GRAPH] STOP (uC):ERROR:4:1:4:9:ai:Variable is no class object. (Path:
    IDLE -> RUN -> STOP)
13 [OVENPLAN] FSM Node is not used in the ovenplan.
14 [OVENPLAN] FSM uC is not used in the ovenplan.
15 RETURN CODE: -2

```

Figure 4.14: Standalone checker sample

configuration values. Nevertheless it provides very useful information in some cases and therefore it is provided to power users.

4.5 Standalone checker

It is not guaranteed that loaded test plans in MoPS are created with the *Test-plan Builder* application or that users do not change generated test plans with a text editor. This results in potentially erroneous test runs. To prevent this case a stand alone command line tool of the *Test-plan Checker* is provided which will be used before a test is started in MoPS. This command line tool implements the commands described in Table 4.4 exclusive some combinations mentioned in Table 4.5. Additionally the application provides return codes described in Table 4.6 for easier usage. Depending on this return code, either the test will be started, or manual investigation is needed. A sample output for a small erroneous test plan is shown in Figure 4.14.

-V, --version	Prints the version of the application.
-u, --updateCache	Updates API, EDS and network configuration caches asynchronously. New cache will be used on next run.
-c, --check	Checks the given test plan. The path to the test plan has to be provided. eg.: app --check "path/to/test plan"
-s, --serialize	Update cache and check the test plan with the new cache. The path to the test plan has to be provided. eg.: app --serialize "path/to/test plan"
-e, --eds	Use alternative EDS files for checking. The path to the directory with the EDS files has to be provided. eg.: app --eds "path/to/eds/dir"

Table 4.4: Standalone checker commands

--serialize & --check	Has no effect since --serialize already checks the test plan. Simultaneous usage could be confusing for users.
--serialize & --updateCache	Has no effect since --serialize already updates everything. Simultaneous usage could be confusing for users.

Table 4.5: Standalone checker prohibited command combinations

1	Validation succeeded but includes warnings
0	Validation succeeded
-1	Validation failed because of an unexpected error
-2	Validation failed because of found errors

Table 4.6: Standalone checker return codes

Chapter 5

Evaluation

This chapter covers the evaluation of the final implementation. The first part focuses on the graphical user interface of the *Test-plan Builder*. The second part focuses on the *Test-plan Checker* which provides error tokens to the *Test-plan Builder*.

5.1 Test-plan Builder

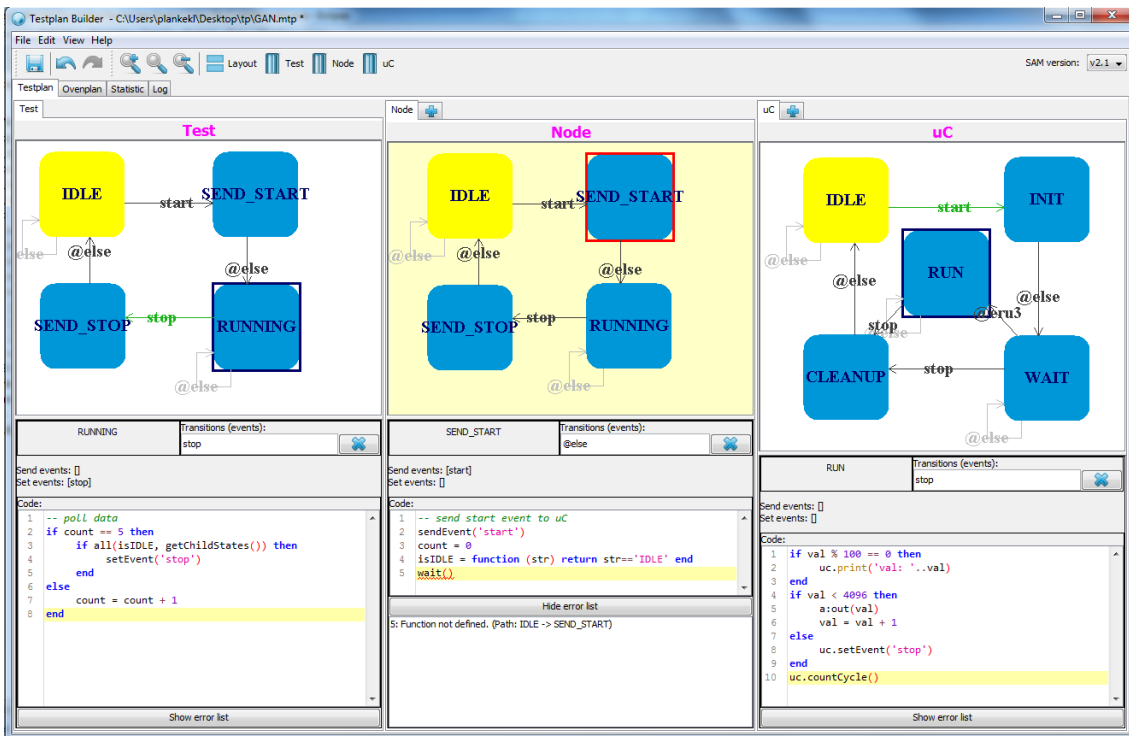


Figure 5.1: Test-plan Builder

The graphical user interface got very intuitive after several user reviews, productive

usage from a small user group during development and refactoring. Over the time it got more and more compact which provides a lot of space for the important areas as you can see in Figure 5.1.

Nevertheless it is hard to design big FSMs if only one third of the screen is available. It is getting worse on very small screens as shown in Figure 5.2. To solve this issue it is possible to hide areas as shown in Figure 5.3. Shortcuts are provided for every possible task to further improve development speed of power users as seen in the same figure.

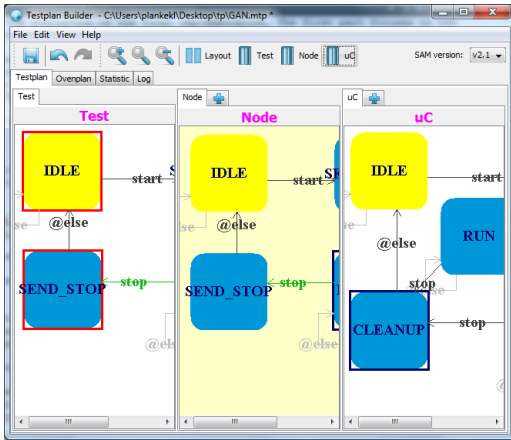


Figure 5.2: Too little space

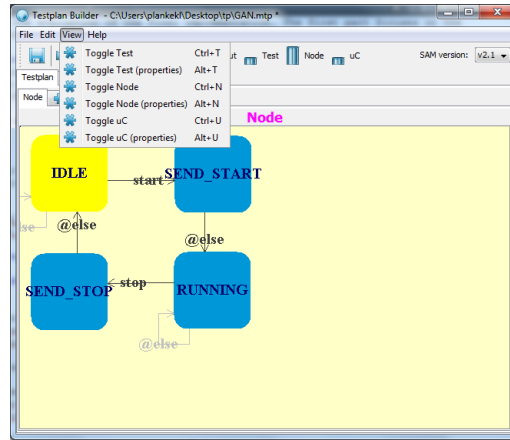


Figure 5.3: Shortcuts

Due to different layout requirements caused by the different combinations of shown areas, the user is able to switch between a horizontal and vertical layout as shown in Figure 5.4 and Figure 5.5.

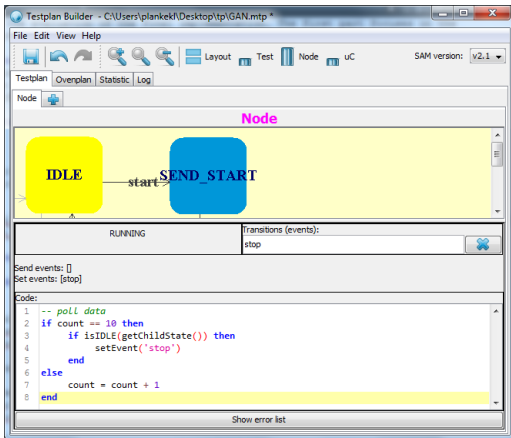


Figure 5.4: Horizontal layout

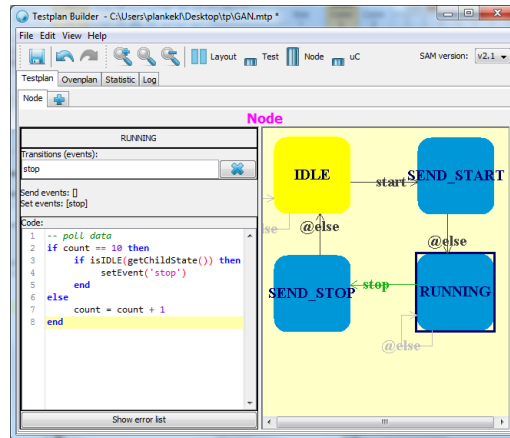


Figure 5.5: Vertical layout

The most difficult part for non computer experts is the Lua input. An auto completer is provided which contains all API access code to talk with the hardware. In all

situations where the auto completer is called, it will give detailed instructions of the possible options as seen in Figure 5.6. Furthermore it provides example code for easy Lua code writing. The shown information is always up to date since it is parsed (and cached) from the documentation server on every *Test-plan Builder* start up.

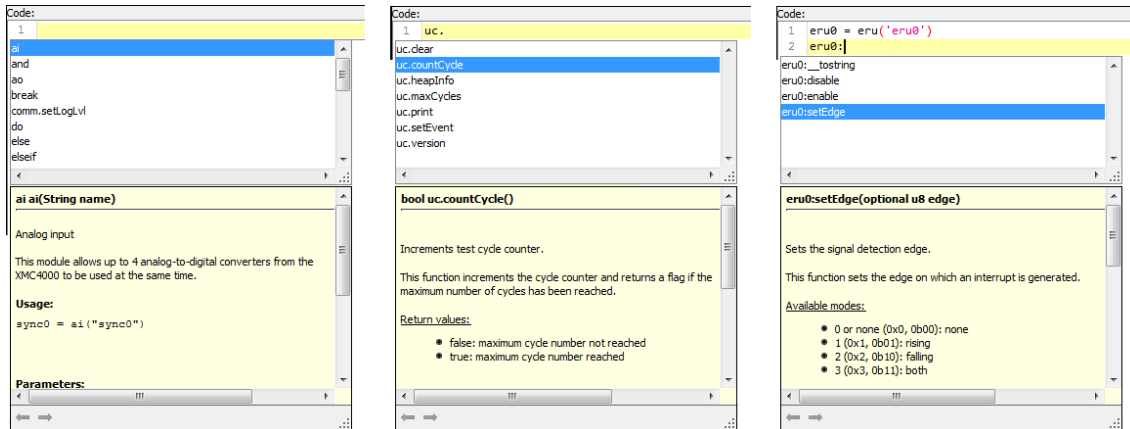


Figure 5.6: AutoComplete

Nevertheless, errors are able to occur in the Lua code. A red line will mark Lua errors and detailed error information is visible in a window below the Lua code. The FSMs are also able to contain errors – these are visualized through red marks and are also listed below the Lua code for detailed error information.

The combination of various graphical user interface layouts, programming support techniques, a history system and visual error indication provides a smooth and intuitive user experience. Generally spoken, the implementation of the *Test-plan Builder* is a huge success. Furthermore, the modular implementation of the graphical user interface and error detection (see next section) enables the MoPS system and future projects to easily check test plans for errors.

5.2 Test-plan Checker

The *Test-plan Checker* provides a very high error coverage of the test plan. There are multiple locations in the test plan where errors can occur. Thus, this evaluation is split into multiple sections for a detailed evaluation. These error sections include FSM, Lua and oven plan data. An evaluation of the experimental script simulator is provided in subsection 5.2.4 which explains why and how this feature is only usable by power users.

5.2.1 MoPS FSM data

All known possible static errors are detected by the *Test-plan Checker*. These errors do not only include real errors like malformed FSMs but also include user errors in

combination with the Lua code or hardware. Every transition has to be able to occur in the software or hardware for example.

Nevertheless, there is no guarantee that the FSMs do not end in an infinity loop during run time since this is only a static check. Some transitions do highly depend on the hardware because some of them are only triggered by the hardware during run time. The FSM is stuck in a state if the hardware do not trigger such an event.

5.2.2 Lua data

The error detection rate in the Lua scripts is pretty high. All basic user mistakes are found by the error checker. This covers all target errors defined for this thesis. There are only problems in advanced use cases where users use self defined nested tables and modules. But this is not surprising since only a static check is performed.

Nevertheless, the focus on maintainability during implementation enables the Lua checker to get easily upgraded by external checkers. As described in Section 4.2, the implementation was tested with the JUnit framework. All errors which should be found were defined in test cases which passed after implementation. Furthermore the checker is already successfully tested by all users who use the *Test-plan Builder*. The exact errors found and examples are listed in subsection 4.2.4.

5.2.3 Oven plan data

All known possible errors are detected by the *Test-plan Checker*. This data is not influenced by run time since it just describes the hardware. Therefore the oven plan data is bullet proof.

5.2.4 Experimental script simulator

The script simulator is an attempt to extend the static checks by a dynamic check. This simulator takes a script and launches it. An error token is generated with the exact location if any error occurs on run time . This is possible because all project specific functions and modules are added to the scripts as precode by the *Test-plan Checker* to generate a valid Lua script.

All possible script combinations are already generated from the *Test-plan Checker* as explained in Section 4.2. This results in the conclusion that if all of these script combinations run without errors, the whole test plan should be valid. For most scripts it does work pretty well but there are some special cases where the script simulator fails:

Detect non halting scripts This results in the halting problem [33] which is unsolvable. A practical solution is to define a script timeout to bypass this problem. If the script does not halt in this time span, it is stopped and an error token is generated with the current run time location of the script.

Handle hardware generated values The Lua code is closely tied to the hardware with an API access. In the script this API access functions are added as precode as mentioned before. These dummy function does never change their return values. Following code snipped is valid and working code on the hardware but will never leave the while loop on the simulator.

```
1 ail = ai("ail")
2 while ail:read(2) < 5 do
3 end
```

Despite these two cases the simulator provides a very useful check – if able to finish. Due to this condition it is not suitable for non power users since they have to analyze the error and decide if it is a real error or not. Therefore the simple script simulator is disabled as default and needs to be enabled by power users with the help of a configuration file.

Chapter 6

Conclusion

Two applications have been implemented during this master thesis. The *Test-plan Builder* which uses the second one – the *Test-plan Checker*. The applications were implemented in close relation to additionally provide the *Test-plan Checker* in form of a console application. This application is used for test plan verification independent from the *Test-plan Builder*. All problems regarding the user friendly test plan creation and verification described in Section 1.3 have been solved during the implementation. The verification of dynamically attached Lua chunks as well as the creation of such a system was not found in any previous project during research.

The creation process is visualized through a GUI (see Chapter 3). This GUI provides a simple and intuitive way to create FSMs – the user is able to “draw” and connect FSM states with the mouse. The Lua code input field per state is enhanced with an auto completer and code highlighting. Additionally, detected errors from the *Test-plan Checker* are visualized in the GUI, for example by displaying a red line in the Lua code. The MoPS-CORE and SAM API is provided as documentation on a TLS secured web server. This documentation is parsed, cached and provided in a usable data structure to the *Test-plan Checker* and to the auto-complete system of the *Test-plan Builder*.

The task of test plan verification was also successfully achieved. The FSM states and transitions were checked to match the MoPS FSM definition. The Lua code is verified by generating all possible paths through the FSM. Since redefinition is allowed in Lua, every node is only visited once per path. This fact hugely simplifies the generation problem and removes issues with loops and revisiting. Every generated path stands for one Lua script which has to be valid in a general Lua checker. A dedicated Lua checker is implemented by a static analysis of the code since no suitable Lua checkers are available. The MoPS-CORE and SAM API is prepended to the checked Lua scripts as valid Lua code. Therefore, the project specific parts are checked in the same way as the general Lua code. This enables developers to easily upgrade the Lua checker because it is possible to add an external general Lua checker to the framework. The verification of the oven plan and other hardware specific values is performed by fetching an electronic data sheet file. This file con-

tains all hardware relevant information and is provided through the TLS secured web server.

Outlook

Although the two implemented applications work pretty well, there is always space for improvements. In the following, some possible improvements and additional implementation ideas are covered.

The first improvement concerns the improvement of the graphical user interface. After some months of productive usage and increase of the user base, criticism about different styles of the applications and usability improvement requests is expected. These comments and ideas should be gathered in an appropriate tracking application. This will grow to a huge resource of visual and usability improvements over time.

The next improvement is to support the loading of test plan files in the *Test-plan Builder*. The test plan file is not designed to hold meta data like graphical state location because this information is not useful for the MoPS system. Therefore, the *Test-plan Builder* has its own save file (*mtp*) since this information has to be stored somewhere. It is expected that users will lose some *mtp* files over time but still have the corresponding test plan file. Therefore, the possibility to import the test plan file should be provided in the future. One interesting problem which arises on this task is the initial positioning of the FSM states.

Another GUI improvement would be the implementation of hierarchical FSMs. This means the support of sub FSMs in an FSM. The MoPS test system cannot handle this but it could be provided by the GUI. The hierarchical FSM has to be translated to a normal FSM during test plan generation. The only downside is the loss of sub FSMs if the *mtp* file is lost.

Although the Lua error detection rate is appreciably high, there are some cases where errors may not be detected. The Lua detection implementation can be extended to solve this issue. Furthermore, additional Lua verification software can be included to catch more errors if useful verification software is available. The *Test-plan Checker* was implemented in an extendable way to support both approaches.

The current implementation of the software and hardware event checks does only cover basic tests. More sophisticated tests could be added. e.g. unambiguous transitions – two different events are set, which are leaving the same node.

Last but not least, the simple experimental simulator could be improved to evolve to a full scale dynamic checker. A method to simulate hardware API values and events has to be implemented. Additionally, a better recognition for non-stopping scripts can be developed.

Index of abbreviations

API Application Programming Interface

AST Abstract Syntax Tree

DUT Device Under Test

EDS Electronic Data Sheet

FOSS Free and Open Source Software

FSM Finite-State Machine

GUI Graphical User Interface

JSON JavaScript Object Notation

KAI Kompetenzzentrum Automobil- u. Industrieelektronik GmbH

MoPS Modular Power Stress [1]

OOP Object-oriented programming

TLS Transport Layer Security

SAM Software Architecture for MoPS

Bibliography

- [1] B. Steinwender, S. Einspieler, M. Glavanovics, and W. Elmenreich, “Distributed power semiconductor stress test & measurement architecture,” in *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*. IEEE, 2013, pp. 129–134.
- [2] D. Crockford, “The application/json Media Type for JavaScript Object notation (JSON),” 2006.
- [3] D. Harel, “Statecharts: a visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987.
- [4] R. Ierusalimsky, L. H. de Figueiredo, and W. Celes, “The evolution of Lua,” in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 2007, pp. 2–1.
- [5] B. Steinwender, “Distributed Smart Controller Network for Modular Power Stress Test,” Ph.D. dissertation, University of Klagenfurt, in writing.
- [6] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [7] V. Massol and T. Husted, *Junit in action*. Manning, 2003.
- [8] I. R. Forman and N. Forman, *Java reflection in action*. Manning, 2004.
- [9] B. C. Smith, “Procedural reflection in programming languages,” Ph.D. dissertation, Massachusetts Institute of Technology, 1982.
- [10] R. Englander, *Developing Java Beans*. O’Reilly Media, Inc., 1997.
- [11] G. Bracha, “Generics in the Java programming language,” *Sun Microsystems*, pp. 1–23, 2004.
- [12] “Gson,” Aug. 2015. [Online]. Available: <https://en.wikipedia.org/wiki/Gson>
- [13] B. Steinwender, M. Glavanovics, and W. Elmenreich, “Executable Test Definition for a State Machine Driven Embedded Test Controller Module,” in *Industrial Informatics (INDIN), 2015 11th IEEE International Conference on*. IEEE, 2015.

- [14] F. Januario, L. Cordeiro, V. De Lucena, and E. De Lima Filho, “BMCLua: Verification of Lua programs in digital TV interactive applications,” in *Consumer Electronics (GCCE), 2014 IEEE 3rd Global Conference on*, Oct 2014, pp. 707–708.
- [15] P. Peti, R. Obermaisser, W. Elmenreich, and T. Losert, “An architecture supporting monitoring and configuration in real-time smart transducer networks,” in *Proceedings of the First IEEE International Conference on Sensors*, 2002, pp. 1479–1484.
- [16] M. John, “Simple example of an object drawing program in Java,” Aug. 2015. [Online]. Available: <https://sites.google.com/site/drjohnbmatthews/graphpanel>
- [17] R. Stallman and G. Joshua, *Free software, free society: Selected essays of Richard M. Stallman*. CreateSpace Independent Publishing Platform, 2009.
- [18] F. Robert, “A syntax highlighting, code folding text editor for Java Swing applications,” Aug. 2015. [Online]. Available: <https://github.com/bobbylight/RSyntaxTextArea>
- [19] Ayman, “Java EditorPane with support for Syntax Highlighting,” Aug. 2015. [Online]. Available: <https://code.google.com/p/jsyntaxpane/>
- [20] F. Robert, “A code completion library for Swing text components, with special support for RSyntaxTextArea,” Aug. 2015. [Online]. Available: <https://github.com/bobbylight/AutoComplete>
- [21] ———, “A library adding code completion and other advanced features.” Aug. 2015. [Online]. Available: <https://github.com/bobbylight/RSTALanguageSupport>
- [22] F. Buschmann, K. Henney, and D. Schimdt, *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. John Wiley & sons, 2007, vol. 5.
- [23] D. Steve, “LDoc - A Lua Documentation Tool,” Aug. 2015. [Online]. Available: <https://github.com/stevedonovan/LDoc>
- [24] D. Riehle, “Composite design patterns,” in *ACM SIGPLAN Notices*, vol. 32, no. 10. ACM, 1997, pp. 218–228.
- [25] B. Muschko, *Gradle in Action*. Manning, 2014.
- [26] S. McIntosh, B. Adams, and A. E. Hassan, “The evolution of ant build systems,” in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 42–51.

- [27] V. Massol and T. M. O'Brien, *Maven: A Developer's Notebook: A Developer's Notebook*. O'Reilly Media, Inc., 2005.
- [28] S. C. Johnson, "Lint, a C Program Checker," 1978, pp. 78–1273.
- [29] K. Kristofer, "Kahlua2," Aug. 2015. [Online]. Available: <https://github.com/krka/kahlua2>
- [30] M. Patrick, "Mochalua," Aug. 2015. [Online]. Available: <https://code.google.com/p/mochalua/>
- [31] R. Jim, "Luaj - Lua vm written in Java," Aug. 2015. [Online]. Available: <http://www.luaj.org/luaj.html>
- [32] N. Andre, "JNLua - Java Native Lua," Aug. 2015. [Online]. Available: <https://code.google.com/p/jnlua/>
- [33] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Journal of Math*, vol. 58, no. 345-363, p. 5, 1936.
- [34] P. Puschner, "Zeitanalyse von Echtzeitprogrammen," Ph.D. dissertation, Vienna University of Technology, 1993.
- [35] R. Kirner, "Integration of Static Runtime Analysis and Program Compilation," Master's thesis, Vienna University of Technology, 2000.

Appendix A

Sample source code

Listing A.1 is an implementation of the class function. It is used to implement classes which can be instantiated like in any object orientated programming language. The implementation of *Undefined classes & methods* in subsection 4.2.4 describes why this is needed.

Listing A.1: Lua class function - <http://lua-users.org/wiki/SimpleLuaClasses>

```
1 function class(base, init)
2   local c = {} — a new class instance
3   if not init and type(base) == 'function' then
4     init = base
5     base = nil
6   elseif type(base) == 'table' then — our new class is a
7     shallow copy of the base class!
8     for i, v in pairs(base) do
9       c[i] = v
10    end
11    c._base = base
12  end
13  — the class will be the metatable for all its objects,
14  — and they will look up their methods in it.
15  c.__index = c
16
17  — expose a constructor which can be called by <classname
18  >(<args>)
19  local mt = {}
20  mt.__call = function(class_tbl, ...)
21    local obj = {}
22    setmetatable(obj, c)
23    if init then
```

```
23     init(obj, ...)
24     else — make sure that any stuff from the base class is
25         initialized!
26         if base and base.init then
27             base.init(obj, ...)
28         end
29     end
30     return obj
31 end
32 c.init = init
33 c.is_a = function(self, klass)
34     local m = getmetatable(self)
35     while m do
36         if m == klass then return true end
37         m = m._base
38     end
39     return false
40 end
41 setmetatable(c, mt)
42 return c
43 end
```