

INSTITUTO TECNOLÓGICO DE AERONÁUTICA



Rubens Miguel Gomes Aguiar

**ADDING SIMULATION CAPABILITY TO SYSTEMIC
MODELS**

Final Paper
2022

Course of Aerospace Engineering

Rubens Miguel Gomes Aguiar

**ADDING SIMULATION CAPABILITY TO SYSTEMIC
MODELS**

Advisor

Prof. Dr. Christopher Shneider Cerqueira (ITA)

AEROSPACE ENGINEERING

SÃO JOSÉ DOS CAMPOS
INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Cataloging-in Publication Data
Documentation and Information Division

Miguel Gomes Aguiar, Rubens
Adding Simulation Capability to Systemic Models / Rubens Miguel Gomes Aguiar.
São José dos Campos, 2022.
56f.

Final paper (Undergraduation study) – Course of Aerospace Engineering– Instituto Tecnológico de Aeronáutica, 2022. Advisor: Prof. Dr. Christopher Shneider Cerqueira.

1. Systemic Engineering. 2. Software Engineering. 3. Capella. I. Instituto Tecnológico de Aeronáutica. II. Title.

BIBLIOGRAPHIC REFERENCE

MIGUEL GOMES AGUIAR, Rubens. **Adding Simulation Capability to Systemic Models**. 2022. 56f. Final paper (Undergraduation study) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

AUTHOR'S NAME: Rubens Miguel Gomes Aguiar
PUBLICATION TITLE: Adding Simulation Capability to Systemic Models.
PUBLICATION KIND/YEAR: Final paper (Undergraduation study) / 2022

It is granted to Instituto Tecnológico de Aeronáutica permission to reproduce copies of this final paper and to only loan or to sell copies for academic and scientific purposes. The author reserves other publication rights and no part of this final paper can be reproduced without the authorization of the author.



Rubens Miguel Gomes Aguiar
Rua H8B, Ap. 219
12.228-461 – São José dos Campos–SP

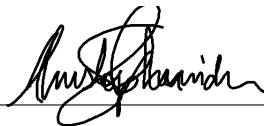
ADDING SIMULATION CAPABILITY TO SYSTEMIC MODELS

This publication was accepted like Final Work of Undergraduation Study



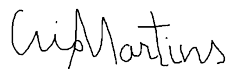
Rubens Miguel Gomes Aguiar

Author



Christopher Shneider Cerqueira (ITA)

Advisor



Prof. Dra. Cristiane Aparecida Martins
Course Coordinator of Aerospace Engineering

São José dos Campos: NOVEMBER 10, 2022.

Soli Deo Gloria.

Acknowledgments

I thank God for guiding me along the way.

To my wife, Diene Xie, for walking the path of life by my side.

To my parents, Miguel and Sara, for giving me life, supporting and encouraging me.

To my friends, Misa, Lustosa, Uchôa, Stitch, Tuzi, Borat, Carlinhos, Greg, Rafa and Robson Katz.

To Prof. Dr. Christopher Shneider Cerqueira for giving me the opportunity to do this work.

To my brothers, Rafael, Robson and Ruth for encouraging me since I was little.

To Farias Brito for giving me the opportunity to study and to Professor Wellington for introducing me to this school.

To Primeira Chance for helping me financially to study.

To Felipe and Dani's teachers for making my eyes shine through my studies.

*“Assim é que são as coisas por aqui.
As coisas vão rolando e não param.”*

— DWIGHT SCHRUTE

Resumo

Na engenharia de sistemas a metodologia model-based permite a representação de modelos em alto nível. Nesse contexto, representa-se modelos através de máquinas de estados visando integrar as diferentes partes do sistema. Por isso, surge a necessidade de efetuar a simulação das máquinas de estados de forma automatizada, para verificar a integridade dos modelos criados. Portanto, foi desenvolvida uma extensão em Python para uma das principais ferramentas de modelagem de engenharia de sistemas, conhecida por Capella, que permita a simulação de statecharts. Com isso, a ferramenta desenvolvida foi aplicada em diferentes máquinas de estados para exemplificar seu comportamento prático.

Abstract

In systems engineering, the model-based methodology allows the representation of models at a high level. In this context, models are represented through state machines in order to integrate the different parts of the system. Therefore, the need arises to perform the simulation of state machines in an automated way, to verify the integrity of the created models. Thus, a Python extension was developed for one of the main systems engineering modeling tools, known as Capella, that allows the simulation of statecharts. Therefore, the developed tool was applied in two state machines to exemplify its practical behavior.

List of Figures

FIGURE 2.1 – Relationships, interfaces and entities in domain integration (LOCKHEED MARTIN CORPORATION, 2015).	16
FIGURE 3.1 – Simulator Extension Software Architecture.	20
FIGURE 4.1 – State machine of the operating modes of an in-flight entertainment system.	22
FIGURE 4.2 – State machine that periodically performs subsystem checks.	23
FIGURE 4.3 – Capella Project Explorer.	23
FIGURE 4.4 – Options after right click at project explorer region.	24
FIGURE 4.5 – Simulator’s command interface.	24
FIGURE 4.6 – Simulator’s state machine interface.	25
FIGURE 4.7 – Simulator’s Parser and SM factories.	26
FIGURE 4.8 – Configuration file.	27
FIGURE 4.9 – IFE Operation Modes State Machine steps 1 to 4.	28
FIGURE 4.10 –IFE Operation Modes State Machine steps 5 to 8.	28
FIGURE 4.11 –IFE Operation Modes State Machine steps 9 to 12.	28
FIGURE 4.12 –IFE Operation Modes State Machine steps 13 to 16.	29
FIGURE 4.13 –IFE Operation Modes State Machine steps 17 to 20.	29
FIGURE 4.14 –IFE Operation Modes State Machine steps 21 to 24.	29
FIGURE 4.15 –IFE Operation Modes State Machine steps 25 to 28.	30
FIGURE 4.16 –Checking State Machine steps 1 to 4.	31
FIGURE 4.17 –Checking State Machine steps 5 to 9.	31
FIGURE 4.18 –Checking State Machine steps 9 to 12.	32

FIGURE 4.19 –Checking State Machine steps 13 to 16.	32
FIGURE 4.20 –Checking State Machine steps 17 to 20.	33
FIGURE 4.21 –Checking State Machine steps 21 to 24.	33
FIGURE 4.22 –Checking State Machine steps 25 to 28.	34

List of Abbreviations and Acronyms

MBSE	Model-Based System Engineering
SysML	Systems Modeling Language
UML	Unified Modeling Language

Contents

1	INTRODUCTION	14
1.1	Motivation	14
1.2	Hypothesis	14
1.3	Objective	14
2	BIBLIOGRAPHIC REVIEW	15
2.1	Model-Based Systems Engineering (MBSE)	15
2.2	Statecharts	17
2.3	Python	17
2.4	Capella	18
3	MATERIALS AND METHODS	19
3.1	Activities Plan	19
3.2	Extension development for the Capella tool	19
4	RESULTS AND DISCUSSION	22
4.1	Simulator Operation	23
4.2	Configuration	26
4.3	Applications	27
4.3.1	IFE Operation Modes State Machine	27
4.3.2	Checking State Machine	30
5	CONCLUSIONS	35
	BIBLIOGRAPHY	36

APPENDIX A – SIMULATOR CODE	37
A.1 Main	37
A.2 Simulator	38
A.3 CapellaModelAPI	41
A.4 State Machine Model	44
A.5 Abstract Factory	46
A.6 Parser Factory	47
A.7 Sismic Parser	47
A.8 Abstract SM	53
A.9 SM Factory	54
A.10 Sismic SM	54
A.11 Test Simulator	55

1 Introduction

1.1 Motivation

Systems engineering arises due to the need to develop complex projects that have several systems interacting with each other. Indeed, it is of paramount importance that the systematic models produced can be subjected to extensive critical reviews.

Because of this, there are softwares that help in the process of developing systems engineering projects based on models from the interaction between the systems. Such software allows engineers to build visual representations of the entities and their interactions in complex designs.

In this context, Capella software emerges as one of the main tools on the market for building model-based systems. However, such a tool does not have the ability to simulate the interactions between the entities of the systems through a state machine logic.

Thus, this work was motivated from the need to perform state machine simulations in systems engineering projects built in the Capella tool.

1.2 Hypothesis

By creating a Python extension for the Capella tool, allowing systemic models to be simulated through their state machine logics.

1.3 Objective

The objective of this work is to enable that, during the development of systems engineering projects using the Capella tool, it is possible to perform system simulations through a state machine logic, through the implementation of a Python extension for the Capella tool.

2 Bibliographic Review

2.1 Model-Based Systems Engineering (MBSE)

According to (LOCKHEED MARTIN CORPORATION, 2015) a model consists of a simplified version of a concept, structure or system, through a visual representation enabling an abstraction of a given entity.

Thus, it is possible to use models as an element that abstracts all the complexity of systems, from requirements, analyses, implementations and verifications, in order to facilitate the understanding of systems in different scenarios.

In fact, from the combination of representation by models with systems engineering, it enables a formalized modeling that includes the different phases of the development cycle of a system: from requirements to simulations or tests (LOCKHEED MARTIN CORPORATION, 2015).

Furthermore, in a systems modeling, preponderant aspects of the system are chosen so that such aspects, through an abstraction, become the focus of the models (SHEVCHENKO, 2020).

On the other hand, a model must be easy to understand, maintain and use, since models demonstrate at a high level to stakeholders the design of systems in order to facilitate understanding and visualization.

This is because the purpose of models is to capture the relationships, interfaces and entities present in a system, thus, according to (SHEVCHENKO, 2020), the modeling covers requirements relationships, behaviors, architecture, verification and validation, demonstrating how these domains are related and interconnected, as we can see in the figure 3.1.

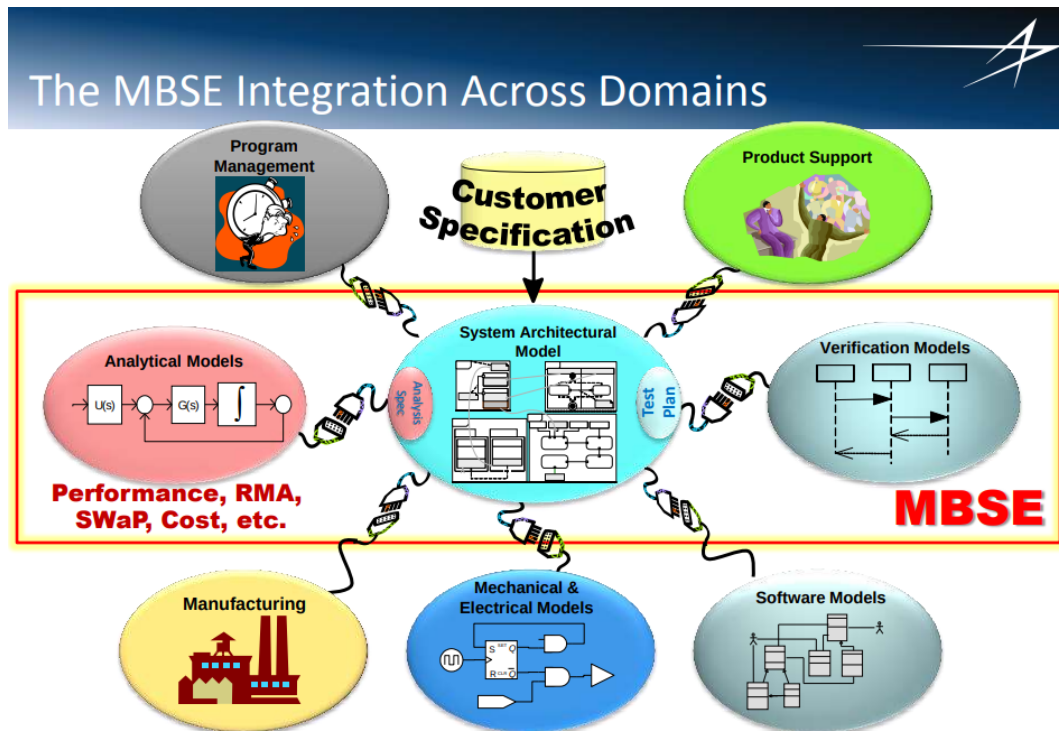


FIGURE 2.1 – Relationships, interfaces and entities in domain integration (LOCKHEED MARTIN CORPORATION, 2015).

Furthermore, an essential feature of MBSE is the formalized language that allows the understanding of models in different contexts. There are, for example, the Systems Modeling Language (SysML) and the Unified Modeling Language (UML). Among these, the SysML language, for being general purpose, has a higher degree of formality, avoiding some ambiguity (SYSML, 2021), on the other hand, UML is used in system modeling software such as Capella.

Thus, in MBSE models, based on the standardization of the language used in the modeling, there is support for the construction of state machine diagrams which allow the dynamic simulation of systems if, and only if, the models created are mathematically simulateable (SHEVCHENKO, 2020). This aspect is extremely important, since it allows analyzing the different states of the models and their transitions, allowing unexpected behaviors to be identified (SYSML, 2021).

The approach of analyzing models seeking to identify errors or unexpected behavior is known as Model-Based Review (INNOSLATE, 2017). In fact, the model review process takes place from the structuring of the capabilities and purposes of each entity involved, with this, the reviewers can produce inferences, corrections and improvements from an analysis of the model. However, for an evaluation of the dynamic behavior of the systems executed state machine diagrams specify dynamic system behaviors for critical situations in terms of time, mission, safety or financial aspects.

2.2 Statecharts

A state machine consists of a finite number of states containing transitions between each of the states, such that the current state and the current event produce a new state, consequently, such a process occurs repeatedly until all possible states occur. In fact, the concept of state consists of the current characteristics of the system, a state can receive inputs, when this occurs, due to the reactivity of the state, it produces outputs as actions to be performed and, if performed, new states are generated (ITEMIS, 2022).

Moreover, for systems that are too complex, there is an approach that uses statecharts to reduce the complexity of state machines. Statecharts consist of combinations of low-level state machines compiled in order to create high-level state machines so that the low-level state machines are orthogonal, that is, not contained within each other. In addition, communication between state machines must be allowed, so that events output from one state machine can be input from another (ITEMIS, 2022).

On the other hand, there is a formalism to describe a state machine, therefore, there is a list of basic concepts that denote specific functions of state machines that can be used in the construction of a diagram. Among the possible formal languages that describe a state machine, there is the UML that uses the following concepts to make a description: initial transition, events, states, actions and transitions, guard conditions, entry actions and exit actions.

In fact, the initial transition specifies the initial state of the system when it starts working. Moreover, the events consist of actions that the system receives and that generate some reaction. The states represent the result of a historical sequence of events transitions or actions of state and the response given by a state to a given event. The guard conditions continually evaluate states so that, if their condition is satisfied, they allow transitions that do not occur if the conditions are not satisfied. At end, run-to-completion execution model must satisfied, this concept says that the state machine process some event and just at the end of processing a new event could be start processing, so the events can't occur simultaneously (EMBEDDED, 2009).

2.3 Python

Python is a high-level, multipurpose open-source programming language with a wide variety of available code libraries that uses object-oriented programming initially conceived in the 1980s (WITMAN, 2021a).

The syntax of the language is simple and intuitive, easy to write and read, becoming extremely popular over the years, being used in various areas of engineering by large

companies such as Google, NASA and IBM, as it is very versatile and can be used in many types of applications, including machine learning applications to interface creation.

Since this programming language is extremely versatile, it can be used in website development, applications, machine learning projects, numerical simulations and data analysis.

The main Python applications in the modern world consist of data analysis and visualization, since there are widely used libraries that allow the creation of complex statistical reports and data processes.

In addition, there are applications in the areas of graphical interface construction, application programming interfaces, artificial intelligence, machine learning, among many other areas. This demonstrates that the python programming language, in fact, has many applications validated in different areas of knowledge and by different agents of society.

2.4 Capella

Capella is a modelling tool open-source that used model-based engineering approach to build a efficient architectural design through a graphical modelling workbench according to ARCADIA method recommendations (WITMAN, 2021b) . In fact, this software was build by PolarSys and is used in many enterprises like Embraer and SIEMENS or universities like ITA (Instituto Tecnológico de Aeronáutica) and UFSM (Universidade Federal de Santa Maria) (ECLIPSE, 2022).

Moreover, Capella focus on the embedded methodology browser, manage architecture complexity, model-to-model transformations and has the capacity of extend this environment with libraries to allows the users creates new features to the tool (WITMAN, 2021b).

In this context, Python4Capella is one of libraries that extend Capella features and allow users build extension to Capella using Python, a programming language that is easy to write and read (LABS4CAPELLA, 2022).

The Capella software supports the construction of statecharts using UML, using this language it is possible to perform dynamic simulations of the behavior of the models through the interpretation of the state machine built in the tool. When a statecharts is built, the Capella software still not performs a behavioral simulation engine and, thus, you can't get all the states and their respective transitions as responses to triggers resulting in actions. Thus, information about the life cycle of a system isn't constructed (SPARX SYSTEMS, 2022).

3 Materials and Methods

3.1 Activities Plan

Initially, a state machine reader is built from the information entered in Capella, this reader abstracts the constructed state machine and serializes it through the human-friendly YAML language.

Subsequently, the serialized information is interpreted through the created extension, enabling the construction of a state object that carries all the logic of the sequential execution of the statecharts, with this, the state machine is executed.

From that, at each step of the simulation, the information of the new states is coded again, sent to Capella and, through the construction of a simulation interface, displayed on the screen denoting the current state and allowing the user to go through the simulation through an interface.

3.2 Extension development for the Capella tool

The creation of a simulation platform in Capella software was carried out from the creation of modules that perform different functions, for example, there is a module whose objective is to be a high-level interface between the Simulator and the Capella software, this module uses the Python4Capella library functions. On the other hand, the Simulator is composed of some functions and modules, among them there are modules that build the command interface and modules that deal with commands generated by the user. The architecture of this system can be seen in figure 3.1.

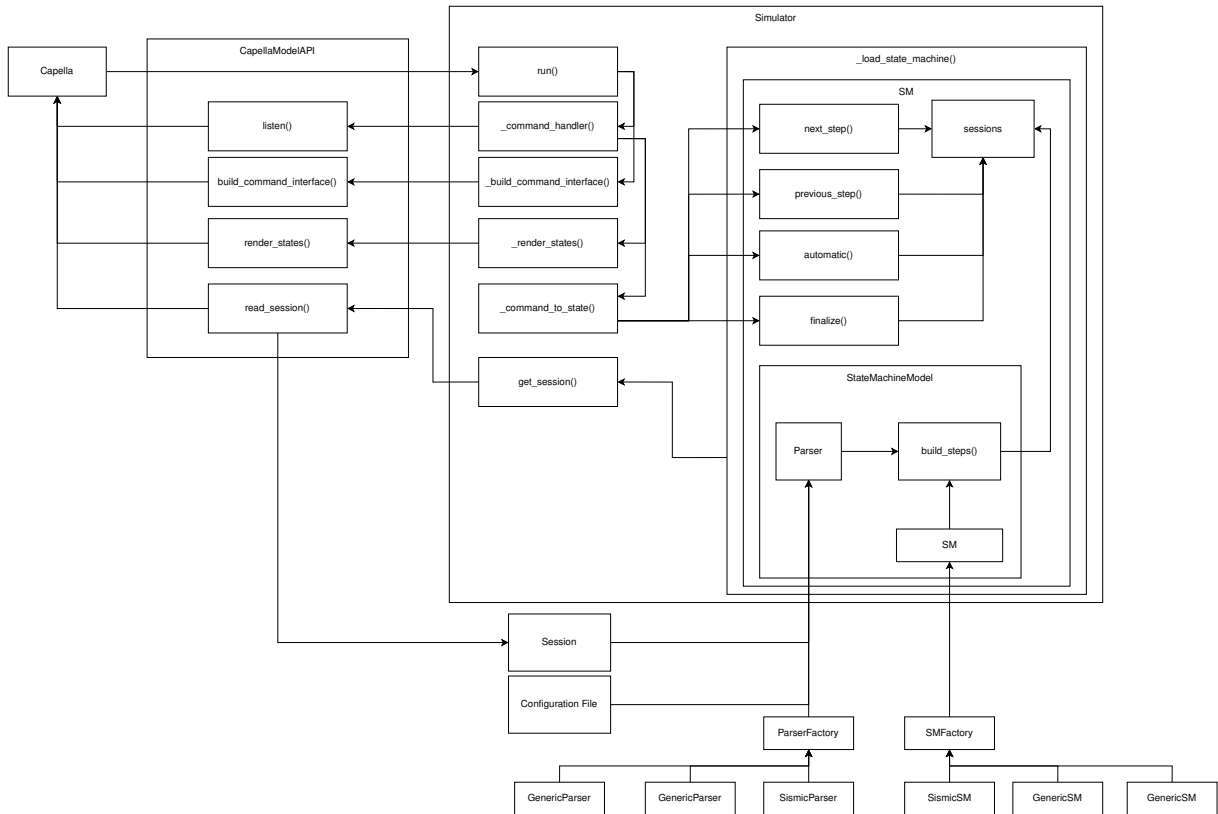


FIGURE 3.1 – Simulator Extension Software Architecture.

In addition, it was also necessary to create a global state machine logic named SM with the concepts of step transition, step execution and completion of the simulation process. This component executes its actions from the interaction with an object whose responsibility is to store the sessions and whose construction is done by the StateMachineModel module that, from the configurations received from the configuration files and from the session data of the Capella software, performs through a design pattern known as Factory the creation of the right Parser and SM objects among the many possibilities of Parsers and SMs.

The purpose of a Parser is to translate the session information into a language that can be interpreted by the SM, in fact, the Parser performs a reading of the session information from the Capella software and organizes this information so that it becomes readable by the SM. Furthermore, the objective of SM is to determine all the simulation steps defining all possible transitions and states, for this, a python library known as Sismic was used, which, from the definition of a document in the YAML extension, performs the simulation of a state machine generating all the states of that simulation.

Thus, to create this system with the aim of running simulations of state machines, it was necessary to architect a system in which there was a high-level communication interface between the Simulator and Capella through the Python4Capella library in which it was possible to execute methods of interface construction and command listening. How-

ever, it was also necessary to create a simulation module named Simulator that would perform a general management, at a high level, of the state machines, initializing such machines and controlling the steps through the reception of commands from the Capella software. Finally, it was also necessary to create the factory design pattern that allows generalizing the system in such a way that it was possible to couple different modules that perform the simulation of state machines and, for each of these modules, different parsers to translate the information from the Capella Session for the coupled simulation modules.

Furthermore, in order to maintain the quality of the software and its correct functioning despite its complex logic, it was necessary to implement a unit testing structure to verify that each part of the code was functioning as expected. This approach allows the software to be less affected by changes that cause critical errors and avoids side effects of eventual implementations that may occur in the future of this software. This approach also guarantees greater reliability to the execution of the software as it guarantees the cohesion of the tested code snippets, with this, a series of tests are created and are executed with each new modification in the project ensuring its correct operation.

4 Results and Discussion

To analyze the proposed operation of a state machine simulator in Capella software, two state machines were used, one obtained from an exemplary Capella project and another created to exemplify a checking system that will allow the verification of the simulation extension developed.

The first state machine can be seen in figure 4.1, it reflects the operation of the operating modes of an in-flight entertainment system.

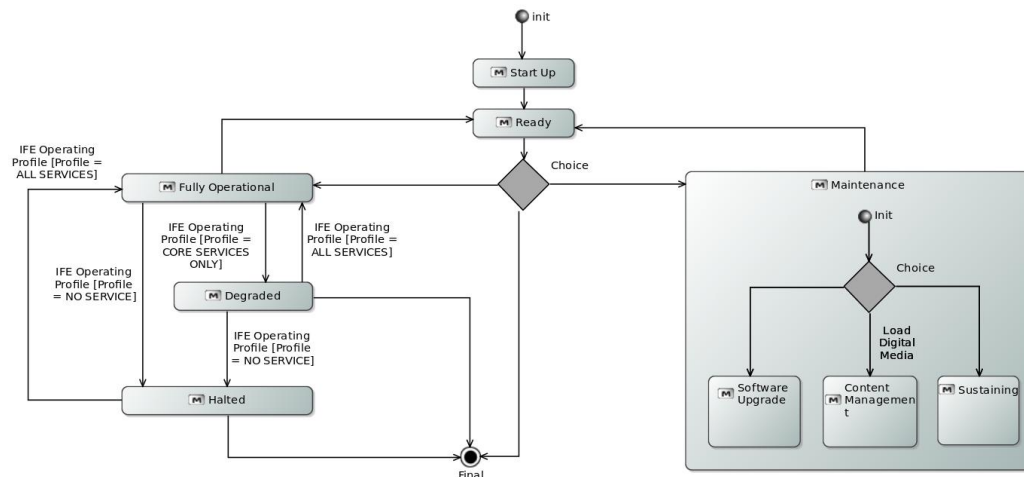


FIGURE 4.1 – State machine of the operating modes of an in-flight entertainment system.

The second state machine can be seen in figure 4.2, it reflects the functioning of a system that performs subsystem checks periodically.

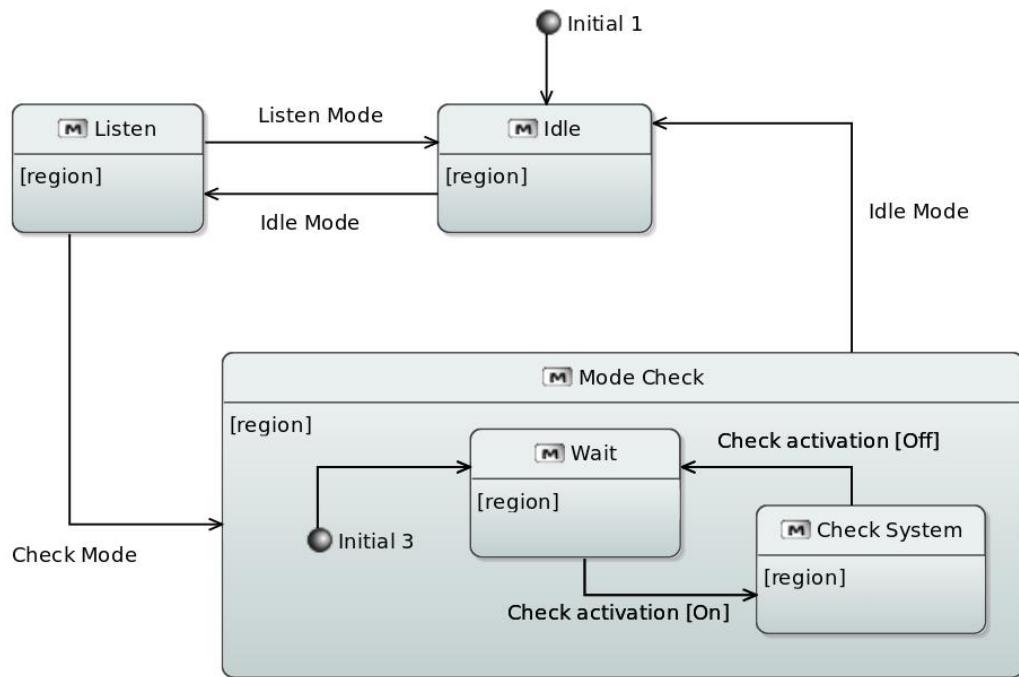


FIGURE 4.2 – State machine that periodically performs subsystem checks.

4.1 Simulator Operation

To run the simulator, it is necessary to right-click any area within the exploration region of the projects, as shown in figure 4.3, after that, it is necessary to click on Simulator, as shown in figure 4.4, with this, the simulation will start and a simulation control interface will be displayed, as shown in figure 4.5.

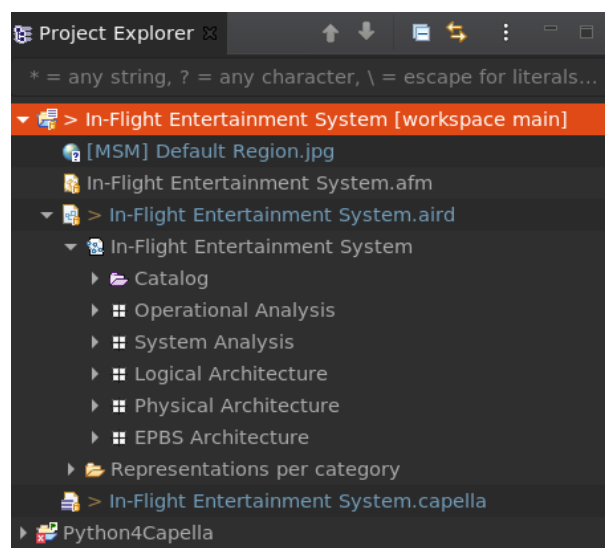


FIGURE 4.3 – Capella Project Explorer.

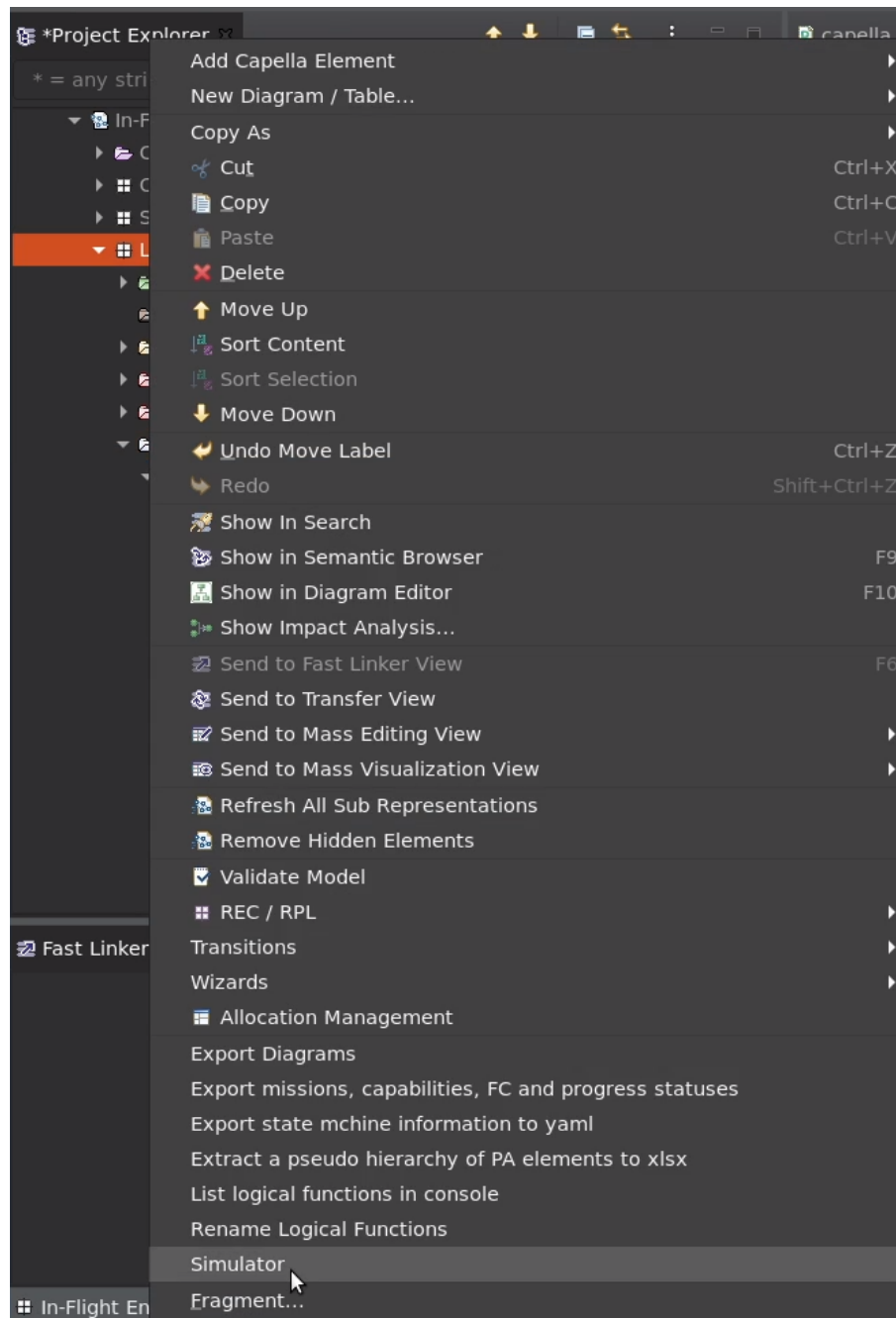


FIGURE 4.4 – Options after right click at project explorer region.

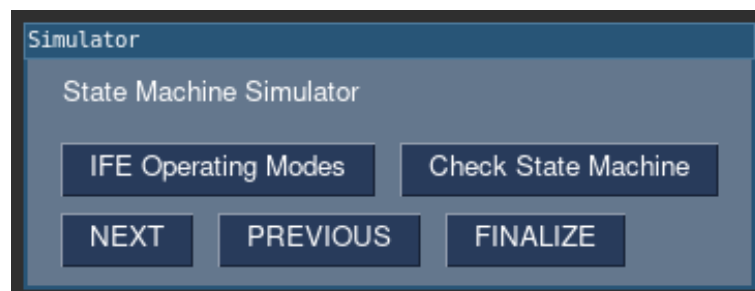


FIGURE 4.5 – Simulator's command interface.

Furthermore, through the command interface, the user can move between the states by going to the next simulation step using the NEXT option or by going to the previous simulation step using the PREVIOUS option. On the other hand, if necessary, the user can end the simulation using the FINALIZE button. In addition, the user can switch between which state machine will be shown by clicking in the option with state machine name.

The simulation results can be observed through images presented within the simulator area, the first image will be displayed after the first click on the NEXT option, as seen in the figure 4.6. Moreover, the yellow color represents the current state and the blue color represents state before the current state.

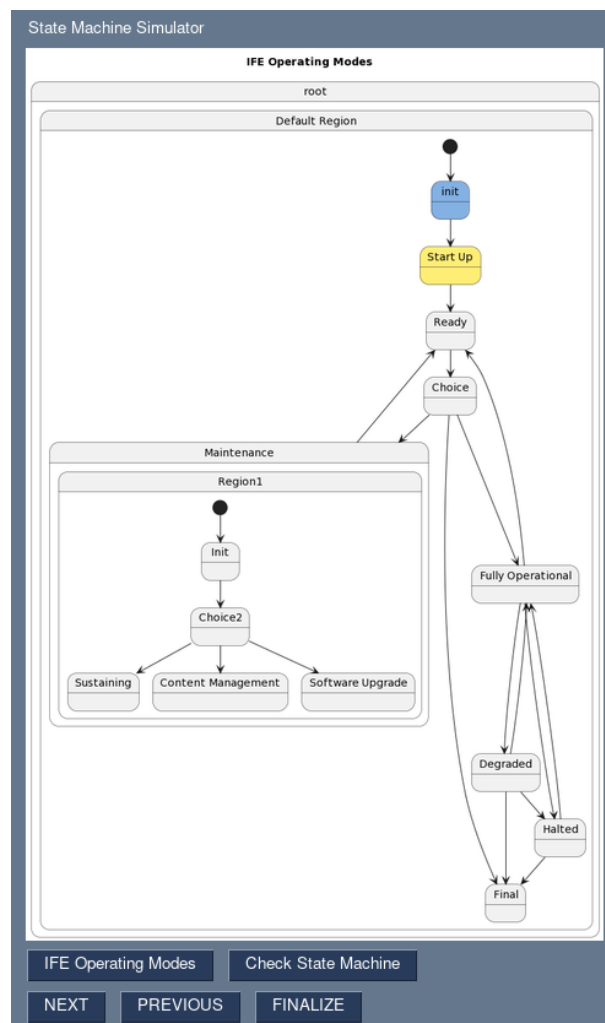


FIGURE 4.6 – Simulator’s state machine interface.

Through the results presented in the simulation, we can verify which transitions occurred between the states, which are the current states of the simulation and which are the states prior to the current state.

4.2 Configuration

For the operation of the Simulator, it is necessary to configure the simulation, as there are parameters that can be varied according to the simulation and the user's objective. The possible current configurations consist of defining the parser that should be selected and the simulation module that will be coupled to the platform, in such a way that the value inserted in the JSON file will be responsible for informing to the factory which type of Parser and SM it should produce, in figure 4.7 it is possible to observe the operation of the factory. From the choice of these two parameters, a file with the JSON extension is created according to figure 4.8.

```
class ParserFactory:
    """
    Parser Factory
    """

    def __new__(cls, type) -> Self:
        products = {
            'sismic': SismicParser
        }
        return products[type]()

class SMFactory:
    """
    State Machine Factory
    """

    def __new__(cls, type) -> Self:
        products = {
            'sismic': SismicSM
        }
        return products[type]()
```

FIGURE 4.7 – Simulator's Parser and SM factories.

```
// JSON Configuration File
{
  'state_type': 'sismic',
  'parser_type': 'sismic'
}
```

FIGURE 4.8 – Configuration file.

4.3 Applications

Thus, to exemplify the practical operation of the State Machine Simulator, two simulations were performed with the two state machines mentioned in the figures 4.1 and 4.2. From these machines, a analysis of the execution of the simulations will be carried out. It is important to note that, for both simulations, the configuration file shown in the figure 4.8 was used, whose parser translates the Capella Session into a YAML file that can be interpreted by the Sismic library and, later, managed by the simulator.

4.3.1 IFE Operation Modes State Machine

From the execution of the simulation, the first 28 micro steps of the simulation are shown in the figures 4.9, 4.10, 4.11, 4.12, 4.13, 4.14 e 4.15.

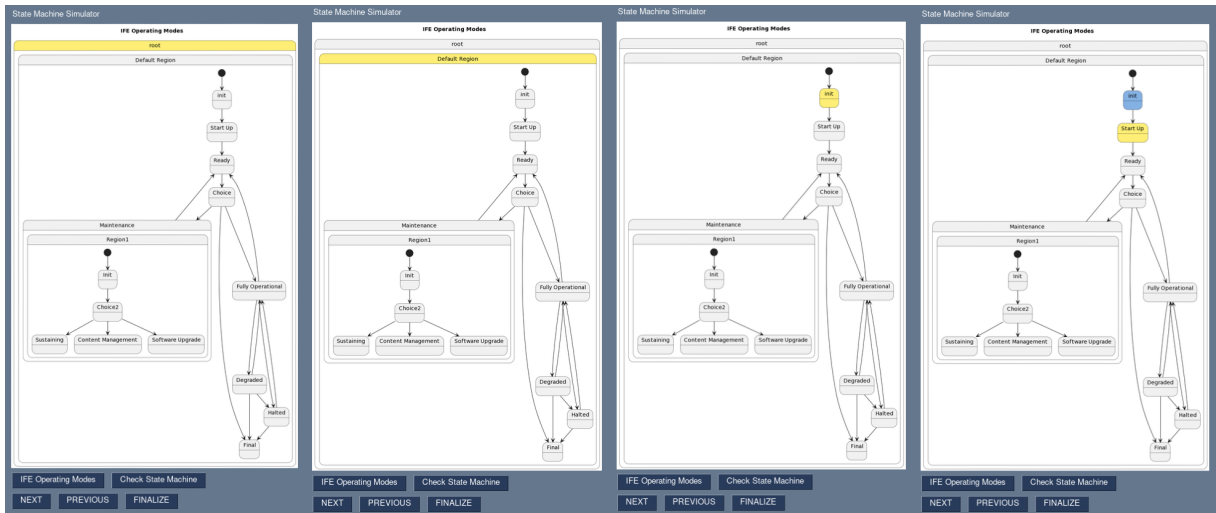


FIGURE 4.9 – IFE Operation Modes State Machine steps 1 to 4.

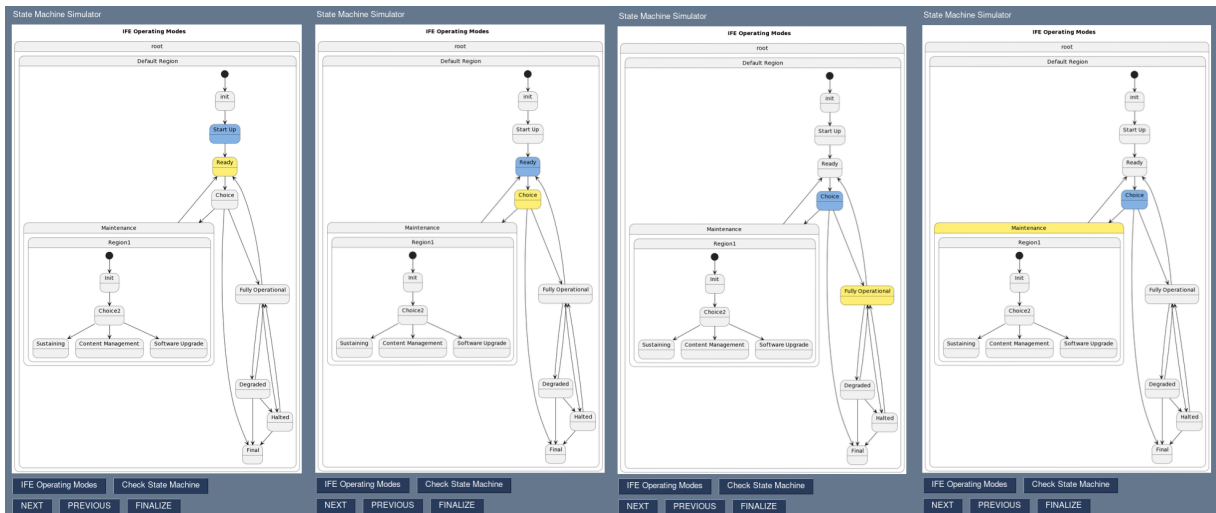


FIGURE 4.10 – IFE Operation Modes State Machine steps 5 to 8.

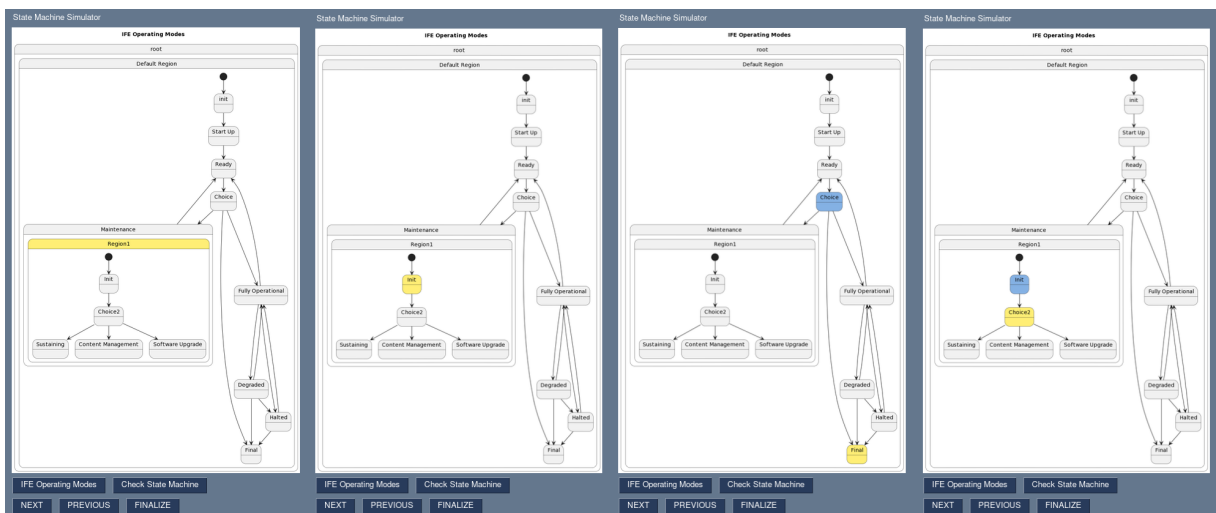


FIGURE 4.11 – IFE Operation Modes State Machine steps 9 to 12.

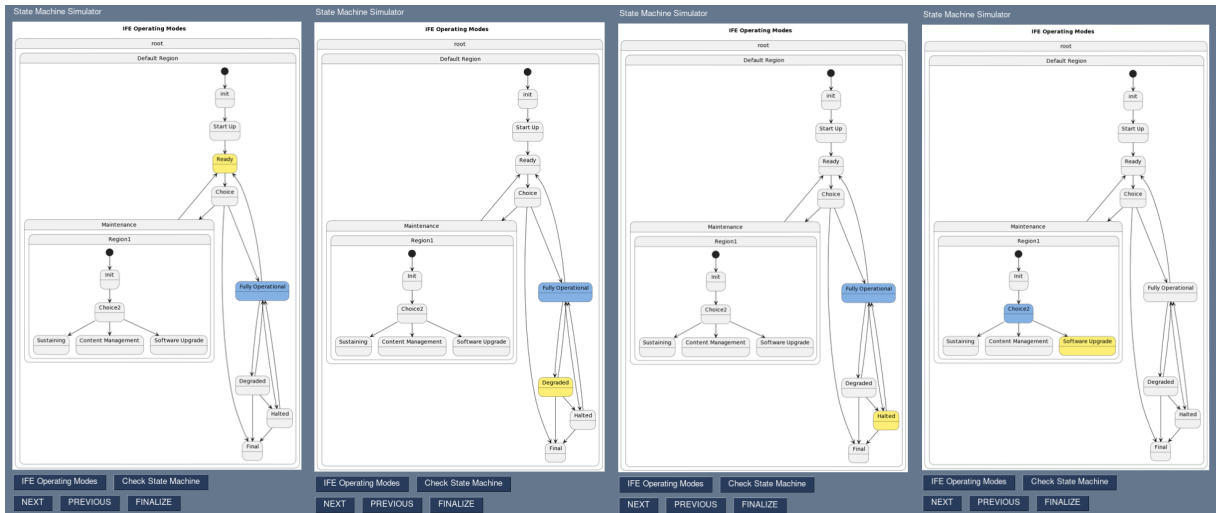


FIGURE 4.12 – IFE Operation Modes State Machine steps 13 to 16.

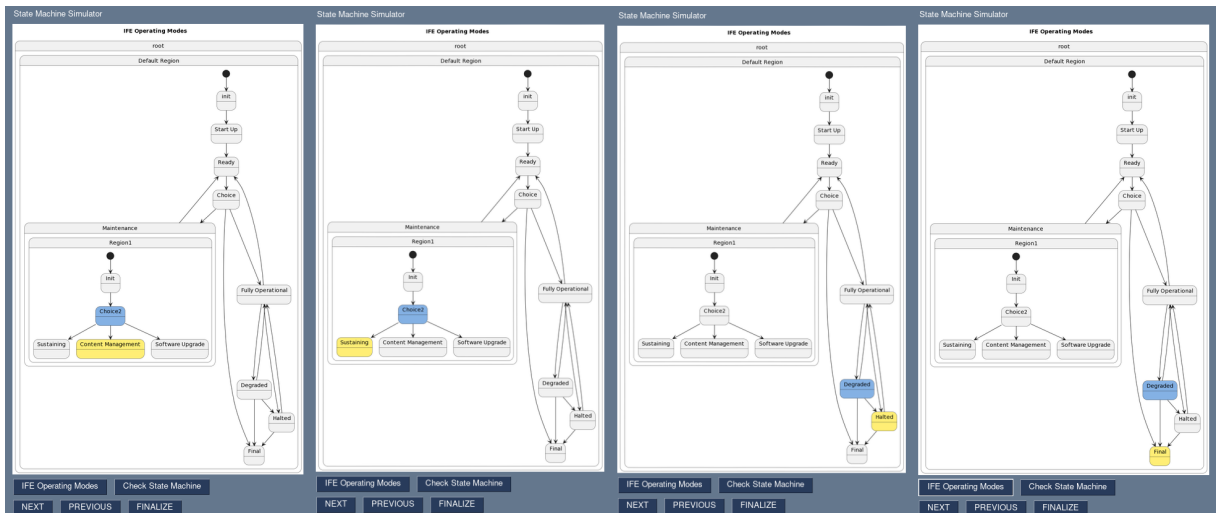


FIGURE 4.13 – IFE Operation Modes State Machine steps 17 to 20.

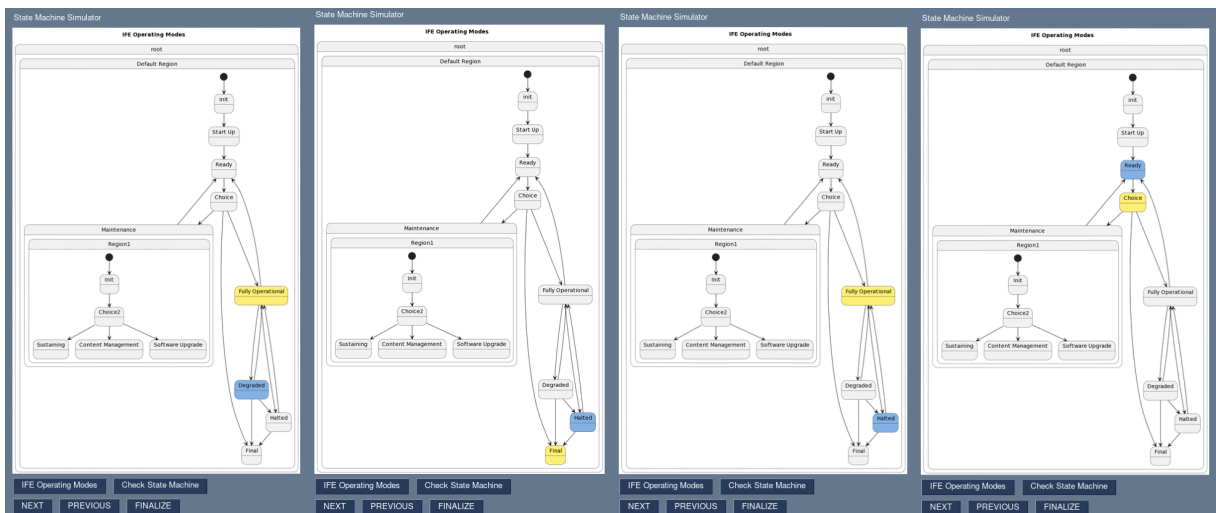


FIGURE 4.14 – IFE Operation Modes State Machine steps 21 to 24.

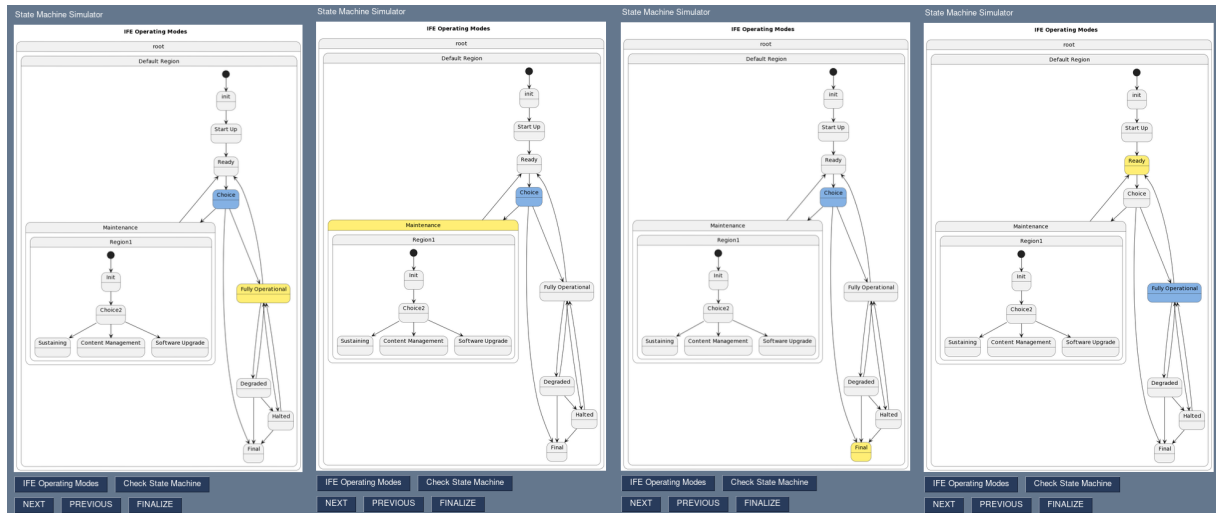


FIGURE 4.15 – IFE Operation Modes State Machine steps 25 to 28.

Therefore, we can observe the coherence of the state machine simulator that achieves its goal by traversing the transitions and states correctly, according to the state machine shown in the figure 4.1. In this case, it is worth mentioning the existence of a possible warning in this state machine, which is the execution of state machines in parallel by the simulator, as can be seen in the figure 4.11, in which during the execution of the Maintenance state, Simultaneously, the Final state is executed from the Choice state.

4.3.2 Checking State Machine

From the execution of the simulation, the first 28 micro steps of the simulation are shown in the figures 4.16, 4.17, 4.18, 4.19, 4.20, 4.21 e 4.22.

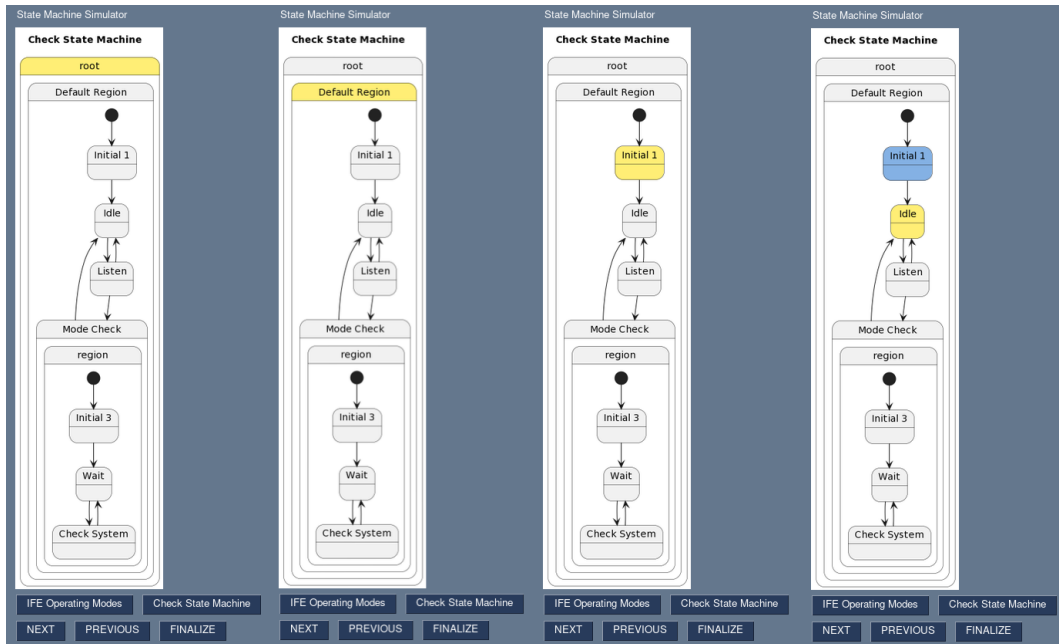


FIGURE 4.16 – Checking State Machine steps 1 to 4.

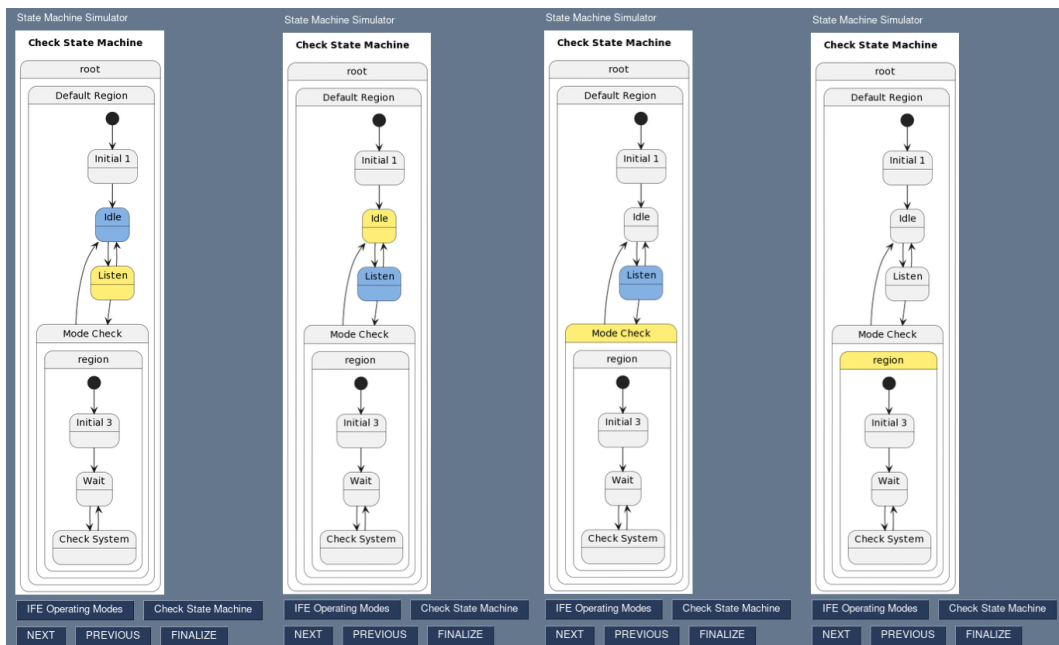


FIGURE 4.17 – Checking State Machine steps 5 to 9.

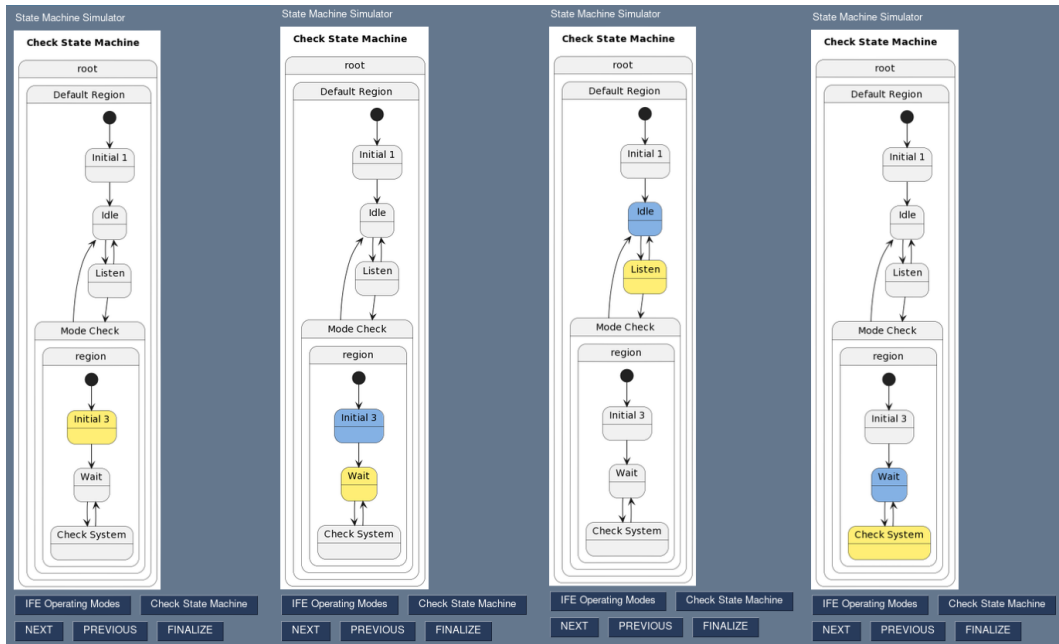


FIGURE 4.18 – Checking State Machine steps 9 to 12.

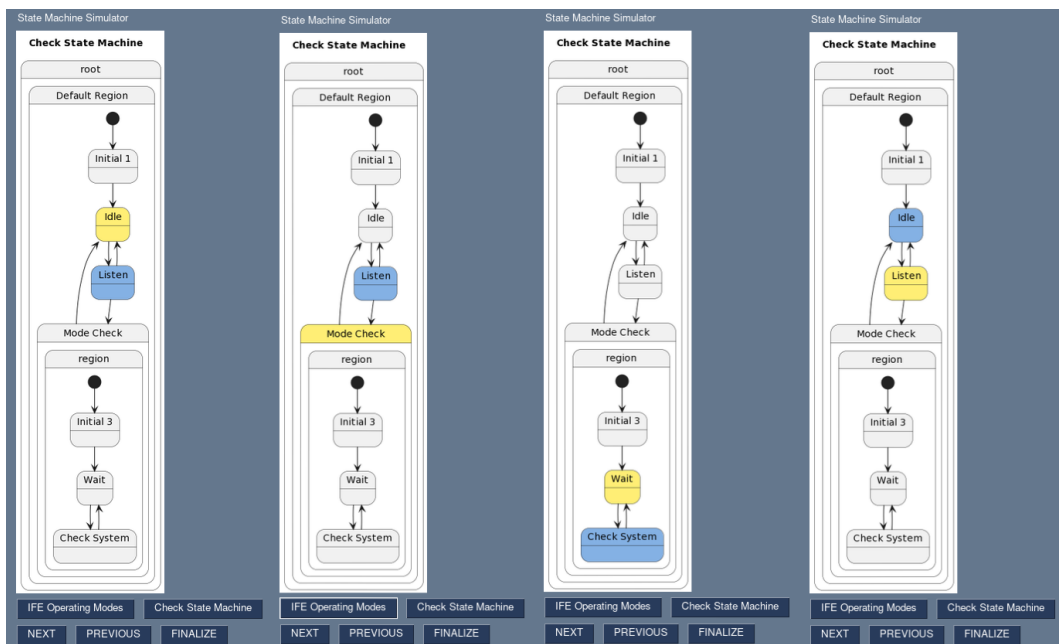


FIGURE 4.19 – Checking State Machine steps 13 to 16.

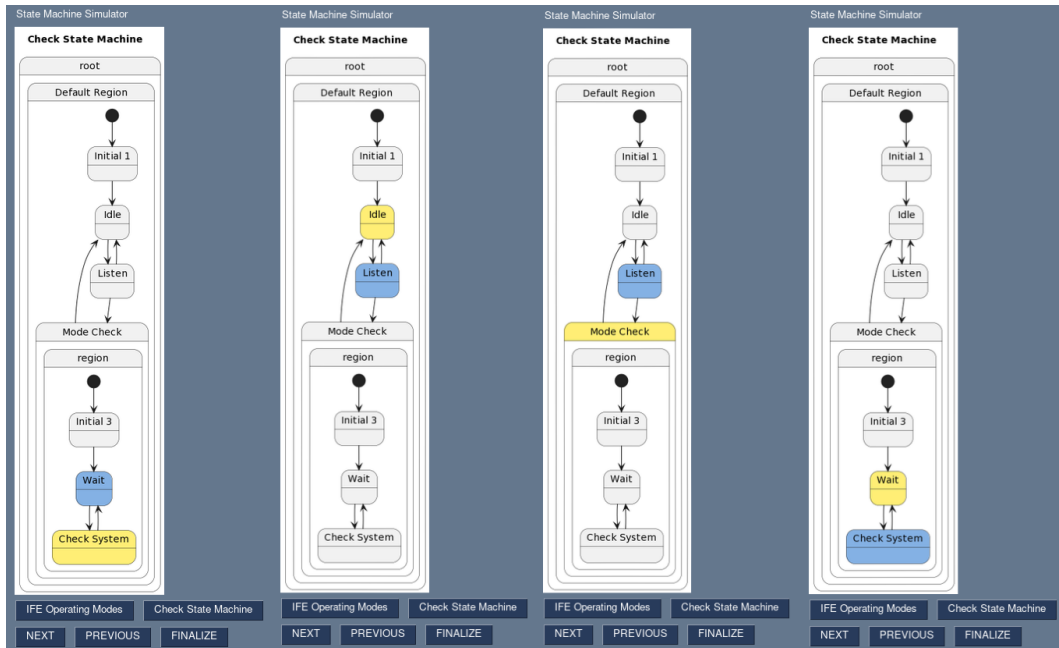


FIGURE 4.20 – Checking State Machine steps 17 to 20.

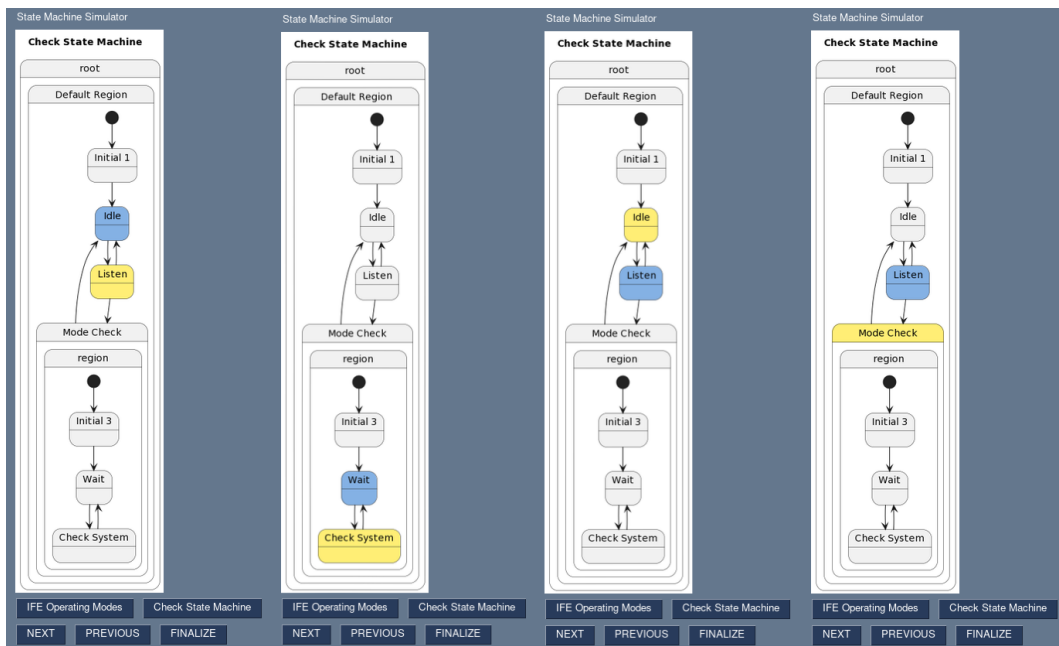


FIGURE 4.21 – Checking State Machine steps 21 to 24.

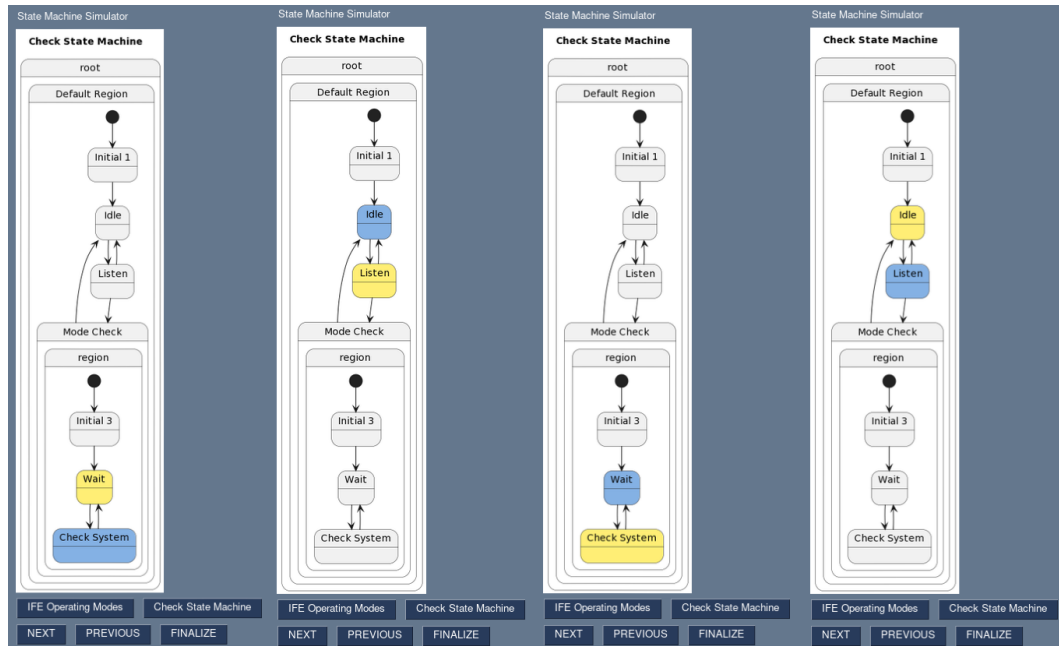


FIGURE 4.22 – Checking State Machine steps 25 to 28.

Therefore, we can observe the coherence of the state machine simulator that achieves its goal by traversing the transitions and states correctly, according to the state machine shown in the figure 4.2. In this case, it is worth mentioning a possible warning in this state machine, which is the existence of a loop between the Idle and Listen states, as seen in the figures 4.17, 4.19, 4.21 and 4.22, in which the Idle and Listen state switches continuously.

5 Conclusions

The development of this work enabled the construction of a tool that extends the Capella software and allows the execution of statecharts through a state machine logic. Therefore, this tool facilitates the development of projects using a model based methodology since the state machine simulations can be performed automatically and reviewed by their authors.

Furthermore, to develop the State Machine Simulator it was necessary to develop a software architecture whose objective was to manage the interaction logics with the Capella software, manage the state steps and enable the coupling of different external state machine simulators to increase the system flexibility. On the other hand, to maintain the quality of the software, a unit testing infrastructure was created to guarantee its correct functioning.

In addition, it is worth mentioning that the main value generated by this work is its ability to abstract the interface logics with the simulator modules and with Capella software, encapsulating the complex logics that involve each of these modules in the construction of steps and in their management through an intuitive interface. This was possible thanks to the study of software engineering techniques that are based on principles of creating understandable, flexible and maintainable software.

Therefore, the present work is the basis for the development of more elaborate state machine simulation methods, since it abstracts all the state management logic through an intuitive interface. Furthermore, the system allows flexibility in the integration of state simulator modules, since it uses the design pattern factory, making possible the integration not only with the Sismic library, but also with other simulators such as MATLAB's Simulink. On the other hand, as observed in the results section, a warning system of possible points of attention can be developed to help the user to find possible problems in his state machine.

Bibliography

ECLIPSE. **ADOPTERS**. 2022. Available from Internet:

<https://www.eclipse.org/capella/adopters.html>. Accessed on: 05 jun. 2022.

EMBEDDED. **A crash course in UML state machines: Part 1**. 2009. Available from Internet: <https://www.embedded.com/a-crash-course-in-uml-state-machines-part-1/>.

Accessed on: 05 jun. 2022.

HART, L. E. Introduction to model-based system engineering (mbse) and sysml. In: LOCKHEED MARTIN CORPORATION, 1., 2015, Delaware Valley. INCOSE, 2015. Available from Internet: <https://www.incose.org/docs/default-source/delaware-valley-/mbse-overview-incose-30-july-2015.pdf>. Accessed on: 07 jun. 2022.

INNOSLATE. **How to Perform a Model-Based Review (MBR)**. 2017. Available from Internet: <https://www.innoslate.com/resource/model-based-review-whitepaper/>.

Accessed on: 05 jun. 2022.

ITEMIS. **How to Perform a Model-Based Review (MBR)**. 2022. Available from Internet: https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines. Accessed on: 05 jun. 2022.

LABS4CAPELLA. **Python4Capella**. 2022. Available from Internet:

<https://github.com/labs4capella/python4capella>. Accessed on: 07 jun. 2022.

SHEVCHENKO, N. An introduction to model-based systems engineering (mbse). p. 1, dez. 2020. Available from Internet:

<https://insights.sei.cmu.edu/blog/introduction-model-based-systems-engineering-mbse/>. Accessed on: 05 jun. 2022.

SPARX SYTEMS. **SysML StateMachine Diagram**. 2022. Available from Internet:

https://sparxsystems.com/enterprise_architect_user_guide/15.2/model_domains-/sysml_statemachine_diagram.html. Accessed on: 05 jun. 2022.

SYSML. **SysML Open Source Project: What is sysml? who created sysml?** 2021.

Available from Internet: <https://sysml.org/>. Accessed on: 05 jun. 2022.

WITMAN, E. **What is Python? The popular, scalable programming language, explained**. [S.l.], 2021. Available from Internet:

<https://www.businessinsider.com/what-is-python>. Accessed on: 05 jun. 2022.

WITMAN, E. **What is Python? The popular, scalable programming language, explained**. [S.l.], 2021. Accessed on: 05 jun. 2022.

Appendix A - Simulator Code

A.1 Main

```
# name: Simulator
# script-type: Python
# description: Simulator
# popup: enableFor(org.polarsys.capella.core.data.capellacore.CapellaElement)

import sys
import subprocess

def version():
    subprocess.check_call([sys.executable, "--version"])

def install(package):
    subprocess.check_call([sys.executable, "-m", "pip", "install", package])

version()

install("pyyaml")
install("pysimplegui")

include('workspace://Python4Capella/sample_scripts/simulator/simulator.py')

# include needed for the Capella modeller API
include('workspace://Python4Capella/simplified_api/capella.py')

# Retrieve the Element from the current selection and its aird model path
selected_elem = CapellaElement(CapellaPlatform.getFirstSelectedElement())
aird_path = '/' + CapellaPlatform.getModelPath(selected_elem)
```

```
model = CapellaModel()
model.open(aird_path)

simulator = Simulator(model, config={
    'model_path': 'default',
    'state_type': 'sismic',
    'parser_type': 'sismic'
})

simulator.run()
```

A.2 Simulator

```
import sys

if "pytest" in sys.modules:
    from simulator.core.capella_api import *
    from simulator.core.state_machine import *
else:
    # CapellaModelAPI
    include('workspace://Python4Capella/sample_scripts' \
           '/simulator/core/capella_api.py')

    include('workspace://Python4Capella/sample_scripts' \
           '/simulator/core/state_machine.py') # SM

class Simulator:
    """
    Essa classe interage em alto nível com o capella e a SM
    """

    def __init__(self, model, config):
        # Carrega configurações
        self.states = None
        self.config = self._parse_config(config=config)
```

```
# Inicializa o capella
self.capella = CapellaModelAPI(model=model)

# Inicializa capella e máquina de estados
self.state_machine = self._load_state_machine()

def run(self):
    # Constrói a interface de comando
    self._build_command_interface()

    # Escuta comandos enviados
    while True:
        self._command_handler()

        if self.states == None:
            print("Simulation finalized!")
            return

def _build_command_interface(self):
    sm_buttons = [state['name'] for state in self.state_machine.states]
    self.capella.build_command_interface(sm_buttons)

def _command_handler(self):
    """
        Escuta e executa comandos.
    """
    command = self.capella.listen() # be awaiting for a new command
    if command:
        self._command_to_state(command)
        self._render_states()

def _command_to_state(self, command):
    self.states = self._map_commands(command)()

def _render_states(self):
    if self.states:
        self.capella.render_states(states=self.states)

def _map_commands(self, command):
```



```
# Mapeia strings a comandos
commands = {
    'next_step': self.state_machine.next_step,
    'previous_step': self.state_machine.previous_step,
    'finalize': self.state_machine.finalize,
    'automatic': self.state_machine.automatic
}
return commands[command]

def _get_session(self):
    return self.capella.read_session()

def _load_state_machine(self):
    # Carrega a state machine
    session = self._get_session()
    state_type = self.config['state_type']
    parser_type = self.config['parser_type']

    return SM(
        session=session,
        state_type=state_type,
        parser_type=parser_type
    )

@staticmethod
def _parse_config(config):
    # Cria configurações válidas
    parsed_config = {
        'model_path': 'default',
        'state_type': 'sismic',
        'parser_type': 'sismic'
    }

    for key, value in config.items():
        parsed_config[key] = value if key in parsed_config else None

    return parsed_config
```

A.3 CapellaModelAPI

```
import re
import io
import PySimpleGUI as sg

from PIL import Image, ImageTk
from plantweb.render import render

class CommandInterface:
    def __init__(self, buttons):
        layout = [
            [sg.Text("State Machine Simulator")],
            [sg.Image(key='-IMAGE-' + button) for button in buttons],
            [sg.Button(button) for button in buttons],
            [sg.Button("NEXT"), sg.Button("PREVIOUS"), sg.Button("FINALIZE")],
        ]
        self.window = sg.Window("Simulator", layout, location=(50, 50))

class CapellaModelAPI:
    def __init__(self, model):
        self.model = model
        self.command_interface = None
        self.images = {}
        self.buttons = []
        self.button = None

    def build_command_interface(self, buttons):
        """
        Constrói no capella uma interface de comando
        para que o usuário gerencie os steps da simulação.

        Obs: enquanto ela não for construída, executaremos
        cada step da simulação de 1 em 1 segundo.
        """

        self.buttons = buttons
```

```
    if len(buttons) > 0:
        self.button = buttons[0]

    self.command_interface = CommandInterface(buttons)

def listen(self):
    """
        Fica escutando caso ocorra uma
        interação entre o usuário e a interface de comando.
        Se ocorrer, exibe o step requisitado pelo usuário.
    """

    if self.command_interface:
        event, _ = self.command_interface.window.read()

        if event in self.buttons:
            self.button = event
            for key in self.buttons:
                self.command_interface.window['-IMAGE-' +
                    key].update(visible=key == self.button)
            return None

        if event == "NEXT":
            return 'next_step'

        if event == "PREVIOUS":
            return 'previous_step'

        if event == sg.WIN_CLOSED or event == "FINALIZE":
            return 'finalize'

    return 'finalize'

def render_states(self, states=None):
    # TODO: render state in capella
    for state in states:
        step = state['step']
        name = state['name']
        plantuml = state['plantuml']
```

```
print("--> MicroStep", name)
print('event:', step.event)
print('transition:', step.transition)
print('entered_states:', step.entered_states)
print('exited_states:', step.exited_states)
print('sent_events:', step.sent_events)

for entered in step.entered_states:
    tmp_plantuml = self._change_plantuml_color(plantuml, \
        entered, "#FFEE75")

for exited in step.exited_states:
    tmp_plantuml = self._change_plantuml_color(tmp_plantuml, \
        exited, "#84B1E4")

print('plantuml', tmp_plantuml)

outfile = render(
    tmp_plantuml,
    engine='plantuml',
    format='png',
    cacheopts={
        'use_cache': False
    }
)

im = Image.open(io.BytesIO(outfile[0]))
im.thumbnail((1024, 728), Image.Resampling.LANCZOS)
self.images[name] = ImageTk.PhotoImage(image=im)

for key in self.images:
    self.command_interface.window['-IMAGE-' + key].update(
        data=self.images[key], visible=key == self.button)

def read_session(self):
    # Read capella session
    return self.model.get_system_engineering()
```

```

def _change_plantuml_color(self, plantuml, state, color):
    pattern = r"{state}"(.*){'.format(state=state)

    for m in re.finditer(pattern, plantuml):
        end = m.end() - 1
        tmp_plantuml = plantuml[:end] + color + " " + plantuml[end:]

    return tmp_plantuml

```

A.4 State Machine Model

```

import sys

if "pytest" in sys.modules:
    from simulator.parsers.factory_parser import *
    from simulator.sms.factory_sm import *
else:
    # ParserFactory
    include('workspace://Python4Capella/sample_scripts' \
           '/simulator/parsers/factory_parser.py')
    # SMFactory
    include('workspace://Python4Capella/sample_scripts' \
           '/simulator/sms/factory_sm.py')

class StateMachineModel:
    """
    Construção de um modelo de máquina de estados
    """

    def __init__(self, session, state_type, parser_type):
        # Inicializa a máquina de estados, parser e sessão
        self.sm = SMFactory(type=state_type)
        self.parser = ParserFactory(type=parser_type)
        self.sessions = self.parser.sessions(session=session)

    def build_steps(self):
        # Chama método de construção de estados

```

```
steps_sms = [self.sm.build_steps(session=s) for s in self.sessions]

# Traduz os estados para a linguagem do capella
return [{
    'name': steps['name'],
    'steps': [self.parser.step(step) for step in steps['steps']],
    'plantuml': steps['plantuml']
} for steps in steps_sms]

class SM:
    """
    Abstração da lógica de máquinas de estado.
    """

    def __init__(self, session=None, state_type=None, parser_type=None):
        # Inicializa os estados, o step e o modelo
        self.step = -1

        self.model = StateMachineModel(
            session=session,
            state_type=state_type,
            parser_type=parser_type
        )

        self.states = self.model.build_steps()

    def next_step(self):
        # Vai para o próximo estado do modelo, se houver
        self.step = self.step + 1

        return [self.get_step(steps) for steps in self.states]

    def get_step(self, steps):
        stepslen = len(steps['steps'])
        if stepslen <= self.step and stepslen > 0:
            self.step = stepslen - 1

        return {
```

```
        'step': steps['steps'][self.step],
        'name': steps['name'],
        'plantuml': steps['plantuml']
    }

def previous_step(self):
    # Vai para o próximo estado do modelo, se houver
    if self.states == None:
        self.start()

    if self.step < 1:
        self.step = 0
    else:
        self.step = self.step - 1

    return [self.get_step(steps) for steps in self.states]

def automatic(self):
    if self.states == None:
        return self.start()

    return self.next_step()

def finalize(self):
    return None
```

A.5 Abstract Factory

```
from abc import ABC

class AbstractParser(ABC):
    """
    Abstração de um parser
    """

    def session(self):
        # abstract
        pass
```

```
def state(self):  
    # abstract  
    pass
```

A.6 Parser Factory

```
import sys  
  
from typing_extensions import Self  
  
if "pytest" in sys.modules:  
    from simulator.parsers.sismic_parser import *  
else:  
    # SismicParser  
    include('workspace://Python4Capella/sample_scripts' \  
            '/simulator/parsers/sismic_parser.py')  
  
class ParserFactory:  
    """  
    Fábrica de Parser  
    """  
  
    def __new__(cls, type) -> Self:  
        # Fabrica o modelo solicitado  
        products = {  
            'sismic': SismicParser  
        }  
        return products[type]()
```

A.7 Sismic Parser

```
import sys  
import yaml  
  
from collections import defaultdict
```



```
fromismic.io import import_from_yaml

if "pytest" in sys.modules:
    from simulator.parsers.abs_parser import *
else:
    include('workspace://Python4Capella/sample_scripts' \
            '/simulator/parsers/abs_parser.py')

class SismicParser(AbstractParser):
    """
        Traduz sessões do capella para o modelo
        Traduz estados do modelo para o capella
    """

    def __init__(self):
        self.sismic_to_capella = defaultdict(lambda: None)

    def _add_name(self, state):
        return {
            'name': state.get_name()
        }

    def _method_by_name(self, method, name: str):
        return getattr(method, name)

    def _build_state(self, owned_region):
        states = []
        # Estado ou Modo
        for owned_state in owned_region.get_owned_states():
            owned_state_obj = {
                'incoming': [],
                'outgoing': [],
                'realized_states': [],
                'realizing_states': [],
            }

            for key in owned_state_obj:
                method_to_call = self._method_by_name(
```

```

        owned_state, 'get_' + key)
    for element in method_to_call():
        owned_state_obj[key].append(element)

    if getattr(owned_state, "get_owned_regions", None):
        owned_state_obj['regions'] = \
            self._build_regions(owned_state)

    owned_state_obj['name'] = owned_state.get_name()
    states.append(owned_state_obj)

    # Mapeio o nome do estado para o elemento do capella
    # e suas transições. Para acessar depois, se necessario.
    self.sismic_to_capella[owned_state_obj['name']] = {
        'state': owned_state
    }

    return states

def _build_regions(self, state_machine):
    regions = []
    # Região da máquina
    for owned_region in state_machine.get_owned_regions():
        owned_region_obj = self._add_name(owned_region)

        states = self._build_state(owned_region)

        if states != []:
            owned_region_obj['states'] = states
            regions.append(owned_region_obj)

    return regions

def _append_state_machines(self, state_machines, capella_state_machines):
    # Máquina de Estado
    for state_machine in capella_state_machines:
        state_machine_obj = self._add_name(state_machine)
        regions = self._build_regions(state_machine)

```

```
        state_machine_obj['regions'] = regions
        state_machines.append(state_machine_obj)

    return state_machines

def _read_capella_state_machines(self, session):
    logical_architecture = session.get_logical_architecture()
    logical_system = logical_architecture.get_logical_system()

    # Inside LC 1
    owned_logical_components = \
        logical_system.get_owned_logical_components()

    state_machines = []

    state_machines = self._append_state_machines(
        state_machines, logical_system.get_owned_state_machines())

    # LC 1 Component
    for owned_logical_component in owned_logical_components:
        state_machines = self._append_state_machines(
            state_machines,
            owned_logical_component.get_owned_state_machines())

    return state_machines

def _dict_to_yaml_str(self, dictionary):
    yaml.dump(dictionary, sys.stdout)
    return yaml.dump(dictionary)

def _parse_standard_dict(self, standard_dict):
    statechart_arr = []

    if isinstance(standard_dict, list):
        return standard_dict

    for key, value in standard_dict.items():
        statechart_dict = {}
        if key[1]:
```

```

        statechart_dict[key[0]] = key[1]

statechart_dict.update(value)
for key_tmp in ['states', 'transitions', 'parallel states']:
    if key_tmp in value:
        statechart_dict[key_tmp] = self._parse_standard_dict(
            value[key_tmp])

statechart_arr.append(statechart_dict)

return statechart_arr

def _handle_empty_states(self, states, handle_states):
    # Handle missing states creating a empty state
    for transition in handle_states:
        if ('name', transition) not in states:
            states[('name', transition)] = {}

    return states

def _region_to_sismic(self, regions):
    parallel_states = {}
    for region in regions:
        initial = None
        states = {}
        parallels = []
        states_transitions = []

        for state in region['states']:
            if initial == None or 'init' in state['name'].lower():
                initial = state['name']

        transitions = {}
        has_parallel = len(state['outgoing']) > 1
        for transition in state['outgoing']:
            targets = transition.get_target()

            for target in targets:
                target_name = target if target == None \

```

```

        else target.get_name()
        transitions[('target', target_name)] = {}
        states_transitions.append(target_name)

        parallel = {
            'incoming': state['name'], 'outgoing': target_name}
        if has_parallel:
            # Solve NonDeterministicError
            parallels.append(parallel)

    if transitions != {} or 'regions' in state:
        states[('name', state['name'])] = {}

    if transitions != {}:
        states[('name', state['name'])]['transitions'] = transitions

    if 'regions' in state and state['regions'] != []:
        parallel_states_2 = self._region_to_sismic(
            regions=state['regions'])
        states[('name', state['name'])]['parallel states'] = \
            self._parse_standard_dict(parallel_states_2)

    states = self._handle_empty_states(states, states_transitions)

    parallel_states[('name', region['name'])] = {
        'states': states
    }

    parallel_states[('name', region['name'])]['initial'] = initial

    return parallel_states

def _capella_to_sismic_state_machines(self, state_machines):
    statecharts = []

    for state_machine in state_machines:
        parallel_states = \
            self._region_to_sismic(regions=state_machine['regions'])

```

```

statecharts.append({
    'name': state_machine['name'],
    'root state': {
        'name': 'root',
        'parallel states': \
            self._parse_standard_dict(parallel_states)
    }
})

return [{
    'statechart': statechart
} for statechart in statecharts]

def step(self, step=None):
    return step

def sessions(self, session=None):
    capella_sms = self._read_capella_state_machines(session)
    sismic_sms = self._capella_to_sismic_state_machines(capella_sms)

    simic_yaml_strs = [self._dict_to_yaml_str(
        sismic_sm) for sismic_sm in sismic_sms]

    # TODO: criar um logger e logar o simic_yaml_str

    return [import_from_yaml(simic_yaml_str) for simic_yaml_str \
        in simic_yaml_strs]

```

A.8 Abstract SM

```

from abc import ABC

class AbstractSM(ABC):
    """
        Abstração de uma máquina de estados
    """

```

```

def build_steps(self):
    # abstract
    pass

```

A.9 SM Factory

```

import sys

from typing_extensions import Self

if "pytest" in sys.modules:
    from simulator.sms.sismic_sm import *
else:
    # SismicParser
    include('workspace://Python4Capella/sample_scripts' \
           '/simulator/sms/sismic_sm.py')

class SMFactory:
    """
    Fábrica de Máquina de Estados
    """

    def __new__(cls, type) -> Self:
        # Fabrica o modelo solicitado
        products = {
            'sismic': SismicSM
        }
        return products[type]()

```

A.10 Sismic SM

```

import sys
from sismic.interpreter import Interpreter
from sismic.io import export_to_plantuml

if "pytest" in sys.modules:

```

```
    from simulator.sms.abs_sm import *
else:
    include('workspace://Python4Capella/sample_scripts' \
           '/simulator/sms/abs_sm.py')

MAX_STEPS = 20

class SismicSM(AbstractSM):
    def build_steps(self, session=None):
        steps = []
        interpreter = Interpreter(session)
        plantuml = export_to_plantuml(session)

        for step in interpreter.execute(max_steps=MAX_STEPS):
            for micro_step in step.steps:
                steps.append(micro_step)

        return {
            'steps': steps,
            'name': interpreter.statechart.name,
            'plantuml': plantuml
        }
```

A.11 Test Simulator

```
import os
import pytest

from simulator.simulator import Simulator
from simulator.tests.mocks.mock_cappella import MockCapellaModel

from pytest_unordered import unordered
from sismic.io import import_from_yaml
```



```
@pytest.fixture
def simulator():
    model = MockCapellaModel()
    return Simulator(model, config={
        'model_path': 'default',
        'state_type': 'sismic',
        'parser_type': 'sismic'
    })

def test_simulator_config(simulator):
    assert simulator.config == {'model_path': 'default',
                               'state_type': 'sismic', 'parser_type': 'sismic'}

def test_sessions(simulator):
    mock_statechart = import_from_yaml(filepath=os.path.dirname(__file__) +
                                       '/mocks/mock.yaml')

    assert len(simulator.state_machine.model.sessions) == 6

    for statechart in simulator.state_machine.model.sessions:
        assert statechart.root == mock_statechart.root
        assert statechart.states == mock_statechart.states
        assert statechart.transitions == unordered(mock_statechart.transitions)
```

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO <p style="text-align: center;">TC</p>	2. DATA <p style="text-align: center;">17 de novembro de 2022</p>	3. DOCUMENTO Nº <p style="text-align: center;">DCTA/ITA/TC 054/2022</p>	4. Nº DE PÁGINAS <p style="text-align: center;">56</p>
5. TÍTULO E SUBTÍTULO: Adding Simulation Capability to Systemic Models			
6. AUTOR(ES): Rubens Miguel Gomes Aguiar			
7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES): Instituto Tecnológico de Aeronáutica – ITA			
8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR: Systemic Engineering; Software Engineering; Capella.			
9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO: Engenharia de sistemas; Simulação; Engenharia de software; Computação; Engenharia aeroespacial.			
10. APRESENTAÇÃO: <input checked="" type="checkbox"/> Nacional <input type="checkbox"/> Internacional ITA, São José dos Campos. Curso de Graduação em Engenharia Aeroespacial. Orientador: Prof. Dr. Christopher Shneider Cerqueira. Publicado em 2022.			
11. RESUMO: <p>In systems engineering, the model-based methodology allows the representation of models at a high level. In this context, models are represented through state machines in order to integrate the different parts of the system. Therefore, the need arises to perform the simulation of state machines in an automated way, to verify the integrity of the created models. Thus, a Python extension was developed for one of the main systems engineering modeling tools, known as Capella, that allows the simulation of statecharts. Therefore, the developed tool was applied in two state machines to exemplify its practical behavior.</p>			
12. GRAU DE SIGILO: <p style="text-align: center;"> <input checked="" type="checkbox"/> OSTENSIVO <input type="checkbox"/> RESERVADO <input type="checkbox"/> SECRETO </p>			