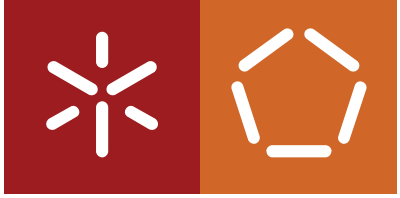
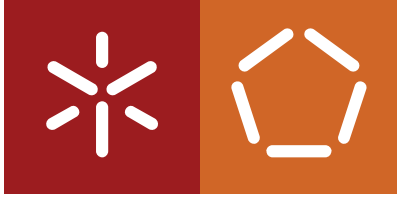


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Afonso João Borges Cabral Cerejeira da Silva

Using Machine Learning to Automatically Infer an Approximation of a Physical System

November 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Afonso João Borges Cabral Cerejeira da Silva

Using Machine Learning to Automatically Infer an Approximation of a Physical System

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
José Nuno Oliveira (U. Minho)
Peter Gorm Larsen (U. Aarhus)

November 2021

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

ACKNOWLEDGEMENTS

I would like to thank my supervisor, José Nuno Oliveira for making this project possible in partnership with the Aarhus University. I am especially grateful for the motivation he conveyed during the final stages of this work. His enthusiasm when teaching formal methods made me choose this field of study for my masters degree. After working in the software industry for 2 years I started sharing his vision for the future of software, and understanding better the need for putting mathematics and software development together.

To Peter Gorm Larsen for kindly receiving me in Aarhus, suggesting this dissertation theme and overall supervision. Without his guidance, patience and encouragement this dissertation would not be possible. Writing a thesis can be lonely, specially when done in a foreign country, so I also thank him for setting me in contact with other researchers and students.

To Hugo Daniel Macedo for his help with the use of the robot and for making me speak Portuguese during my stay abroad.

To the Erasmus+ mobility program for allowing thousands of students like me to live a wonderful experience abroad. I am also thankful to my roommates at Børglum Kollegiet for their hospitality and sympathy.

And finally a personal thanks to my parents, João and Ana Paula, my sisters Mafalda and Sofia, and especially to Catarina for being an inspiration for me and never allowing me to give up on this dissertation, giving me always their full support. To you I dedicate this dissertation.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

The development of Cyber-physical Systems (CPSs) models is a complex process which requires deep multi–disciplinary knowledge of the intended topic to model. Added to this complexity is the difficulty of combining multiple models, sometimes without access to their source code, and make them communicate in a harmonious and integrated way in order to represent the vicissitudes of the environment where the physical system is inserted into. Functional Mockup Interface is a set of C headers that define a protocol that allows the interoperability of different models, independently of the programming languages and tools that generated them. A model that implements this interface is called Functional Mockup Unit (FMU).

This dissertation explores the usage of Machine Learning to generate automatically a FMU from parsing a dataset containing the inputs and outputs obtained during the observation of a physical system. A Command-line Interface (CLI) tool named AutoFMU is also presented here, and it accepts as parameters a set of CSV tables and the names of the column that correspond to the inputs and outputs, using several supervised learning algorithms to infer the relationships between these variables. Its invocation results in a file containing a valid FMU ready to be used.

In order to assess its feasibility in a real context, the tool AutoFMU was used to generate approximations of a controller of a line follower robot. The generated models were then simulated in the INTO-CPS program and the robot movements under the purview of the new controller were observed. The values generated by the new models were also compared with the datasets of the original physical unit.

KEYWORDS Cyber-physical System, Functional Mockup Interface, Machine Learning, Python

RESUMO

O desenvolvimento de modelos de sistemas ciber-físicos é um processo complexo que exige profundos conhecimentos multi–disciplinares do tópico que se pretende modelar. A esta complexidade acresce ainda a dificuldade de combinar múltiplos modelos, por vezes sem acesso ao seu código fonte, e fazê-los comunicar de uma forma harmoniosa e integrada de forma a representar as vicissitudes do ambiente onde o sistema físico se insere. A Functional Mockup Interface é um conjunto de cabeçalhos C que define um protocolo comum que permite a interoperabilidade de diferentes modelos, independente das linguagens de programação e ferramentas que os geraram. Um modelo que implementa esta interface é chamado de FMU.

Esta dissertação explora a utilização de Machine Learning para gerar automaticamente um FMU a partir da análise de um conjunto de dados contendo os *inputs* e *outputs* obtidos durante a observação de um sistema físico. Apresenta-se também uma ferramenta de linha de comandos de nome AutoFMU que aceita como parâmetros um conjunto de tabelas CSV e os nomes das colunas que correspondem aos *inputs* e *outputs*, utilizando diversos algoritmos de aprendizagem supervisionada para deduzir as relações entre estas variáveis. Da sua invocação resulta um ficheiro que contém um FMU válido pronto a ser utilizado.

De forma a avaliar a sua viabilidade num contexto real, a ferramenta AutoFMU foi utilizada para gerar aproximações de um controlador de um robot que segue uma linha desenhada no chão. Os modelos gerados foram depois simulados no programa INTO-CPS tendo-se observado e comparado os movimentos efetuados pelo robot sob a alçada do novo controlador. Os valores gerados pelos novos modelos foram também comparados com os *datasets* da unidade física original.

PALAVRAS-CHAVE Sistemas ciber-físicos, Functional Mockup Interface, Machine Learning, Python

CONTENTS

Contents iii

1 INTRODUCTION 1
- 1.1 Motivation . . . 1
- 1.2 Aims . . . 2
- 1.3 Contribution . . . 3
- 1.4 Document structure . . . 3

2 STATE OF THE ART 4
- 2.1 Cyber–Physical Systems . . . 4
  - 2.1.1 Background . . . 4
  - 2.1.2 Formal Methods in the development of Cyber–Physical Systems . . . 5
  - 2.1.3 Modeling Cyber–Physical Systems . . . 6
  - 2.1.4 The Functional Mockup Interface . . . 7
- 2.2 Machine Learning and Big Data analytics . . . 8
  - 2.2.1 Applications . . . 8
  - 2.2.2 Approaches . . . 8
  - 2.2.3 Machine Learning and Cyber–Physical Systems . . . 9
- 2.3 Summary . . . 10

3 DEDUCING AN APPROXIMATION OF A CYBER-PHYSICAL SYSTEM 11
- 3.1 Motivation . . . 11
- 3.2 Goal . . . 12
- 3.3 Challenges . . . 12
  - 3.3.1 Data analysis and statistical modelling . . . 12
  - 3.3.2 Generating FMU source code . . . 13
  - 3.3.3 Compiling the FMU . . . 18
  - 3.3.4 Testing and evaluating the generated FMU . . . 21
- 3.4 AutoFMU . . . 21
  - 3.4.1 Overview . . . 21
  - 3.4.2 Architecture . . . 23
  - 3.4.3 Development . . . 24
- 3.5 Summary . . . 26

4 CASE STUDY: A LINE FOLLOWER ROBOT 28
- 4.1 The line follower robot . . . 28
- 4.2 Modelling the line follower robot . . . 30
- 4.3 Simulating the line follower robot movement . . . 31
- 4.4 Using AutoFMU to approximate a component of the line follower robot . . . 33

4.5 Testing the generated components . . . 34
4.6 Results analysis . . . 35
4.7 Summary . . . 38
5 CONCLUSIONS AND FUTURE WORK 39
5.1 Conclusions . . . 39
5.2 Prospect for future work . . . 40

Bibliography 42

I APPENDICES

A AUTOFMU REFERENCE MANUAL 50
A.1 User guide . . . 50
A.1.1 Installation . . . 50
A.1.2 Usage . . . 51
A.2 API Reference . . . 51
A.2.1 autofmu . . . 51
A.3 Command Line Reference . . . 53
A.3.1 Positional Arguments . . . 53
A.3.2 Named Arguments . . . 53
A.4 License . . . 54

LIST OF FIGURES

Figure 1 Screenshot of INTO-CPS project window 6
Figure 2 High level overview of the approximation tool inputs and outputs 12
Figure 3 Screenshot of the FMU builder website 20
Figure 4 FMU comparison process pipeline 21
Figure 5 Overview of AutoFMU pipeline from data analysis to FMU compilation . 23
Figure 6 Screenshot of the documentation website 26
Figure 7 Screenshot of the code coverage analysis website 27
Figure 8 The line follower robot used in this work 29
Figure 9 Example map that contains the path for the robot to follow 29
Figure 10 SysML diagram that shows the interactions between the robot FMUs . . 30
Figure 11 Path followed by the robot in the model simulation 34
Figure 12 Robot movement resulting from different approximation algorithms . . . 36
Figure 13 Servo values variation over time using the linear regression approximation 37
Figure 14 Servo values variation over time using the logistic regression approximation 38

LIST OF TABLES

Table 1 Example Comma-separated Values (CSV) table containing an AutoFMU dataset. 22
Table 2 Variable classification by FMU 31
Table 3 Excerpt from the table of results from the simulation 33
Table 4 r^2 and mean squared error scores for each approximation strategy . . . 36

LIST OF LISTINGS

3.1 Relationship code for a linear regression strategy 14
3.2 Relationship code for a logistic regression strategy 15
3.3 Template to generate the relationships C code 16
3.4 Declaration of the “VARIABLES” buffer 17
3.5 Implementation of the “fmi2GetReal” function 17
3.6 Implementation of the “fmi2SetReal“ function 18
3.7 Installation with Git 22
4.1 Launching the Co-orchestration Engine (COE) server 32
4.2 Creating a session in the COE server 32
4.3 Simulating the multi-model with the COE 32
4.4 Destroying a session in the COE server 32
4.5 Invoking AutoFMU with a linear regression strategy 33
4.6 Invoking AutoFMU with a logistic regression strategy 33
4.7 Running FMU Compliance Checker on the generated FMU 34
4.8 Bash script for creating multi-models for the approximated FMUs 35

GLOSSARY

Arduino An open-source electronic prototyping platform that enables users to create interactive electronic objects. 28

AutoFMU The program developed in this work that uses machine learning to analyze a dataset and produce aN FMU. v–vii, c, d, 11, 21–25, 33, 34, 38–41, 50

C A general-purpose, procedural computer programming language. vii, c, d, 11, 13, 14, 16–19, 23, 39, 40, 50, 52

CMake A cross-platform tool for build automation, testing, packaging and installation of software. 19, 23, 52, 53

cURL A command-line tool for getting or sending data including files using URL syntax. 32

Docker An open source platform for building, deploying, and managing containerized applications. 20, 50

formal methods Techniques based on mathematical formalism for developing, specifying and verifying software and hardware systems. 3, 5–7

Git A free and open source distributed version control system. vii, 22, 50

Java A high-level, class-based, object-oriented programming language. 31

Make A tool that builds executable programs and libraries from source code by reading Makefiles. 19

Makefile A special file containing shell commands for building a project. 19

Python An interpreted high-level general-purpose programming language. c, d, 21, 23–26, 40

scikit-learn A machine learning library for Python. 23

UniFMU A tool that makes it possible to implement FMUs in any language. 40

YAML A human-readable data-serialization language. 25

ACRONYMS

AGC Apollo Guidance Computer. 4
API Application Programming Interface. 7, 32

CD Continuous Delivery. 25
CI Continuous Integration. 25, 26
CLI Command-line Interface. c, 11, 12, 21, 32, 40
COE Co-orchestration Engine. vii, 7, 21, 31, 32, 35
CPS Cyber-physical System. c, 1, 3–11, 39
CSV Comma-separated Values. vi, c, d, 2, 21, 22, 32, 33, 35, 53

DLL Dynamic-link Library. 7, 19

FMI Functional Mockup Interface. c, d, 1, 2, 7, 11, 13, 17, 21, 26, 30, 34, 39, 40, 52
FMU Functional Mockup Unit. v–vii, c, d, 1–3, 7, 11–13, 17, 19–23, 26, 30–35, 38–40, 50–53

GCC GNU Compiler Collection. 19, 50
GUI Graphical User Interface. 7, 31, 32

IFDSE Integrated Formal Development Support Environments. 6
INTO-CPS Integrated Tool Chain for Model-based Design of Cyber-Physical Systems. v, c, d, 6, 7, 19, 20, 30–32

JSON JavaScript Object Notation. 32, 35

MDP Markov Decision Processes. 9
MinGW Minimalist GNU for Windows. 19, 50, 52
ML Machine Learning. c, d, 2–4, 8–12, 14, 23, 26, 39, 40
MOS metal-oxide-silicon. 4
MSE Mean Squared Error. 36, 37

OEM Original Equipment Manufacturers. 7

pip Package Installer for Python. 22, 23, 40, 50
PyPI Python Package Index. 22, 50

REST Representational State Transfer. 7
RST reStructuredText. 25

SVM Support Vector Machines. 40
SysML Systems Modeling Language. v, 6, 30

VDM Vienna Development Method. 7, 30, 33

XML Extensible Markup Language. 7, 52

1

INTRODUCTION

> Any component of an agent can be improved by learning from data
>
> Russell, Norvig, and Davis (2010)

1.1 MOTIVATION

CPSs are physical engineered devices that operate within a digital context being capable of communicating with other devices and sharing data with other systems (Lee, 2008). By interacting with the physical world, CPSs are often equipped with sensors that collect large amounts of data that can be analysed to provide interesting insights on the environment the CPS operates on (Jazdi, 2014).

Some CPSs are expensive to build, and errors in programming them can have catastrophic consequences, particularly if those are safety-critical systems and human lives depend on them (Knight, 2002). For this reason, CPSs development relies heavily on abstract software models that can be simulated using appropriate tools. This approach has several benefits, as Beydeda, Book, and Gruhn (2005) write: "models provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones".

When modelling complex CPSs it can be useful to split the system into smaller components, simpler to reason independently. Each of these can be considered a separate model that receives some input data, performs an action with that data, and produces output values to pass to other models. The coordination and assembly of a set of different models is called a co-simulation, as defined by Gomes et al. (2018): "Co-simulation consists of the theory and techniques to enable global simulation of a coupled system via the composition of simulators. Each simulator is broadly defined as a black box capable of exhibiting behaviour, consuming inputs and producing outputs". For these different models to be able to interact with each other they all need to implement the same protocol. The Functional Mockup Interface (FMI) defines, among other features, the set of inputs and outputs that a model can deal with. A model that implements such an interface is called a FMU (Blockwitz et al., 2012).

Developing a model for an existing physical device can prove to be a challenging task, especially if there is no access to the device source code or user manual. A careful observation of the device behavior is thus required in order to be able to create a model that somehow resembles the original one, requiring a plethora of good reverse engineering techniques and skills to successfully achieve the desired results (Samuelson and Scotchmer, 2002). This situation gets even more complicated when working in a multi-model project, as it is important to make sure that each model interacts properly with others. As an example, consider the following scenario:

1. A new device is to be developed involving multiple components, one of which is made by a third-party entity;
2. A co-simulation project is created and a FMU is defined for each device component to better test how they interact together;
3. However, the third-party component is proprietary and the source code is not available. Its behavior is also difficult to grasp and formalize into a model, thus making reverse engineering very expensive.

The main motivation for this work lies in scenarios of this kind, as it would be very useful to be able to automatically generate a FMU for a physical device based solely on the way it maps the inputs received to the outputs produced.

Multiple statistical modelling and Machine Learning (ML) techniques exist for inferring the relationship between FMU input and output data. In this case the values consumed and produced by a physical device can be used as training data, consisting of a list of pairs that map an input to an output. This approximation approach is called *supervised learning*, and according to Mohri, Rostamizadeh, and Talwalkar (2018) it is "the most common scenario associated with classification, regression, and ranking problems", being that "the learner receives a set of labeled examples as training data and makes predictions for all unseen points". The resulting prediction would consist on the algorithm that dictates the FMU behavior, and the relation inferred from the inputs and outputs could then be translated into real code, thus generating a truly valid FMU.

1.2 AIMS

This dissertation aims the development of a program that reads CSV files that contain a list of input and output values, and uses supervised learning techniques to find the relationships between them. Having found that relationship, the program should generate valid FMU source code that defines the proper behavior of the model. To finish the FMU generation, the program will also compile the code and build the appropriate binaries, so that the FMU is ready to be used in a simulation by other programs.

Particularly during the development of this work it is intended to answer the following questions:

1. Is it possible for a program to generate valid FMUs that correctly implement the Functional Mockup Interface?

2. How human readable can the code generated by ML algorithms be?
3. How does an approximated FMU perform in a multi-model environment, *i.e,* how well does it interact with other models when running a simulation?
4. How accurate are supervised learning results for deducing the relationships between a set of input and outputs?

1.3 CONTRIBUTION

The main outcome of this work is the source code of the program that deduces approximations of a FMU based on a given dataset containing the input and output values. This open-source program should be easy to install and provide proper user documentation, describing its configuration and usage. Firther to the program distribution, this dissertation outlines the following contributions:

- evaluation of the performance of the FMUs generated by the program;
- comparison between different ML algorithms when deducing a model approximation;
- usage of the program to generate an approximation model of a real world CPS unit.

1.4 DOCUMENT STRUCTURE

The remainder of this dissertation is structured as follows:

- Chapter 2 gives an overview of the state of the art regarding CPSs modelling techniques and tools, and how formal methods can be employed for that purpose. This chapter also encompasses a brief survey on current ML development and its usage in data intensive software projects.
- Chapter 3 explains in detail the process used to create a program able to approximate a FMU of a CPS from a given dataset, using different machine learning algorithms.
- Chapter 4 uses the program created in Chapter 3 to deduce a model of a component of a real physical unit of a line follower robot. The generated model is then used in a multi-model simulation in order to compare its behavior with the original physical unit.
- Chapter 5 discusses the results obtained in Chapter 4 and finishes this document with some prospect of future work.

2

STATE OF THE ART

CPSs are quickly changing the world as we know it, filling the gap between the physical and the digital domains. Advances in hardware manufacturing and the need for connected smart devices allow for the development of new digital systems that seamlessly interact with the physical world. On the other hand there has been solid development in ML and statistical analysis that make for a better understanding of the huge amount of data produced by CPSs.

This chapter is split into two different sections. The first one presents the current state of the art of CPSs, their usefulness and how to model them. The last one provides a brief survey on current Machine Learning development focusing primarily on its usage for reverse engineering projects.

2.1 CYBER–PHYSICAL SYSTEMS

Monostori et al. (2016) defines CPSs as "systems of collaborating computational entities which are in intensive connection with the surrounding physical world and its on–going processes, providing and using, at the same time, data–accessing and data-processing services available on the Internet". In short, a CPS corresponds to a physical engineered device that is integrated in a digital system.

2.1.1 *Background*

Digital systems that interact with the physical world via electronic or mechanical devices are nothing new: in fact, the so-called *embedded systems* have been a presence in industry since the dawn of microprocessors and microcontrollers and their usage is as old as the development of the metal-oxide-silicon (MOS) integrated circuit in the 1960s (Gregorian and Temes, 1986). The development of the Apollo Guidance Computer (AGC) for the Apollo project in 1965 is regarded as a major achievement in the field. The AGC controlled the command, service and lunar modules of the Apollo XI mission, operating on a set of strict requirements and on very limited hardware (O'Brien, 2010). The success of this mission resulted in the widespread of embedded systems to other areas of knowledge which led to gradual improvements in their capabilities and design until today.

Almost 50 years later the world is no longer the same and the way human beings interact with digital systems is now totally different. The internet has reached everywhere, causing a huge demand for devices capable of communicating with each other from anywhere in the globe. In

fact, economists such as Schwab (2017) argue that today we are in an era of unprecedented digital transformation, the so-called Industry 4.0, or fourth industrial revolution, that will bring huge improvements to industrial operations and production efficiency, all of this thanks to the large scale employment of CPSs.

It is in this context that CPSs come into play, as a response to the need to develop networked devices capable of analyzing and interacting with the real world. Such devices communicate with each other following standard communication protocols across which they are able to share data and calculations based on the readings from their physical sensors, forming a mesh commonly called "Internet of Things" (Gubbi et al., 2013).

As they interact with the vicissitudes of the real world, CPSs are used in a wide range of different contexts. As stated by Shi et al. (2011), "applications of CPSs include medical devices and systems, assisted living, traffic control and safety, advanced automotive systems, process control, energy conservation, environmental control avionics and aviation software, instrumentation, critical infrastructure (*e.g.* power, water), distributed robotics, weapons systems, manufacturing, distributed sensing command and control, smart structures, biosystems, communications systems, *etc.*". This is a very comprehensive set of applications, and it is possible to acknowledge that CPSs usage is in general tied with the development of "distributed real-time embedded systems" that "interact with each other in a very complex manner" (Kim and P. R. Kumar, 2012).

2.1.2 *Formal Methods in the development of Cyber–Physical Systems*

As a result of a multidisciplinary effort, systems that interact with the uncertainty of the physical world require a holistic development process based on the rigor of the mathematical knowledge (Rajkumar et al., 2010). As the complexity of the system grows, so does the challenges of reasoning about it, yet according to Wolf (2009) "we have a surprisingly small amount of theory to tell us how to design computer–based control systems". How is it possible then to ensure that the system will behave as it should when dealing with the volatility of the real world?

It is within this complex scenario that formal methods come to rescue. As it turns out, CPSs development is a perfect example for the need of application of formal methods techniques, as will be shown below.

Formal methods are a set of techniques with sound mathematical basis used for rigorously describing the properties of a system, usually defining the semantics and syntax of a specification format to write such properties (Wing, 1990). Widely employed in the development of safety-critical and security-critical systems, the application of these techniques helps ensuring the correctness of a system against a set of well defined specifications. The process of gathering and formally describing the system requirements prevents a whole class of bugs that normally would only be found in the testing phase of the project, as demonstrated by Clarke and Wing work, where the quality of a handful of real–world world projects was greatly improved by employing formal methods techniques.

It is possible to employ formal methods for modelling both hardware and software systems and there exist already Integrated Formal Development Support Environments (IFDSE) that combine a plethora of different tools for project development in this field (Bowen and Hinchey, 1995). In this work, Integrated Tool Chain for Model-based Design of Cyber-Physical Systems (INTO-CPS) will be used as the main IFDSE for modelling a line follower robot, as will be explained in Chapter 4. The tool, *per se*, will be further analyzed and discussed in Section 2.1.3.

2.1.3 *Modeling Cyber–Physical Systems*

Modeling CPSs can prove to be a challenging task, as they operate in a non-controlled environment and are expected to handle unpredictable conditions and adapt do subsystem failures (Lee, 2008). Different techniques and tools exist to model devices of this kind, and for this work we will focus mainly on INTO-CPS usage, as it integrates multiple modeling tools.

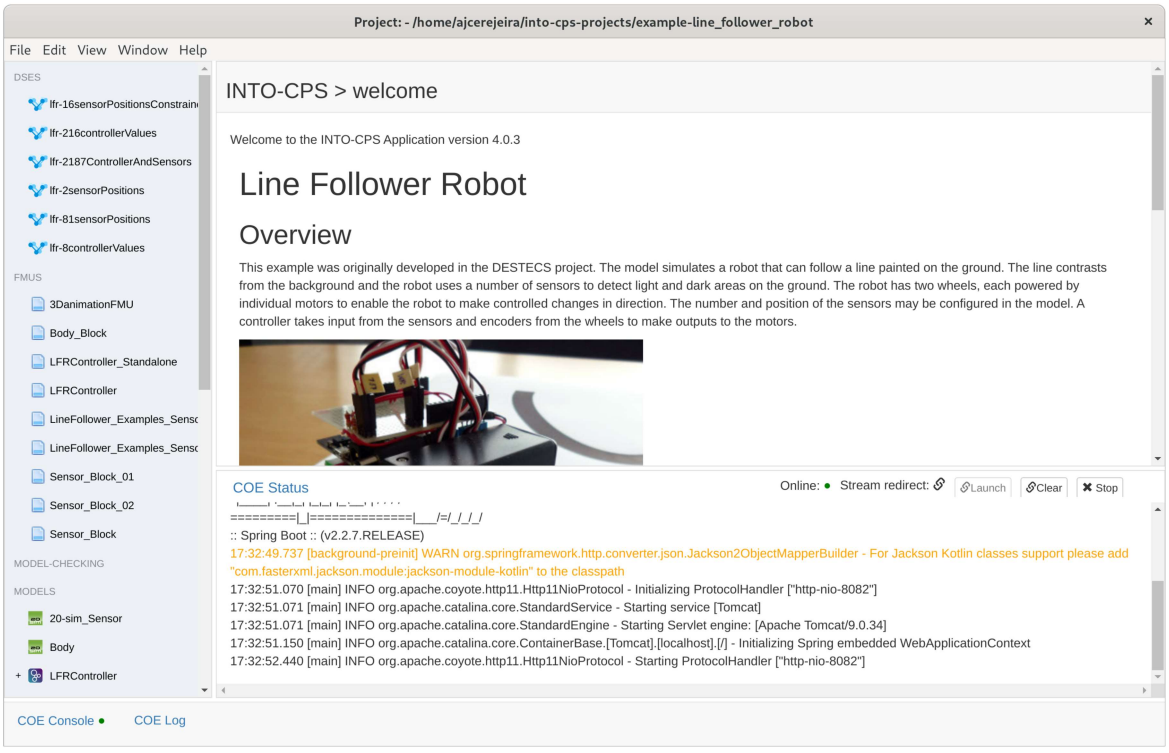


Figure 1: Screenshot of INTO-CPS project window

INTO-CPS is a IFDSE that aggregates other tools allowing for a multidisciplinary model development during all project phases, from requirements gathering to code implementation (Larsen, Fitzgerald, et al., 2016). The tools supported by INTO-CPS are described as follows:

- *Modelio* — a modeling tool that supports the Systems Modeling Language (SysML), allowing designers to “simultaneously depict and specify several aspects of the system from requirements to the hardware/software architecture through use case specification, and system functional design” (Bagnato et al., 2016).

- *Overture* — a framework "built on an open and extensible platform based on the Eclipse framework" that integrates a "range of tools for constructing and analysing formal methods of systems using the Vienna Development Method (VDM)", including the formal language VDM++ for specifying and analysing system models (Larsen, Battle, et al., 2010).
- *20–sim* — "a tool for modeling and simulation of dynamic behavior of engineering systems (...) that span multiple physical domains and the information domain" (Broenink, 1999).
- *OpenModelica* — "a modern, strongly typed, declarative, and object–oriented language for modeling and simulation of complex systems" (Fritzson et al., 2006).

Another key aspect of INTO-CPS development is the COE that allows for multiple FMUs to be coupled in a full system simulation (Thule et al., 2019). In this particular case, the COE also provides a REST API which allows for running simulations without a Graphical User Interface (GUI) in an automated and scripted manner.

2.1.4 *The Functional Mockup Interface*

As explained in the previous section (2.1.3) there exists a wide array of different tools to help modelling CPSs. For this reason, it is of utmost importance for the models generated by these tools to be able to be independently inter-exchangeable. To standardize tool independent exchange of dynamic models and allow for co–simulation scenarios, the German automotive company, Daimler AG, developed the Functional Mockup Interface (Blochwitz et al., 2011; Blockwitz et al., 2012), which was early adopted and supported by multiple Original Equipment Manufacturers (OEM) (Bertsch, Ahle, and Schulmeister, 2014).

This standard defines the structure for distributing models for two different scenarios, which are both described by Blockwitz et al.:

- *Model exchange* consists of "a dynamic system model in the form of an input/output block (...) that can be utilized by other modeling and simulation environments".
- *Co–simulation* is an environment where two or more models are coupled with solvers to exchange data through a restricted set of communication points.

A FMU, *i.e,* a model that implements the FMI, consists of a zipped directory (with the `.fmu` file extension) that contains the following components (Blochwitz et al., 2011):

- A XML file named `modelDescription.xml` that defines the metadata associated with the model (name, authors, description, *etc.*) as well as the definition of the model inputs and outputs. It also includes the parameter settings for model exchange and/or co–simulation.
- A `binaries/` directory that contains, as the name implies, the set of binaries for each platform that the model supports. For Windows systems this means Dynamic-link Library (DLL) files, and for Linux and MacOS, shared-object library files. The libraries under this directory must correctly implement the functions defined by the FMI header files.

- Optional extra directories containing the model source code, documentation, resources, or any other data that the model depends on.

2.2 MACHINE LEARNING AND BIG DATA ANALYTICS

Much has been written and studied recently about ML and recent advances in the field have significantly improved the way we approach data analysis and prediction making. This area of study is, however, older than it looks like: in fact, the term Machine Learning was coined by Arthur L. Samuel in the 50s, thus making him a pioneer in this field (McCarthy and Feigenbaum, 1990). Samuel successfully applied ML techniques when developing a self–learning program for playing the famous checkers game, proving the usefulness and applicability of this area of research (Samuel, 1959). Considered a part of the artificial intelligence field, ML empowers digital systems to be intelligent, in the way that they have the ability to learn and adapt to environment changes that were not foreseen by the system designer (Alpaydin, 2010, Chapter 1, pp. 3). As discussed in Section 2.1, CPSs are built with the intention of being exposed to external agents, expecting them to be resilient against environment changes. Based on the previous statement it is fair to conclude that CPSs development can be improved by the application of ML techniques resulting in systems better adapted to behave properly in unexpected situations.

2.2.1 *Applications*

ML has a wide array of applications, and it is fair to say that it can be used wherever data mining and data analytics are needed. ML can be seen as a set of techniques for performing data pattern recognition, information extraction and predictions (Ge et al., 2017a). Being such a general topic with deep roots in statistical modelling, it is virtually impossible to enumerate the different types of applications that ML can be used for. Among them, usage of ML in the development of autonomous vehicles (Janai et al., 2020), medical image analysis (Ge et al., 2017b) and natural language processing (Olsson, 2009) stand out as stellar examples of the application of these techniques.

2.2.2 *Approaches*

There are multiple learning approaches to take into account when working on big data analysis. Usually, they can be classified as either *supervised*, *unsupervised* or *reinforced*. The next paragraphs describe briefly the differences between such approaches and their principal use cases. It is important to note that different learning algorithms produce different results with varied performance, depending on the method calibration and parameters, as well as the context of the data being analyzed (Caruana and Niculescu-Mizil, 2006).

Supervised learning

Supervised learning consists in a set of algorithms for deducing the relationship between sets of inputs and outputs. The data that contains these values is called training data, providing a set of examples where each one contains one or more inputs and the value for the desired output. Mathematically, each example can be represented by a vector, which when coupled with the remaining examples form the training matrix. In order to better estimate the output value for a new input, supervised learning algorithms work in a iterative way by optimizing an objective function to better fit the model, using regression and classification techniques (Ghahramani, 2003).

Unsupervised learning

Contrary to the previous technique, in *unsupervised learning* the machine knows nothing about the target outputs and does not receive any environment rewards. According to Ghahramani (2003) this model can be used for "finding patterns in the data above and beyond what would be considered pure unstructured noise". It is therefore a technique useful for clustering analysis, where the goal is to "find similarities in the training data" based on a large amount of information. This can be useful when there is no previous knowledge on the relations among the features of the dataset (Ayodele, 2010).

Reinforcement learning

Different from the previous two approaches, *reinforcement learning* assumes little to no knowledge about the environment where the model is placed. Instead, it defines a function that decides, based on the reading of an environment state, if the model should or not be rewarded. The environment is often represented as a Markov Decision Processes (MDP) and *reinforcement learning* algorithms typically rely on dynamic programming techniques, aiming to optimize the decisions taken by the model (Otterlo and Wiering, 2012).

Regarding similar projects that combine both ML and CPS development, the master thesis by Neves (2021) contains interesting work on modelling a line following robot, similar to the one studied in Chapter 4. A set of patterns reward the robot's behavior when its movement correctly follows a line drawn on the floor (Neves, 2021). Although sharing some similarities, the approach by Neves provides a solution to a problem different from what is analyzed here: instead of providing a reward function to infer a model, this work takes instead a supervised learning approach, by providing *a priori* all the training data required to approximate a model.

2.2.3 *Machine Learning and Cyber–Physical Systems*

Usually operating in real–time scenarios and interacting with external elements and other systems, CPSs end up collecting huge amounts of information that needs to be further analyzed (Marwedel, 2021, Chapter 1, pp. 15). This makes for a strong case for the purpose of this dissertation: by

analyzing such huge chunks of data it will be possible to get a better understanding of the system behavior rules.

2.3 SUMMARY

This chapter was devoted to reviewing the state of the art of two distinct areas of software engineering: CPSs and Machine Learning. These fields are vast on their own, providing a wide array of topics and themes to study. The remainder of this dissertation will address both areas of study. In particular, the development of a solution for producing CPSs using ML will be thoroughly discussed in Chapter 3.

3

DEDUCING AN APPROXIMATION OF A CYBER-PHYSICAL SYSTEM

As reflected in the title of this dissertation, the main purpose of the work it reports is to study and understand how ML can be used to infer an approximation of a CPSs.

This chapter shows the process of developing an open source CLI program named AutoFMU which generates an approximation of a CPS by analyzing a tabular dataset of input and output results. The whole development process is hereby described from the planning and architecture stages to the implementation phase, exposing in detail the techniques and technologies used to successfully build this program.

3.1 MOTIVATION

Section 2.1 of the previous chapter explained that modelling CPSs is a multidisciplinary effort that requires tools of different kinds, depending on the component of the system that one intends to model. To allow easy communication and integration between the models, regardless of the tool used, they all implement a common FMI.

Assuming there is a CPS without any documentation or model of its behavior and structure, in order to produce a valid multi-model of this system it would be necessary to study extensively the device behavior, that is, to re-engineer it. This process would be manual and time consuming, depending on the skills of the person analyzing the system and therefore susceptible to human mistakes inherent in reverse engineering processes (Chikofsky and Cross, 1990).

Another problem arises when distributing FMU whose source code is supposed to be kept private: even if not including the C source files in the FMU, it is still possible to disassemble the binaries and thus reconstruct the original code. A program that generates an FMU based on patterns discovered by ML algorithms could also work as a source code obfuscation tool, since the generated code would only contain the mappings between the inputs and outputs of the FMU, being incomprehensible for anyone trying to disassemble it (Collberg and Thomborson, 2002).

This chapter proposes an approach to solve these problems, using the techniques previously explained in Section 2.2 to generate an FMU that approximates the behavior of a CPS relying on large sets of timestamped data produced by the physical system.

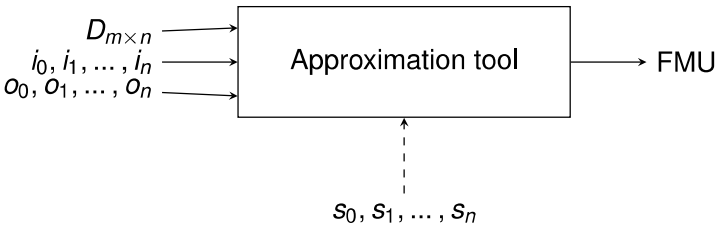


Figure 2: High level overview of the approximation tool inputs and outputs

3.2 GOAL

The main artifact resulting from this work is an approximation CLI program able to analyze multiple datasets containing collected, timestamped data of a system and infer an approximated FMU.

Figure 2 shows a high level overview of the approximation tool and its expected inputs and outputs, namely:

- $D_{m\times n}$ is the dataset, represented as $m \times n$ matrix;
- $i_0, i_1, \ldots, i_n$ are the parameters that the program will take into account, i.e., the inputs of the FMU;
- $o_0, o_1, \ldots, o_n$ are the variables to predict based on the requested parameters, i.e. the outputs of the FMU;
- $s_0, s_1, \ldots, s_n$ are the strategies used to deduce the approximation of the FMU. For some cases a simple linear regression strategy is enough, but for complex models it may be worth to use a full ML strategy.

3.3 CHALLENGES

To the best of the author's knowledge, at the time of writing the approach hereby described has never been attempted before, meaning that the results that will be obtained are rather unpredictable. It is plausible to anticipate that it may be impossible for such a tool to achieve a satisfactory result at the level of automatic FMU generation. On the other hand, could it be the case that such an automated tool performs better than its handwritten counterpart? It is then fair to assume that there are some challenges that must be addressed in order to produce a high quality and reliable FMU generation tool.

3.3.1 *Data analysis and statistical modelling*

The biggest challenges when building an FMU approximation tool are the data analysis and statistical modelling steps. Intended as a generic tool, the program developed in this work will accept any type of tabular dataset. The origin and context of the data will be indifferent to its

functioning, since their sole purpose is to deduce the relationships between the different columns of the dataset. Intended as a completely automated process, its results will not benefit from any kind of manual exploratory analysis of the data. This may prove to be a challenge for the program, as such and analysis would allow for a better understanding of the characteristics of the data and their vicissitudes (Tukey, 1962).

The cleaning and preparation of the dataset is another challenge that arises in the development of programs of this kind. Once again, as it is an automatic process, it is very complicated for this tool to understand the characteristics of the data it is analyzing. For example, the detection of outliers is usually a manual and human process (Almeida et al., 2007), and automating that may or may not compromise the results obtained. Another problem with this automatic approach is related to the classification of variables. For instance, when analyzing a column in the dataset, the program has no way of knowing whether its data are discrete or continuous. For this reason, it is impossible to automatically choose the best approximation algorithm, as different algorithms might be best suited for different kinds of data. In that way it is important to allow the user to select the algorithm strategy that better suits the data.

3.3.2 *Generating FMU source code*

The approximation tool must generate valid C source files that correctly implement the functions defined in `fmi2Functions.h` header file. To do this effectively it is necessary to deeply understand how the generated output should look like, in this case a FMU C source file. In his 2008 book "*Domain-specific modeling: enabling full code generation*", Kelly and Tolvanen suggest that "the best way to do that is to have a working example of the output" (Kelly and Tolvanen, 2008, Chapter 11, pp. 268). For this specific domain the desired output is already known: a valid C program that follows the FMI standard.

Translating relationships to C functions

To generate valid FMU code it is necessary to convert the relationship R, calculated during the data analysis and statistical modelling of the datasets into an adequately C function that map the relationship inputs with its outputs. Therefore, for an approximation of a model with n inputs and m outputs R can be defined as:

$$I \xrightarrow{R} O = \{I \xrightarrow{R_0} O_0,\ I \xrightarrow{R_1} O_1,\ \ldots,\ I \xrightarrow{R_m} O_m\}$$

Translating this relationships to C code is a matter of generating a function R whose parameters are the reference to the output variable to calculate and the array of input values. The return value should correspond to the calculated output value for the specified inputs. Given that in C arrays do not hold the number of elements and capacity information (Kernighan et al., 1988, Chapter 5, pp. 97–100), it is also necessary to pass the number of outputs and inputs to the function. The signature for this function would be as follow where `o` is the value reference for the output to predict,

`x` is the array that contains the input values, `n` is the number of inputs and `m` is the number of outputs:

```
fmi2Real R(fmi2ValueReference o, fmi2Real x[], size_t n, size_t m);
```

The chosen strategy during the data analysis and statistical modelling step will define how the body of the function *R* is generated. Different ML algorithms will produce different C functions since each one of them uses a different algorithm for prediction. In the program developed here, the approximation strategies considered were both linear and logistic regression. The reason for this is due to the fact that these algorithms are more easily mapped in mathematical prediction functions. Other strategies were also considered (such as support vector machines, *k* nearest neighbors, neural networks, *etc.*), however their use was discarded due to the complexity of translating the prediction models to C code. The detailed explanation and code translation for the selected strategies is described in the next paragraphs.

LINEAR REGRESSION A linear regression strategy produces a vector β_{n+1} for each output variable, where n is the number of inputs, representing the set of coefficients to multiply by each input. The sum of the coefficients by the respective inputs produces the approximation value for the desired output where β_0 is the intercept term and is not multiplied by any input (Freedman, 2009, Chapter 4, p. 42). Therefore it is possible to predict the value for each specified output based on the input variables with the following equations:

$$\begin{aligned} y_0 &= \beta_{0,0} + \beta_{0,1} * x_0 + \cdots + \beta_{0,n+1} * x_n \\ &\cdots \\ y_m &= \beta_{m,0} + \beta_{m,1} * x_0 + \cdots + \beta_{m,n+1} * x_n \end{aligned} \tag{3.1}$$

```
#define n NINPUTS
#define m NOUTPUTS

void linear_regression(const double x[], double y[]) {
    const double b[m][n] = {
        { b_0_0, ..., b_0_n },
        ...
        { b_m_0, ..., b_m_n }
    };

    y[0] = b[0][0] + b[0][1] * x[0] + ... + b[0][n] * x[n - 1];
    ...
    y[m] = b[m][0] + b[m][1] * x[0] + ... + b[m][n] * x[n - 1];
}
```

Listing 3.1: Relationship code for a linear regression strategy

LOGISTIC REGRESSION A logistic regression strategy produces a matrix $\beta_{o \times n+1}$ for each output variable, where o is the number of possible outcomes and n is the number of inputs. The

outcomes vector, k_o, contains all the possible values for each y output. Before calculating the value of each output variable it is required to build the score matrix P_o that contains the probabilities of that specific outcome being chosen for a given input. This matrix is calculated by applying the dot product between the coefficient matrix (β) and the set of inputs X_m (Hosmer Jr., Lemeshow, and Sturdivant, 2013, Chapter 2, pp. 37–42), as demonstrated by the following equation:

$$score(X_i, k) = \beta_k \cdot X_i \tag{3.2}$$

Having calculated the P matrix, then each output value can be obtained by selecting the correspondent output row from the matrix and choosing the column, n with highest probability value. The number of this column can then be used to retrieve the final value from the outcome vector, thus k_n will hold the output value.

$$y_n = k_{n,i} \qquad where \quad \forall_{x \in X} \, P_{n,i} > P_{n,x} \tag{3.3}$$

```
/**
 * Returns the index of the max value in an array.
 * @param v array of numbers
 * @param n number of elements in the array
 * @return index of the max value in the array
 */
size_t maxindex(const double v[], size_t n) {
    size_t index = 0;
    for (size_t i = 0; i < n; i++) {
        if (v[i] > v[index]) {
            index = i;
        }
    }
    return index;
}

/**
 * A linear predictor function that constructs the score from a set of
 * coefficients and inputs.
 * @param b array of coefficients
 * @param x array of size n that contains the inputs to read
 * @return the calculated probability
 */
double score(const double b[], const double x[]) {
    double y = b[0] + b[1] * x[0] + ... + b[n] * x[n - 1];
    return 1 / (1 + exp(-y));
}

#define n NINPUTS
#define m NOUTPUTS
#define o NOUTCOMES
```

```
void logistic_regression(const double x[], const double y[]) {
    const double k[o] = { k_0, ... k_o };
    const double b[m][o][n] = {
        { { b_0_0_0, ..., b_0_0_n }, ..., { b_0_o_0, ..., b_0_o_n } },
        ...
        { { b_m_0_0, ..., b_m_0_n }, ..., { b_m_o_0, ..., b_m_o_n } }
    };
    const double p[m][o] = {
        { score(b[0][0], x), ..., score(b[0][o], x) },
        ...
        { score(b[m][0], x), ..., score(b[m][o], x) }
    };

    y[0] = k[maxindex(p[0])];
    ...
    y[m] = k[maxindex(p[m])];
}
```

Listing 3.2: Relationship code for a logistic regression strategy

Using a template engine to generate C source code

A template engine is a mechanism for text generation based on template definitions that have instructions for embedding data available during its processing (Kelly and Tolvanen, 2008, Chapter 11, p. 272). Widely used when developing web servers that dynamically generate and serve HTML pages, template engines usually function as an extension of the language to be generated, allowing usage of extra constructs like variable interpolation, conditionals and loops. The result of template processing is a string that contains all the specified substitutions done by the engine (Parr, 2004).

Having the previous proposed C code for the set of relationships R in mind, it is possible to sketch a template that generates the required C functions:

```
/**
 * Relationship function that returns the approximation result for an output.
 * @param o index of output to calculate
 * @param x array of size n that contains inputs to read
 * @param n number of inputs
 * @param m number of outputs
 * @return the calculated value for the output i
 */
fmi2Real R(fmi2ValueReference o, fmi2Real x[], size_t n, size_t m) {
    // Empty array to store the outputs
    fmi2Real y[m] = {};

    /*% if strategy == "linear" %*/
        linear_regression(x, y);
    /*% elif strategy == "logistic" %*/
```

```
        logistic_regression(b, k, x, y);
    /*% endif %*/

    return y[o];
}
```

Listing 3.3: Template to generate the relationships C code

The commands between "/*%" and "%*/", besides being valid C comments, are also template tags that dictate how the substitutions will be done. In this particular case, these commands describe a conditional statement that based on the context passed to the template (the variable `strategy`) decide which part of the code should be included in the final interpolation of the source code.

Implementing the Functional Mockup Interface

Having the means to correctly translate the relationship R to valid C code, it is necessary to make sure that the generated code implements the FMI. Since the only task of the FMU approximation tool is to dictate the mapping between the different inputs and outputs, only functions that deal with setting and getting values need to be addressed. All the other functions specified in the standard can simply return an OK status code.

The first step towards FMI compliance is to define a global buffer to store the values of the different variables (inputs and outputs). For the context of this work only variables of type *real* shall be considered, due to the fact that the domain and codomain of the relationship R is the set of real numbers, $\mathbb{R}$. Each variable is uniquely identified by a reference number, which in this particular scenario corresponds to its index in the buffer.

```
#define NINPUTS /*{{ len(inputs) }}*/
#define NOUTPUTS /*{{ len(outputs) }}*/

fmi2Real VARIABLES[NINPUTS + NOUTPUTS];
```

Listing 3.4: Declaration of the "VARIABLES" buffer

The function `fmi2GetReal` dictates what values should the variables hold on an given instance. In this scenario, it can be considered the "core" of the program, where the outputs are actually calculated based on the input values of the buffer. The function R described before is used to select the pointer to the function that corresponds to the mapping between the outputs and inputs. The array `vr` of size `nvr` contains the reference indexes of the variables to update and the array `value` is the buffer to store the calculated results (*Functional Mockup Interface for Model Exchange and Co-Simulation* 2020, Chapter 2, pp. 24–25). An OK status code is returned upon a successful calculation of the variables.

```
fmi2Status fmi2GetReal(fmi2Component c,
                       const fmi2ValueReference vr[],
                       size_t nvr,
```

```
                        fmi2Real value[]) {
    size_t i = 0;
    for (size_t i = 0; i < nvr; i++) {
        fmi2ValueReference vref = vr[i];
        value[i] = R(vref - 1, VARIABLES, NINPUTS, NOUTPUTS);
    }
    return fmi2OK;
}
```

Listing 3.5: Implementation of the “fmi2GetReal” function

The function `fmi2SetReal` is responsible for assigning new values to the variables. Once again, the array `vr` of size `nvr` contains the reference indexes of the variables to update, but this time the array `value` contains the actual values of these variables (*Functional Mockup Interface for Model Exchange and Co-Simulation* 2020, Chapter 2, pp. 24–25). `fmi2SetReal` will always return an OK status code because its only task is to update the `VARIABLES` buffer with the new given values.

```
fmi2Status fmi2SetReal(fmi2Component c,
                       const fmi2ValueReference vr[],
                       size_t nvr,
                       const fmi2Real value[]) {
    for (size_t i = 0; i < nvr; i++) {
        fmi2ValueReference vref = vr[i];
        VARIABLES[vref - 1] = value[i];
    }
    return fmi2OK;
}
```

Listing 3.6: Implementation of the “fmi2SetReal“ function

3.3.3 *Compiling the FMU*

After the C source files are generated they need to be compiled into shared library objects suitable for the target platform. The compilation result is then put inside the *binaries* directory under the target system architecture folder:

```
binaries/
    darwin32/
        model.dylib
    darwin64/
        model.dylib
    linux32/
        model.so
    linux64/
        model.so
    win32/
        model.dll
```

```
    win64/
        model.dll
```

Different platforms and compilers have different ways to compile shared library objects. For example, on a Linux machine, GNU Compiler Collection (GCC) provides the flags `-shared` and `-fPIC` (position independent code) to compile a given C file into a shared library object (Stallman et al., 2003, Chapter 3, p. 213):

```
$ gcc -shared -fPIC model.c -o model.so
```

On a Windows machine with Microsoft Visual C++ tool chain installed the following command compiles a C source file into a DLL (*Visual C++ Documentation* 2019):

```
cl /LD model.c /model.dll
```

Cross-compilation

As explained above, the compiled shared libraries can only be used by machines with the same architecture and operating system, therefore if it is intended to distribute the FMU in different platforms, each system will need to re-compile the source code accordingly. Cross-compilation tools make it possible to generate binary code for different platforms other than the host (Stallman et al., 1999, Chapter 4, pp. 139–140), this way it is possible for the machine that generates the FMU to easily distribute it with other systems. Furthermore INTO-CPS only allows loading co-simulation models that are already compiled, that is why it is important for the approximation tool to be able to generate a ready to use FMU packed with the binaries for the different platforms.

There are various approaches for FMU cross-compilation, each one with its advantages and disadvantages. The next paragraphs make a slight comparison between these methods:

MAKEFILE The most straightforward way to achieve FMU cross-compilation is to also generate a Makefile alongside the sources, and execute it before joining the files in the FMU archive. This Makefile would call all the required compilers with the flags needed to build the different shared libraries. For example it would execute GCC to create Linux shared object libraries and Minimalist GNU for Windows (MinGW) to create Windows DLL. This approach requires the user to have correctly installed the different compilers and will work only on UNIX systems, since Make is not available on Windows platforms (unless a GNU compatibility layer like Cygwin is also installed) (Mecklenburg, 2004).

CMAKE CMake is a cross-platform build system generator that uses a configuration file (named `CMakeLists.txt`) to generate native build scripts for different architectures. It has builtin support for cross-compilation provided that the configuration is split into “toolchain files” that specify each target platform vicissitudes (K. Martin and Hoffman, 2008, Chapter 8, pp. 126–128). Like the Makefile approach discussed in the previous paragraph, using CMake for cross-compilation also requires the user to have correctly installed all the needed compilers, having the advantage of being a cross-platform tool that can be run either on Windows, Linux or on Mac.

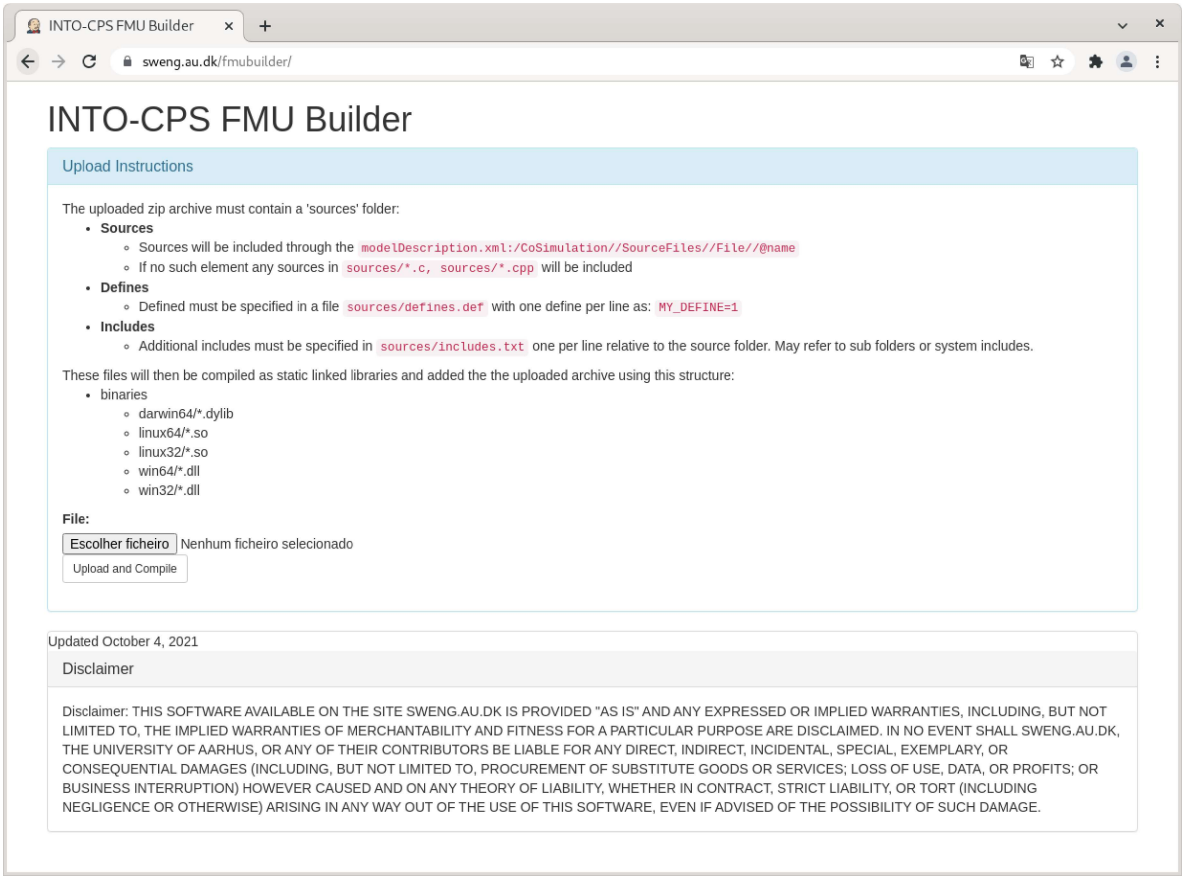


Figure 3: Screenshot of the FMU builder website

DOCKER Docker is an open-source technology that helps developing and deploying software in containers, using operating-system-level virtualization. A container is an instance of an image which in turn contains the commands to run and build itself. Simplistically Docker containers can be seen as lightweight virtual machines (Turnbull, 2014). For the purpose of this work it would be interesting to build a Docker image that contains all the required tools for the cross-compilation process of the source code. This way when using the approximation tool the user would not have to install all of the required compilers, being only necessary to have a running Docker instance.

FMU-BUILDER Since the whole process of compiling an FMU is tedious and error prone, it is also a valid option to delegate this work to a specialized service. Such is the case of INTO-CPS FMU–Builder (Lausdahl et al., 2016), an online website that allows the user to upload a *zipfile* containing the source code of the FMU. The website cross–compiles the code for 3 different platforms (Darwin, Linux and Windows) and 32 and 64 bit architectures. A download link is then presented to the user, containing the ready to use FMU with the various compiled shared libraries.

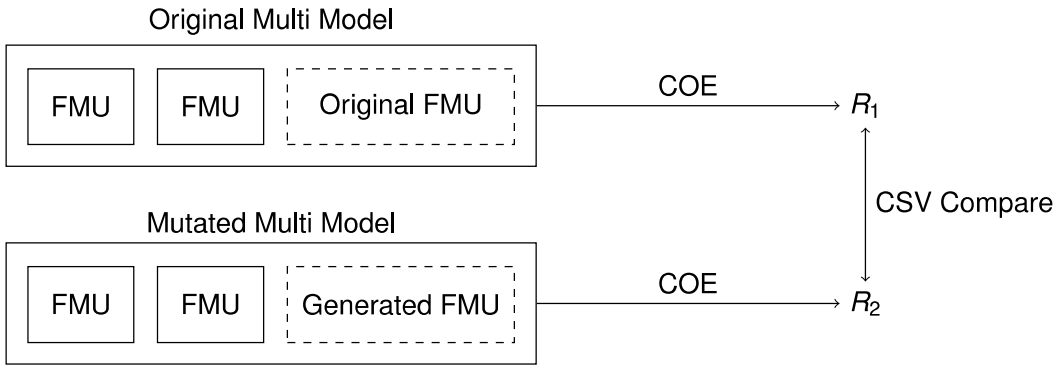


Figure 4: FMU comparison process pipeline

3.3.4 *Testing and evaluating the generated FMU*

Generating an FMU is only halfway of the process of building a reliable FMU approximation tool. It is required for the generated result to actually behave similarly to the unit to approximate.

The first step on evaluating the generated FMU is to ensure that it actually implements the FMI correctly. To automatize this process Modelica developed a CLI program, named FMU Compliance Checker (Nakhimovski, Fredriksson, et al., 2012), that verifies if a given FMU is valid or not (Bertsch, Ahle, and Schulmeister, 2014). Second, it is also important to test the generated FMU in a co-simulation environment, since it is expected for the FMU to interact correctly with other FMU. Assuming that there is a multi model containing the original FMU it is possible to compare the results of the performance of both original and generated FMU by creating a new "mutated" multi model copy where the original FMU is replaced by the generated one. Both multi models shall be executed by the same COE and then the produced results, R_1 and R_2, can be compared. Since these results are stored in CSV format it is possible to check their similarity by comparing the corresponding plots, which precisely what CSV Compare (Rütz, Sjölund, Beutlich, et al., 2013), another open-source tool by Modelica, does.

3.4 AUTOFMU

As a proof of concept of this chapter, a CLI Python program named AutoFMU was developed to meet the requirements presented in the previous sections.

3.4.1 *Overview*

Installation

Similarly to many other Python programs, AutoFMU uses *distutils* for building and installing a distributable Python package (Hetland, 2017, Chapter 18, pp. 402). Its releases are published to

Python Package Index (PyPI), therefore it is easy to install AutoFMU with Package Installer for Python (pip) by running the following command:

```
$ pip install autofmu --user
```

It is also possible to download and manually install the latest development version of AutoFMU, by cloning the Git repository (Chacon and Straub, 2014, Chapter 1, pp. 11) locally and running pip inside the cloned directory:

```
$ git clone https://github.com/ajcerejeira/autofmu.git
$ cd autofmu/
$ pip install . --user
```

Listing 3.7: Installation with Git

Usage

After it is installed AutoFMU can be run as a command in the shell.

```
$ autofmu dataset.csv --inputs x y --outputs z --strategy=linear -o model.fmu
```

Above is an example on how to run AutoFMU and some of its parameters where:

- *dataset.csv* is the name of the CSV file that contains the data that will be used to train the model for deducing the approximation. Table 1 contains a simple example of a minimal CSV file where the header contains the name of the variables, and each row a set of values for each variable.

x	y	z
0.24	0.72	0.00
0.53	0.61	0.91
0.12	0.47	0.71
	...	

Table 1: Example CSV table containing an AutoFMU dataset.

- *x* and *y* are the names of the columns of the dataset that contain the input values;
- *z* is the name of the column of the dataset that contain the output values;
- *linear* is the name of the strategy to use for approximating the FMU;
- *model.fmu* is the filename for the generated FMU.

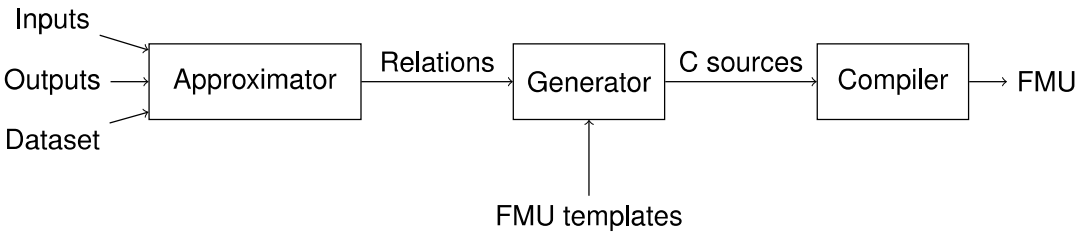


Figure 5: Overview of AutoFMU pipeline from data analysis to FMU compilation

3.4.2 *Architecture*

One of the core tenets of the UNIX philosophy is to "do one thing and do it well" (Raymond, 2003, Chapter 1, pp. 36), and based on that principle AutoFMU is composed by three main modules each one with a single scope and purpose:

- *Approximator* is the primary module of this project and the "brain" behind AutoFMU. It reads the dataset and finds the relationships between the specified inputs and outputs. Before performing the approximation it sanitizes the dataset, removing empty entries and checking for missing values. After the data is ready for analysis it uses the specified approximation ML strategy to deduce a possible relationship between the inputs and outputs. Internally it uses scikit-learn, a simple but powerful ML library (Pedregosa et al., 2011), for applying the different deduction strategies. The structure of the resulting artifact for this module will depend on the used strategy, and it will contain all the information required for building the mapping function. For example, for a linear regression strategy the result will hold all the coefficients and intercepts needed for calculating an output value.

- *Generator* is the module responsible for generating the source code for the FMU. It reads the calculated relationships between the inputs and outputs variables and inserts these values into a FMU template. Internally the C files that provide the skeleton for the FMU are written in Jinja2 template language which is powerful enough to represent the constructs needed for a valid C program (Ronacher, 2008). The artifact of running this module is a valid FMU that contains all the sources and model description files without any compiled binary. Therefore, at this stage the FMU is not runnable as it is.

- *Compiler* cross compiles the sources of an FMU to multiple platforms. Invokes CMake for the compilation process since it makes it simple to generate shared library binaries for multiple platforms. It can also be installed with pip which means that it can be specified as a Python dependency, so when installing AutoFMU the user will automatically get the binaries for running CMake. The final result of this process is a valid FMU ready to be used either standalone or in a multi-model project.

3.4.3 *Development*

For tackling the complexity that inevitably arises when developing such a broad and complex program it is imperative to employ the best software engineering practices. From the beginning of this project a great focus has been given to topics such as such as testing, documentation and code quality, always aligning the development of new features with this mindset, to make sure that the final result is a robust program of great quality.

Testing

It is well known that Dijkstra claimed that "program testing can be used to show the presence of bugs, but never to show their absence" (Dijkstra, 1970, Section 3, pp. 6). While acknowledging this corollary, automatic software testing can be really useful specially when working with dynamic languages such as Python. Having a full unit test suite allows for better reasoning when implementing new features and refactoring, making sure no regression bugs are introduced in the development process. In fact R. C. Martin goes as far as claiming that "having an automated suite of unit tests that cover the production code is the key to keeping your design architecture as clean as possible" (R. C. Martin, 2009, Chapter 9, pp. 124).

For the AutoFMU program the tests are written with *unittest* module which is the official testing framework provided by Python standard library (Percival, 2014, Chapter 2, pp. 16), therefore it is possible to run the full test suite with the following command:

```
$ python -m unittest
```

The program contains a comprehensive test suite being that for each module there exists an associated unit test. This allows for a total of 97% code coverage which means that every time the test suite is run nearly all of the program code is executed. Although it is a fact that this metric can be misused (Marick et al., 1999), having a high code coverage percentage helps ensuring that no errors are raised when the unit tests execute the program code.

Code quality and static code analysis

Being a dynamically typed language with no compilation step, Python development usually relies on code quality and static code analysis tools to minimize the number of bugs and ensuring the code keeps a consistent style. In fact, some static code analysis tools can even help detecting security vulnerabilities without actually running the program (Goseva-Popstojanova and Perhinschi, 2015). For these reasons a set of high-quality code analysis tools were used when developing AutoFMU. Flake8 is one of them, being described as a linter that checks for code smells, complexity and pep8 code style (official Python style guide (Van Rossum, Warsaw, and Coghlan, 2001)) (Cordasco, 2016). It can be used to check the whole codebase with this command:

```
$ python -m flake8
```

Despite being a dynamically typed language, since the release of version 3.5, Python has support for type annotations and even includes a typing module (Van Rossum, Lehtosalo, and Langa, 2014). Python code annotated with "type hints" can then be type checked using an external tool like Mypy that will analyze and perform static type checking on the program (Lehtosalo et al., 2014). This helps preventing a plethora of type related bugs without introducing a big overhead on project development, as the static typing analysis is optional and allows for a progressive enhancement of the codebase, thus joining the benefits from both dynamic and static typing worlds (Meijer and Drayton, 2004). AutoFMU includes a Mypy configuration file which allows performing the whole type checking with the following command:

```
$ python -m mypy src/ tests/
```

Documentation

Recognizing the importance of good documentation for the success of software engineering projects (Lethbridge, Singer, and Forward, 2003), AutoFMU is bundled with documents written in reStructuredText (RST) that are used as the sources for generating the documentation in different formats. The user manual can then be generated with Sphinx which is the *de facto* documentation generator for Python projects (Brandl, 2010).

The web version of the documentation is freely hosted by readthedocs.org service (Cerejeira, 2020b) and is built automatically by the Continuous Integration (CI) pipeline every time there is a commit pushed to the *master* branch.

Assuming the current directory as the AutoFMU source directory it is also possible to build the documentation locally with the following command:

```
$ cd docs/ && make html
```

The resulting HTML pages will be placed in the `_build/` directory and can be browsed by opening the `_build/html/index.html` file.

Continuous integration and delivery

Being advocated as modern development best practices, CI and Continuous Delivery (CD) pipelines provide workflow automation that bring a more cohesive experience when building, testing and deploying software projects (Fowler and Foemmel, 2006).

For this project, GitHub Actions were used to build and host the CI and CD pipelines. Since AutoFMU source code is already hosted on GitHub, GitHub Actions provided a great integration between the repository and version control events (commits, releases, branches, *etc.*). The pipeline itself is defined in a set of YAML workflows files that describe the steps needed for running the CI commands (Kinsman et al., 2021). On every pushed commit the code is automatically linted, type checked and tested, using the tools described before. The code coverage result is then uploaded to Codecov (Cerejeira, 2020a) an external service dedicated to measuring code coverage across repository modules.

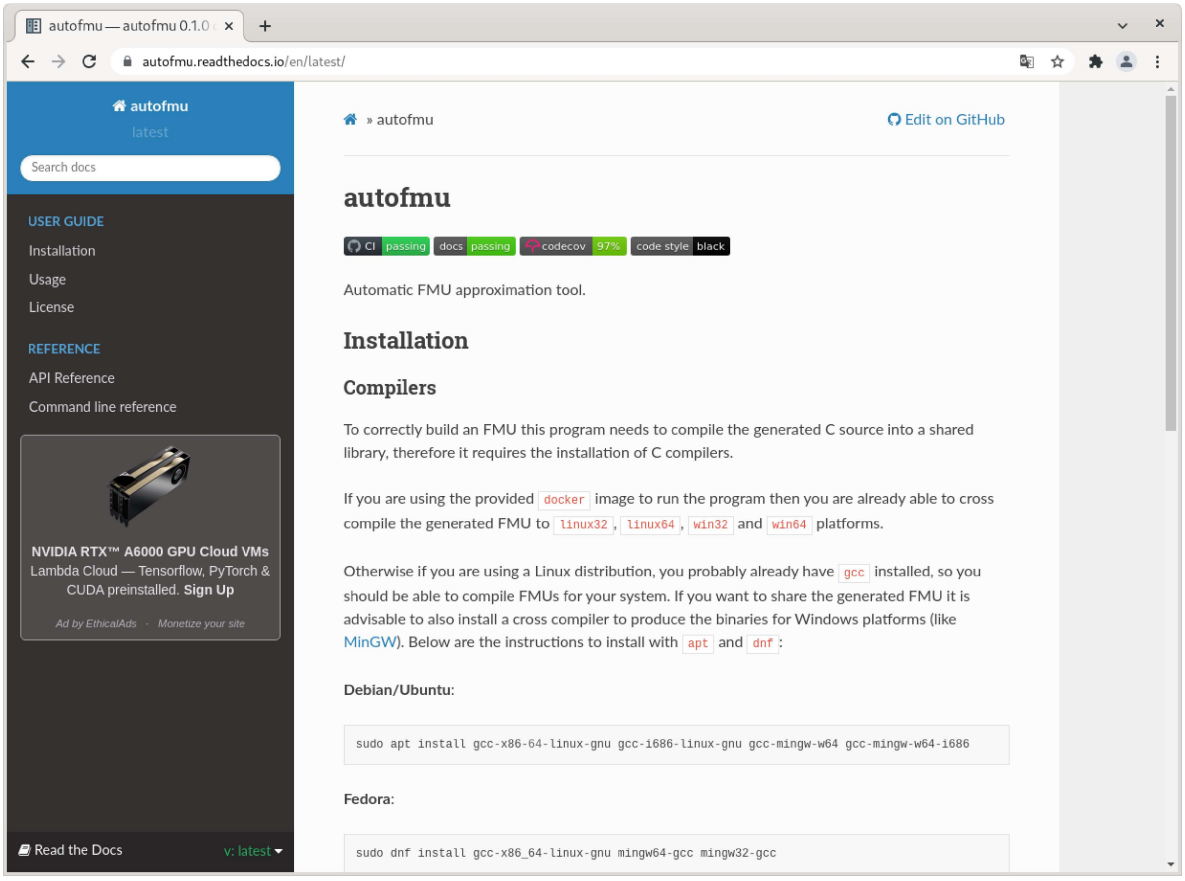


Figure 6: Screenshot of the documentation website

If any error occurs the pipeline stops and the error is automatically reported. This allows for a more robust and iterative development process, enabling the developer to quickly inspect faulty commits that might have introduced bugs in the codebase.

The delivery process is also handled by GitHub Actions. Whenever there is a push to the `master` branch and the CI pipeline succeeds, the release process is activated, generating and uploading the program documentation and creating a new release in Python package index.

The automation of the whole process of code integration and delivery brought many benefits to the project development, allowing the developer to focus more on feature development while ensuring the robustness and high quality of the code.

3.5 SUMMARY

This chapter has shown how a tool for inferring FMUs can be built using adequate techniques and libraries. A detailed understanding of the FMI is required in order to produce valid FMUs that can be immediately used by other programs. To infer the relationships between inputs and outputs in the given dataset, the program leverages existing ML frameworks, the main challenge consisting of converting the such relationships into C code. By organizing the program into different modules,

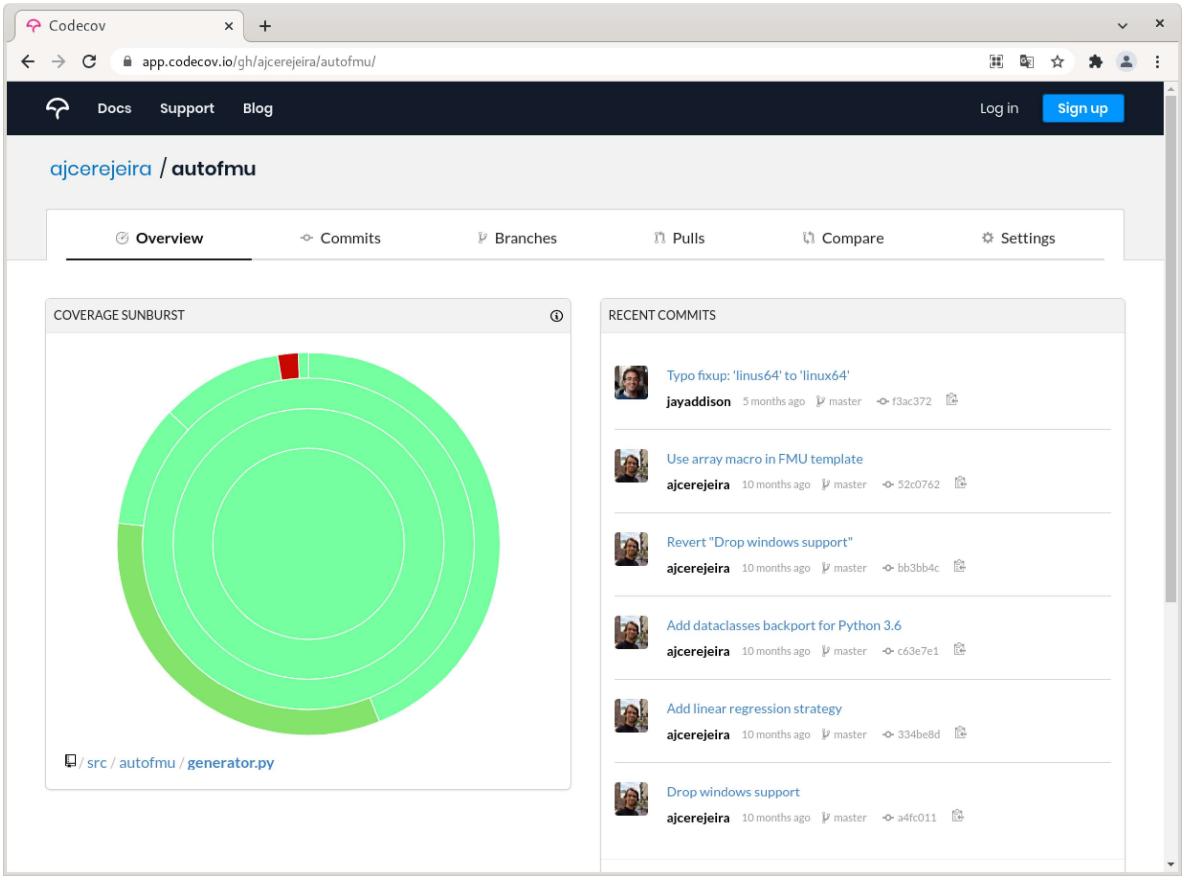


Figure 7: Screenshot of the code coverage analysis website

each one with a specific purpose in mind, it is possible to create a solid program that merges these different fields of knowledge under the same purpose.

4

CASE STUDY: A LINE FOLLOWER ROBOT

In order to evaluate the viability and quality of the approach discussed in this document, the software developed in Chapter 3 was used and tested in a real world–environment.

This chapter shows the process of deducing an approximation of a model component of the line follower robot, using the techniques described in the previous chapters. The results gathered are hereby discussed and further evaluated.

4.1 THE LINE FOLLOWER ROBOT

A line follower robot is an autonomous device that can read a line drawn on the ground and move according to its trajectory. The line can be a visible black path, or an invisible magnetic field. To detect and analyze the line, the robot is equipped with infrared sensors under its body. These sensors are connected to a microcontroller that processes the incoming data and decides what kind of movement the robot should follow. To accomplish the desired movement, there are two wheels with independent motors, meaning that each wheel can move with a different velocity, and this difference is what sets up the direction of the robot (Pakdaman and Sanaatiyan, 2009).

This kind of device has a lot of useful applications. One of them is automated SmartCarts used by Tesla Motors, that follow magnetic strips on the floor to transport Tesla Model S to the assembly center (Blankenshi, 2012). In a health care management system line follower robots can be used to monitor and transport medicine to the patients (Punetha, N. Kumar, and Mehta, 2013). Its usefulness extends also to the entertainment field, already existing a robot designed to entertain children in shopping centers, capable of transporting up to five passengers (Colak and Yildirim, 2009).

For the purpose of this work, a R2-G2P line follower robot unit was deeply studied. The content of this chapter is based on the observations to this particular unit.

The robot is composed by the following physical components that interact with each other (Payne et al., 2016):

- *Controller* an Arduino powers the controller of the robot. It reads the values from the sensors and calculates the amount of energy to apply to each wheel, sending this output to the body.
- *Body* composed by two wheels on the back, each powered by individual motors, and a shopping cart like wheel on the front. It holds 3 AA batteries that power the whole system.

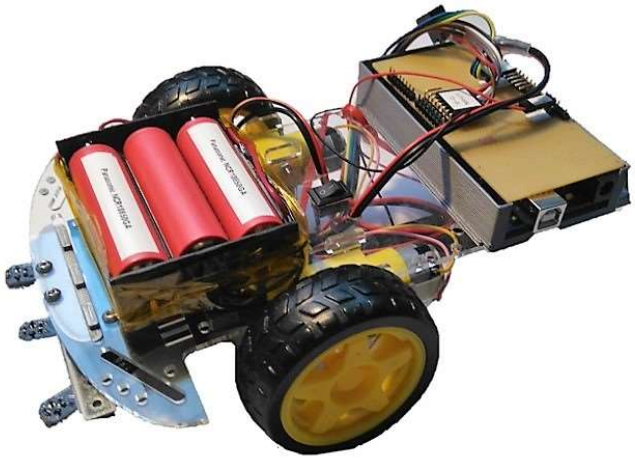


Figure 8: The line follower robot used in this work

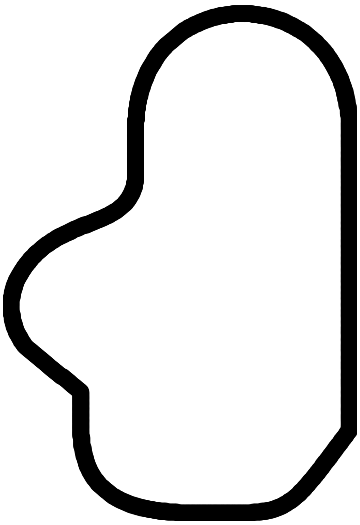


Figure 9: Example map that contains the path for the robot to follow

- *Sensors* three light sensors, placed in the front of the robot that detect light and dark areas on the ground.
- *On/Off button* a single button that turns the robot on and off.

To get the robot in motion, it should be placed at the beginning of the black line of the trajectory and then activating the on/off switch. The robot will then try to follow that line until its sensors can no longer detect it, making the robot stop the movement. For testing purposes, this work uses a sample map that contains a path for the robot to follow (Fig. 9). This path has already been built to contain segments representing different difficulties for the robot to follow, from straight lines to tight curves.

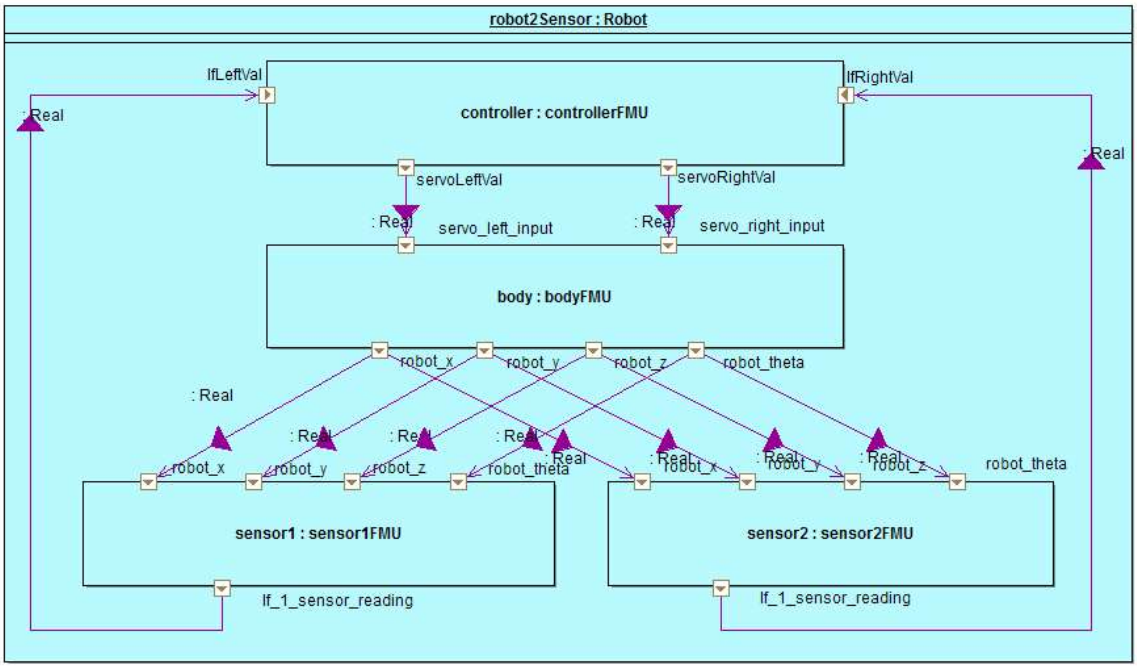


Figure 10: SysML diagram that shows the interactions between the robot FMUs

4.2 MODELLING THE LINE FOLLOWER ROBOT

The act of choosing the line follower robot as a case study for this work was also influenced by the fact that there is already a INTO-CPS project (The INTO–CPS Association, 2016) that contains a set of multi-models for simulating the line follower robot behavior and interactions. In this project it is clearly visible that the FMI enables a great interoperability between FMUs built with different tools, not tying the model developers to a specific software. From SysML diagrams that provide a high level overview on how the different robot components interact between each other to the specific controller algorithms written in VDM, this project proves that multidisciplinary model designing is not only possible but also produces quality results.

A quick understanding of the system as a whole can be achieved by looking at the SysML diagrams that accompany this project (Fig. 10) where the communication points between the different robot FMUs are visible. From that figure it follows that for each robot component there is an associated FMU with a clearly defined interface that specifies its inputs and outputs.

CONTROLLER The controller is the logic unit that based on the ambient light values read from the sensors calculates the amount of energy to put on each rear wheel. The algorithm that defines the rules for this calculation is based on a piecewise-defined function where each branch corresponds to a type of movement that the robot can follow: moving forward, rotate to the left and right and stop. Based on the code for the controller FMU this function can be summed up by Eq. (4.1), where:

- $servo_L$ and $servo_R$ are the amount of energy to put on each wheel.

	FMU			
	controller	body	sensor1	sensor2
Variable name(s)				
`lf_1_sensor_reading/lfLeftVal`	`INPUT`		`OUTPUT`	
`lf_2_sensor_reading/lfRightVal`	`INPUT`			`OUTPUT`
`servoLeftVal/servo_left_input`	`OUTPUT`	`INPUT`		
`servoRightVal/servo_right_input`	`OUTPUT`	`INPUT`		
`robot_x`		`OUTPUT`	`INPUT`	`INPUT`
`robot_y`		`OUTPUT`	`INPUT`	`INPUT`
`robot_z`		`OUTPUT`	`INPUT`	`INPUT`
`robot_theta`		`OUTPUT`	`INPUT`	`INPUT`

Table 2: Variable classification by FMU

- $sensor_L$ and $sensor_R$ are the ambient light values read from the sensors.
- fs, fr and br are constants that can be overwritten by the multi-model and correspond to the forward speed, forward rotation and backward rotation values respectively.

$$(servo_L, servo_R) = \begin{cases} (fs, -fs) & sensor_L < 150,\ sensor_R < 150 \\ (fr, -br) & sensor_L < 150,\ sensor_R > 150 \\ (br, -fr) & sensor_L > 150,\ sensor_R < 150 \\ (0, 0) & sensor_L > 150,\ sensor_R > 150 \end{cases} \tag{4.1}$$

BODY The body is the FMU responsible for reading the energy values calculated by the controller and determine the position coordinates for the robot based on these values, thus being the model that makes the robot actually move.

SENSOR The sensor reads the current position of the robot from the body FMU and returns a number between 0 and 255 that corresponds to the amount of ambient light that the sensor of the robot can read on that position. Since the model hereby studied uses two sensors, this INTO-CPS project has two instances of this same FMU positioned on the left and right sides of the front part of the robot.

The default multi model configuration defines the coordinates of the initial position of the robot and the values of energy to apply to each one of the servo wheels.

4.3 SIMULATING THE LINE FOLLOWER ROBOT MOVEMENT

INTO-CPS has an easy to use and intuitive GUI that allows the model developers to do all of the simulation work directly in the project window. Additionally INTO-CPS also allows running the COE directly in the shell as long as the system has Java runtime environment installed. In this work

the simulations will be demonstrated using the CLI interface for a number of different reasons, one of which is the fact that it allows a better understanding of the COE Application Programming Interface (API) and the fact that using the CLI is more platform agnostic and flexible, being easily automated with scripting, which is not possible when using the GUI (Stephenson, 1999).

Before starting the simulation the INTO-CPS COE server must be running. To launch it open a shell in the INTO-CPS install downloads directory and run:

```
$ java -jar coe.jar
Version: 1.0.0
Now running on port 8082
```

Listing 4.1: Launching the COE server

The COE server should now be running locally on port 8082 and is accessible at `http://localhost:8082/`. Therefore it is now possible to perform requests to the COE and fully access its functionalities. In this work the CLI tool, cURL is used for easily making requests to the server directly on the terminal shell. The first step to prepare the multi-model for the simulation is to create a INTO-CPS session, which can be achieved with the following command:

```
$ curl localhost:8082/createSession
{"sessionId": "85109b8c-8e05-47e6-8b2b-91cf72276127"}
```

Listing 4.2: Creating a session in the COE server

The previous command will return an identifier for the session which will be required for subsequent requests. In this context it is time to prepare the COE for the simulation, therefore a `POST` request with a payload that includes the contents of the multi-model JavaScript Object Notation (JSON) configuration file will be sent to the server:

```
$ curl localhost:8082/simulate/85109b8c-8e05-47e6-8b2b-91cf72276127 \
       -X POST -H "Content-Type:␣application/json" \
       -d '{"startTime":␣0,␣"endTime":␣40}'
{
    "status": "Finished",
    "sessionId": "85109b8c-8e05-47e6-8b2b-91cf72276127",
    "lastExecTime": 2885
}
```

Listing 4.3: Simulating the multi-model with the COE

To conclude the simulation it is also advisable to close the session in the COE server:

```
$ curl localhost:8082/destroy/85109b8c-8e05-47e6-8b2b-91cf72276127
Session 80931ae4-d626-43b2-b38a-eb4bb5b8d1d0 destroyed
```

Listing 4.4: Destroying a session in the COE server

After the COE concludes the simulation a CSV file containing the results data table will be created. This table includes a set of time based entries for each FMU variable, allowing the reader

time	θ	*x*	*y*	*z*	$servo_L$	$servo_R$	$sensor_L$	$sensor_R$
0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00
0.01	0.00	0.14	−0.08	0.00	0.40	−0.40	143.00	144.00
0.02	0.00	0.14	−0.08	0.00	0.40	−0.40	206.00	206.00
				⋮				
39.99	9.65	−0.12	−0.10	0.00	0.50	−0.10	238.00	254.00

Table 3: Excerpt from the table of results from the simulation

to observe how each variable varies in time and depending on other variable values. Table 3 contains an excerpt of the of the results data from the simulation above.

With this data it is possible to visualize the robot movement by selecting the *x* and *y* columns and creating a plot as shown in Fig. 11.

4.4 USING AUTOFMU TO APPROXIMATE A COMPONENT OF THE LINE FOLLOWER ROBOT

The previous simulations have resulted into a set of useful data that can be used as a control group in this experiment, acting as the reference for the generated FMUs to be compared against.

In this work the component of the robot that will be approximated is the controller unit. This choice is supported by the fact this is the main logic unit that actually performs the calculations required to put the robot in movement. Furthermore its behavior is fully specified in Eq. (4.1) derived from the VDM model sources, which means that it is possible to compare the algorithm generated by the approximation with the original one. Another reason for choosing this unit for approximation lies on the fact that it is a self contained system with a clear definition of the inputs and outputs, therefore being possible to treat it as a black box, leaving the internal implementation for the approximation deduced by AutoFMU.

Assuming the Table 3 from the previous section was stored into a CSV file named `results.csv` it is possible to feed AutoFMU with these results, letting the program learn the relationships between the desired inputs and outputs. In this work two different FMUs will be generated, one approximated using a linear regression strategy and another with a multinomial logistic regression algorithm. The results for both of these approaches can be achieved with the following two commands:

```
$ autofmu results.csv \
    --inputs sensorL sensorR \
    --outputs servoL servoR \
    --strategy linear \
    -o linear-regression-model.fmu
```

Listing 4.5: Invoking AutoFMU with a linear regression strategy

```
$ autofmu results.csv \
    --inputs sensorL sensorR \
```

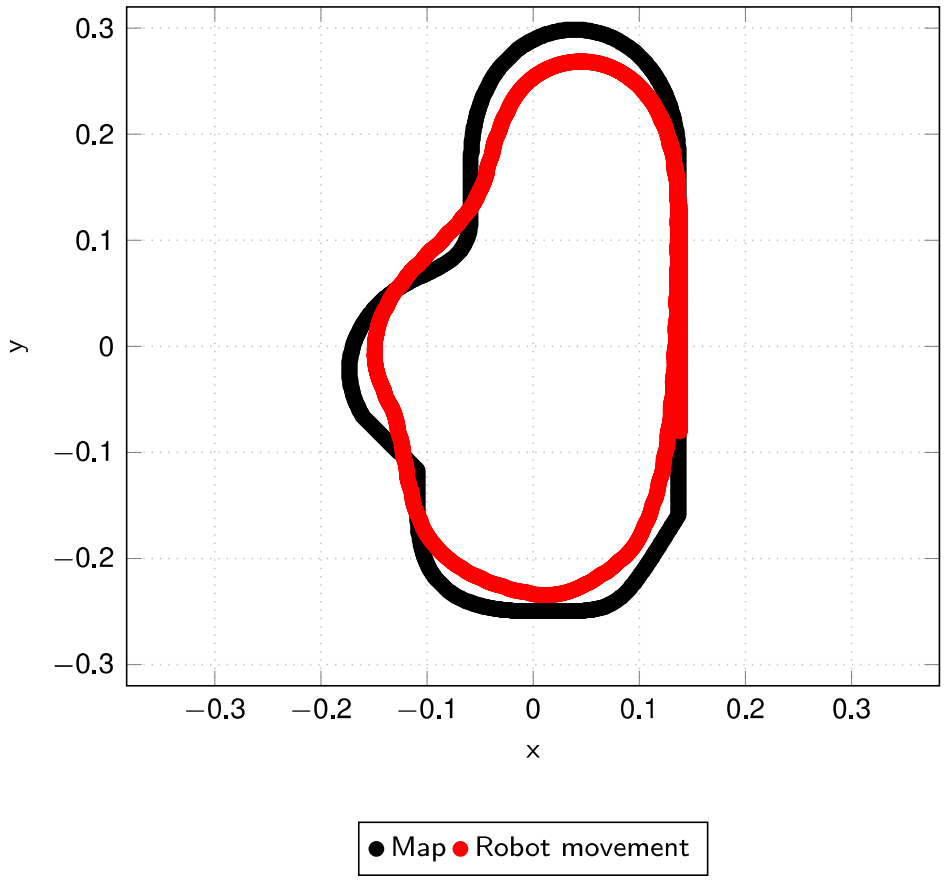


Figure 11: Path followed by the robot in the model simulation

```
    --outputs servoL servoR \
    --strategy logistic \
    -o logistic-regression-model.fmu
```

Listing 4.6: Invoking AutoFMU with a logistic regression strategy

4.5 TESTING THE GENERATED COMPONENTS

Having run the commands from the previous section there should now exist two different FMU files named `linear-regression-model.fmu` and `logistic-regression-model.fmu` respectively. To ensure that both of the generated FMUs are valid and that they properly implement the FMI, FMU Compliance Checker (as introduced in Chapter 3 Section 3.3.4) was used as demonstrated by the following snippet:

```
$ fmuCheck linear-regression-model.fmu
"time","servoLeftVal","servoRightVal"
FMU check summary:
FMU reported:
  0 warning(s) and error(s)
Checker reported:
```

```
  0 Warning(s)
  0 Error(s)
```

Listing 4.7: Running FMU Compliance Checker on the generated FMU

Having confirmed that the generated FMUs are valid it is now time to test them in a multi-model simulation scenario. For each approximated model it will be created a multi-model environment similar to the original one, but with the approximated controller FMU instead (as suggested by the test methodology proposed in Chapter 3 Section 3.3.4). This process can be automated with a bash script that copies the original multi-model configuration directory and replaces the controller FMU file with the generated one, while updating the multi-model JSON configuration file:

```
$ cp linear-regression-model.fmu logistic-regression-model.fmu \glspl{fmu}/

# Create a copy of the original multi model and update it
# to use the approximated \glspl{fmu}
$ cd Multi-models/
$ cp -r lfr-non3d/ lfr-non3d-linear/
$ sed -i \
    's/LFRController_Standalone/linear-regression-model/g' \
    lfr-non3d/lfr-non3d.mm.json
$ cp -r lfr-non3d/ lfr-non3d-logistic/
$ sed -i \
    's/LFRController_Standalone/logistic-regression-model/g' \
    lfr-non3d/lfr-non3d.mm.json
```

Listing 4.8: Bash script for creating multi-models for the approximated FMUs

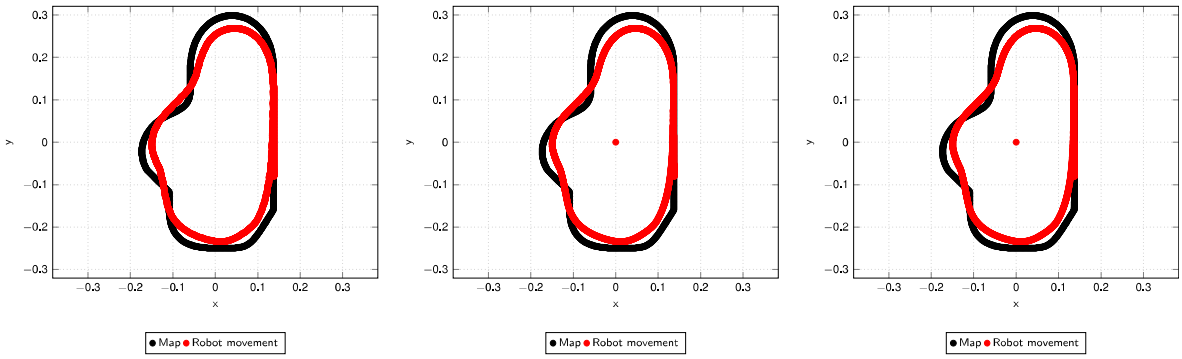
After the multi-models are created and placed in their respective folders (`lfr-non3d-linear` and `lfr-non3d-logistic` in this particular case) the simulations can be run by launching the COE and using the same commands demonstrated in Section 4.3). After the simulations complete the generated CSV files will be available to be analyzed and compared.

4.6 RESULTS ANALYSIS

To analyze the performance and correctness of the data generated by the approximation algorithms it is necessary to define a set of well defined criteria that allow a proper comparison of the results.

The first and simplest measurement is to compare the path followed by the robot between the different approximations and the original simulation. Since the movement happens on a two-dimensional plane, it is possible to visualize the followed path by drawing a scatter plot of the robot x and y coordinates. These values are accessible through the body FMU that provides `robot_x` and `robot_y` as output variables.

Figure 12 draws a comparison of the different results obtained by using each approximation strategy. At a glance it is possible to observe that the approximations performed quite well compared to the original simulation, in fact there seems to be almost no visible difference in the



(a) Original simulation (b) Linear regression (c) Logistic regression

Figure 12: Robot movement resulting from different approximation algorithms

	$robot_x$		$robot_y$		$servo_L$		$servo_R$	
	r^2	*MSE*	r^2	*MSE*	r^2	*MSE*	r^2	*MSE*
Linear	0.9858	0.0002	0.9831	0.9831	0.1838	0.1838	0.1317	0.0272
Logistic	0.8863	0.0013	0.8260	0.0041	-0.5369	0.0519	-0.5893	0.4000

Table 4: r^2 and mean squared error scores for each approximation strategy

path followed by the robot across these different results. The small dots at the center of the linear and logistic regressions approximations represent an outlier containing the initial robot position at coordinates $(0, 0)$ which can be ignored during this analysis.

Despite the apparent good performance of both approaches, it is important to apply objective measures to compare the approximation results in order to know exactly how precise each strategy was. The Mean Squared Error (MSE) is used in statistics to evaluate the quality of an estimator by measuring the average squared difference between the estimated values and the reference values. The smaller the MSE the more efficient is the estimator (Dekking et al., 2005, Chapter 20, pp. 305). For this case study the estimators to be compared and evaluated are the robot position (x and y coordinates) and servo left and right values. Table 4 draws a comparison between both approximation strategies and their r^2 and *MSE* values, calculated in regard to the original simulation result.

As expected by taking into account the graphics of the robot movement (Fig. 12) the *MSE* for the body position estimators (x and y) is almost zero, meaning that both approximations closely follow the original movement. It also shows that the linear regression strategy produces a slightly more accurate result compared to the logistic regression strategy. A curious fact that arises from analyzing the table is that in both cases the approximation of both servo left and right values were quite poor, in fact for the logistic regression strategy results the r^2 is negative. In statistical terms, this means that the approximated results fit worse than an horizontal line (Snijders and Bosker, 1994), so it is possible to conclude that the approximation was quite poor.

Another way of analyzing the results for the servo approximations is to draw a plot over time of the variation of the force applied to each wheel and compare it against the original approximation plot for the same variables. Figures 13 and 14 depict these same plots, where the main difference between

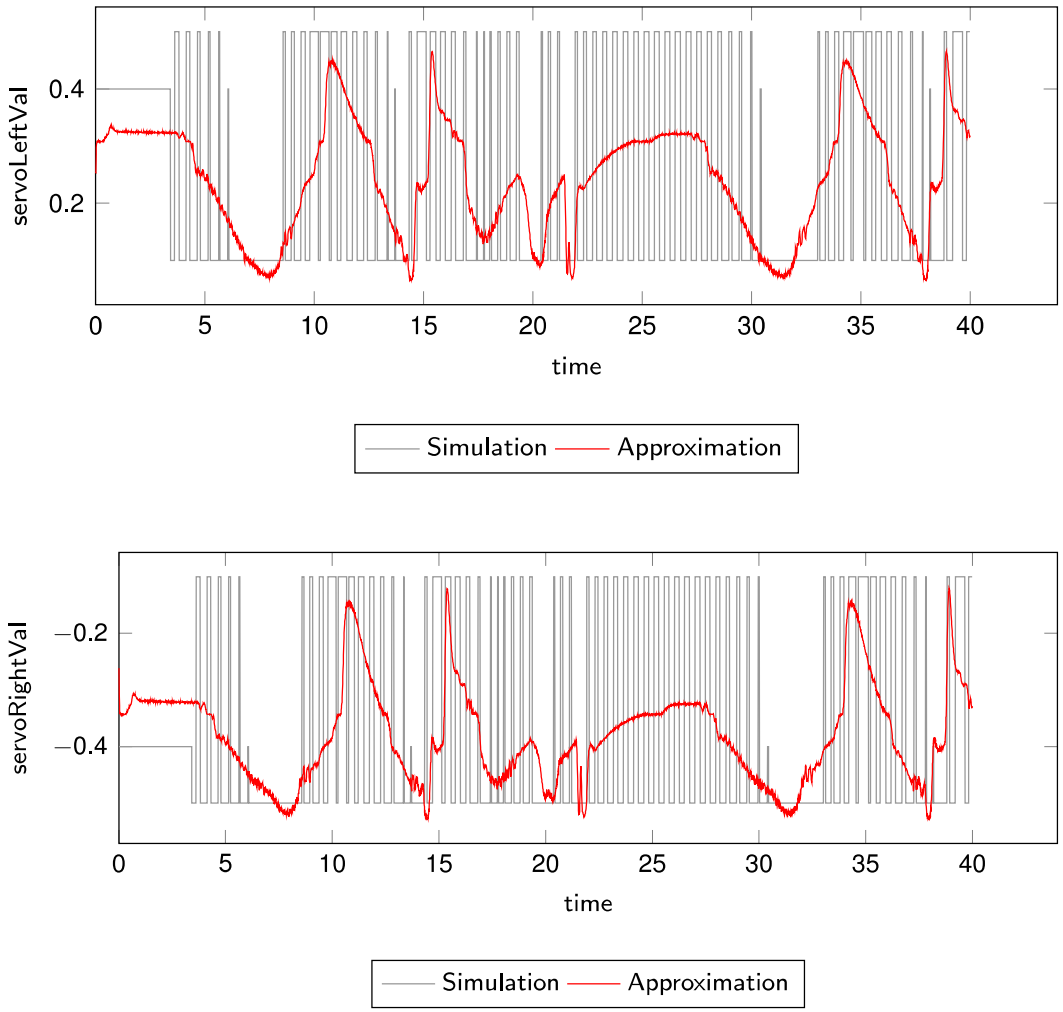


Figure 13: Servo values variation over time using the linear regression approximation

the different strategies can be observed. For the linear regression chart it is possible to observe that the plot of the approximated unit follows the tendencies of the original values, increasing and decreasing at the same time. Moreover it is important to notice that the original algorithm that dictates the servo values is a discrete function and the linear regression approximation results are continuous. On the other hand, the values obtained with the logistic regression strategy predicatively match the expected ones, *i.e,* they always assume one of the possible outputs of the original servo equation. Interestingly, when looking solely at the generated plot, this strategy seems to provide a better fit for the original values, however this is not true, based on the MSE calculated previously.

Bearing in mind that both results of the approximations of the servo values are not the best, it is legitimate to question why the robot followed so closely the original path – it would be expected for the robot movement to be completely incorrect based on the low score of the strategies hereby employed. A possible explanation would be that the time step used between each calculation is so small that even if the calculated value for the servo vastly differs from the original ones the robot still moves in the right direction.

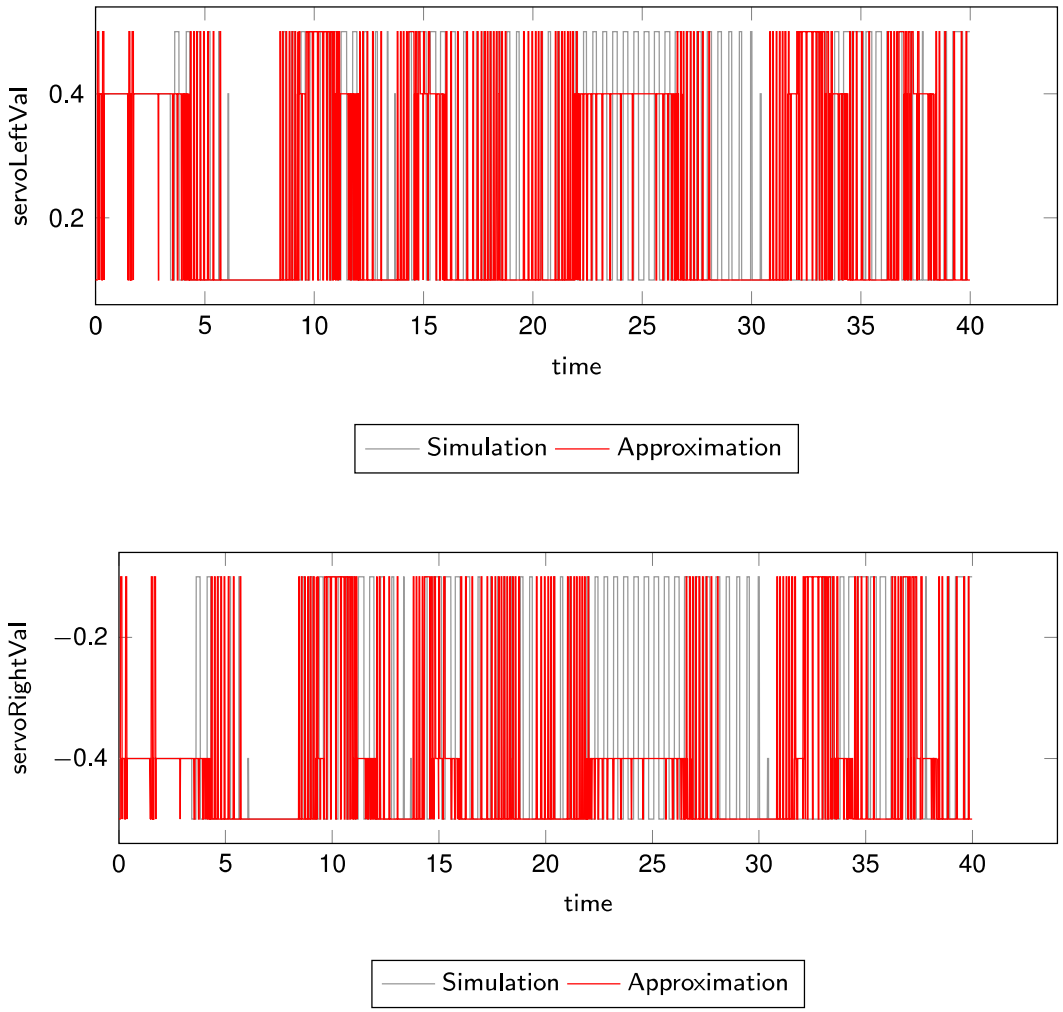


Figure 14: Servo values variation over time using the logistic regression approximation

Overall it is fair to conclude that, despite the approximations of both servo units were not very accurate, the behavior demonstrated by the robot when using the generated FMUs was satisfactory, since it closely followed the line drawn on the map. Therefore, for this specific physical unit, AutoFMU is a program able to deduce a reliable approximation. In Chapter 5 future work ideas will be discussed in order to improve the approximation algorithms and to test the program among a different number of devices.

4.7 SUMMARY

In this chapter the program developed alongside this work was put to the test in a real–world scenario. An approximated FMU for the controller part of a line follower robot is inferred, based on the data gathered from a real physical unit. The generated FMU was then placed in a multi–model representation of the robot where its behavior was tested and simulated. The path followed by the robot and the data issued by the FMU were analysed, as discussed in the conclusions chapter that follows.

5

CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSIONS

The development of CPSs requires a strong multidisciplinary knowledge of the various components of the models that interact with the physical world. The creation of a program to automate such processes is bound *a priori* to the author's own knowledge on CPSs development. One such program was developed throughout this work, which has been used in a real–world scenario: the automatic generation of a controller model for a line–follower robot from the analysis of data referring to a physical unit.

As described at the end of Chapter 4, overall the results are satisfactory. On the one hand, from an experimental point of view, the behavior exhibited by the approximate model is quite satisfactory. The robot controller correctly makes the necessary decisions to follow the line drawn on the floor, behaving faithfully to the physical unit. In this respect, the results of the two different implemented strategies, linear and logistic regressions, are similar. On the other hand, after a more objective analysis of the data generated by the controller when reacting to the different inputs, it can be observed that these differ significantly from the results of the original unit.

The main difficulties experienced during the development of this program were essentially due to the author's inexperience in the two major topics covered here. The intersection of both topics, modelling CPSs and ML, is wide enough to require more development work. Their enormous scope requires a greater ability to focus on the issues really relevant to this dissertation.

Regarding FMU modeling, this in itself can become quite complex, requiring much knowledge of C to be able to properly implement the FMI. Building and distributing a FMU is also not a trivial task, especially considering that the models require cross-compilation in order to be usable by multiple systems architectures. It is fair to claim that both of these challenges were properly addressed in this work by limiting the FMI implementation to the functions that actually deal with the manipulation of input and output variables and using modern compilation techniques that make it easier to produce binaries for multiple platforms.

The data analysis and ML component of this work also proved to be an interesting challenge. On itself, ML development requires a depth understanding of the underlying problem – in AutoFMU case this is more difficult because, being a generic approximation program, we do not know *a priori* on which context it will be used, which means there is no specific knowledge about the data it consumes.

When building AutoFMU an important aspect that was taken into consideration was the ease of use of the program and its distribution and availability. Being able to quickly install it with pip and simple to understand CLI arguments makes for a smooth and intuitive user experience. Another advantage of being a CLI program is that it allows for easy composability with other programs making it easy to automate the generation of FMUs.

Altogether, the work that lead to the conclusion of this project proved to be very useful for the author, allowing him to better understand the different themes hereby discussed. By working at the same time on the theoretical level and on the implementation of a concrete program, in this dissertation both types of knowledge were deemed the same importance. Properly building a CLI tool, distributing it, writing documentation and tests allowed the application of the recommended software engineering principles, ensuring that the final bundle is robust and ready to be used in a real–world environment.

5.2 PROSPECT FOR FUTURE WORK

There is much potential for improving and expanding the functionality of the program developed in the work reported in this dissertation.

The most immediate way to make better model approximations would be to include other ML algorithms beside linear and logistic regression. Strategies like Support Vector Machines (SVM), decision trees, k–nearest neighbor and neural networks could be added to the codebase to allow for a broader approximation-algorithm choice. In fact, at the time of writing this work there has been some recent progress in the usage of neural networks for constructing models when simulating dynamical systems (Legaard, Schranz, et al., 2021). These strategies are more complex than the ones implemented here, and for certain scenarios they might provide better and more accurate results. A step further in the program automation pipeline is to automatically choose the strategy with the better results and better score, relieving the user from having to choose one at all.

Regarding the compilation of FMUs and implementation of the FMI header definitions, a new modular approach called UniFMU was recently released. It makes FMI easier to implement for programming languages other than C, and even includes support for building cross–platform binaries, eliminating the need for cross-compilation toolchains (Legaard, Tola, et al., 2021). Future versions of AutoFMU could provide a bundle of UniFMU and use it to generate the code of the final FMU, leveraging the Python support while simplifying the compilation process.

Another way to improve the quality of AutoFMU would be to have a better code generation algorithm that outputs more human readable source code. At the moment it is difficult to understand the algorithms generated by the program as it simply outputs a set of numeric values to apply to the decision strategy, *i.e* the values needed to build a linear, or logistic regression equation. Instead, it would be interesting to study other ways to generate the source code and make it look like it was written by a human. Once more this would probably be another highly complex task worth another dissertation by itself.

Finally another interesting approach to better test this program would be to apply it to other real–word scenarios and other physical units. In theory, the program is generic enough to be able to output decent results for other use cases, and comparing those results could help evaluating on which scenarios AutoFMU performs better and is more useful.

BIBLIOGRAPHY

Almeida, J. et al. (2007). “Improving hierarchical cluster analysis: A new method with outlier detection and automatic clustering”. In: *Chemometrics and Intelligent Laboratory Systems* 87.2, pp. 208–217. ISSN: 0169-7439. DOI: 10.1016/j.chemolab.2007.01.005.

Alpaydin, E. (2010). *Introduction to machine learning*. 2nd ed. MIT press. ISBN: 978-0-262-01243-0.

Ayodele, T. O. (2010). “Types of Machine Learning Algorithms”. In: *New Advances in Machine Learning*. Ed. by Y. Zhang. IntechOpen. Chap. 3, pp. 19–48. DOI: 10.5772/9385.

Bagnato, A. et al. (2016). “SysML for Modeling Co-simulation Orchestration over FMI: the INTO-CPS Approach”. In: *Ada User Journal* 37.4, pp. 215–218. URL: https://www.ada-europe.org/archive/auj/auj-37-4.pdf#page=35.

Bertsch, C., E. Ahle, and U. Schulmeister (2014). “The Functional Mockup Interface - seen from an industrial perspective”. In: *Proceedings of the 10th International Modelica Conference; March 10-12; 2014; Lund; Sweden*. Linköping University Electronic Press, pp. 27–33. ISBN: 978-91-7519-380-9. DOI: 10.3384/ecp1409627.

Beydeda, S., M. Book, and V. Gruhn (2005). *Model-Driven Software Development*. Springer. 464 pp. ISBN: 978-3-540-25613-7.

Blankenshi, G. (June 19, 2012). *Inside Tesla*. URL: https://www.tesla.com/en_GB/blog/inside-tesla-061912 (visited on 03/29/2019).

Blochwitz, T. et al. (2011). “The functional mockup interface for tool independent exchange of simulation models”. In: *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*. Ed. by C. Clauß. Linköping Electronic Conference Proceedings. Linköping University Electronic Press, pp. 105–114. DOI: 10.3384/ecp11063105.

Blockwitz, T. et al. (Nov. 19, 2012). “Functional Mockup Interface 2.0. The Standard for Tool independent Exchange of Simulation Models”. In: 9th International MODELICA Conference, Munich, Germany. Linköping University Electronic Press, pp. 173–184. DOI: 10.3384/ecp12076173.

Bowen, J. P. and M. G. Hinchey (1995). “Seven more myths of formal methods”. In: *IEEE software* 12.4, pp. 34–41. DOI: 10.1109/52.391826.

Brandl, G. (2010). *Sphinx Documentation*. URL: https://www.sphinx-doc.org/_/downloads/en/master/pdf/ (visited on 11/17/2021).

Broenink, J. F. (1999). “20-sim software for hierarchical bond-graph/block-diagram models”. In: *Simulation Practice and Theory* 7.5, pp. 481–492. ISSN: 0928-4869. DOI: 10.1016/S0928-4869(99)00018-X.

Caruana, R. and A. Niculescu-Mizil (2006). “An Empirical Comparison of Supervised Learning Algorithms”. In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML

'06. Association for Computing Machinery, pp. 161–168. ISBN: 1595933832. DOI: 10.1145/1143844.1143865.

Cerejeira, A. (2020a). *AutoFMU coverage report*. URL: https://codecov.io/gh/ajcerejeira/autofmu (visited on 11/18/2021).

Cerejeira, A. (2020b). *AutoFMU documentation*. URL: https://autofmu.readthedocs.io/en/latest/ (visited on 11/18/2021).

Chacon, S. and B. Straub (2014). *Pro Git*. Springer Nature. URL: https://git-scm.com/book/en/v2.

Chikofsky, E. J. and J. H. Cross (1990). "Reverse engineering and design recovery: A taxonomy". In: *IEEE Software* 7.1, pp. 13–17. DOI: 10.1109/52.43044.

Clarke, E. M. and J. M. Wing (1996). "Formal methods: State of the art and future directions". In: *ACM Computing Surveys (CSUR)* 28.4, pp. 626–643. ISSN: 0360-0300. DOI: 10.1145/242223.242257.

Colak, I. and D. Yildirim (2009). "Evolving a Line Following Robot to use in shopping centers for entertainment". In: *2009 35th Annual Conference of IEEE Industrial Electronics*. IEEE, pp. 3803–3807. DOI: 10.1109/IECON.2009.5415369.

Collberg, C. S. and C. Thomborson (2002). "Watermarking, tamper-proofing, and obfuscation – tools for software protection". In: *IEEE Transactions on Software Engineering* 28.8, pp. 735–746. DOI: 10.1109/TSE.2002.1027797.

Cordasco, I. S. (2016). *Flake8*. URL: https://flake8.pycqa.org/ (visited on 11/18/2021).

Dekking, F. M. et al. (2005). *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer Science & Business Media. DOI: 10.1007/1-84628-168-7.

Dijkstra, E. W. (1970). *Notes on structured programming*.

Fowler, M. and M. Foemmel (2006). *Continuous Integration*. URL: https://martinfowler.com/articles/continuousIntegration.html (visited on 11/17/2021).

Freedman, D. A. (2009). *Statistical Models: Theory and Practice*. 2nd ed. Cambridge University Press. ISBN: 978-0521743853. DOI: 10.1017/CBO9780511815867.

Fritzson, P. et al. (2006). "OpenModelica - A free open-source environment for system modeling, simulation, and teaching". In: *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*. IEEE, pp. 1588–1595. DOI: 10.1109/CACSD-CCA-ISIC.2006.4776878.

Functional Mockup Interface for Model Exchange and Co-Simulation (2020). Version 2.0.2. MODELISAR Consortium and Modelica Association Project "FMI". URL: https://github.com/modelica/fmi-standard/releases/download/v2.0.2/FMI-Specification-2.0.2.pdf.

Ge, Z. et al. (2017a). "Data Mining and Analytics in the Process Industry: The Role of Machine Learning". In: *IEEE Access* 5, pp. 20590–20616. DOI: 10.1109/ACCESS.2017.2756872.

Ge, Z. et al. (2017b). "Data Mining and Analytics in the Process Industry: The Role of Machine Learning". In: *IEEE Access* 5, pp. 20590–20616. DOI: 10.1109/ACCESS.2017.2756872.

Ghahramani, Z. (2003). “Unsupervised Learning”. In: *Summer School on Machine Learning*. Ed. by O. Bousquet, U. von Luxburg, and G. Rätsch. Springer Berlin Heidelberg, pp. 72–112. ISBN: 978-3-540-28650-9. DOI: 10.1007/978-3-540-28650-9_5.

Gomes, C. et al. (May 23, 2018). “Co-Simulation: A Survey”. In: *ACM Computing Surveys* 51.3, 49:1–49:33. ISSN: 0360-0300. DOI: 10.1145/3179993.

Goseva-Popstojanova, K. and A. Perhinschi (2015). “On the capability of static code analysis to detect security vulnerabilities”. In: *Information and Software Technology* 68, pp. 18–33. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2015.08.002.

Gregorian, R. and G. C. Temes (1986). *Analog MOS Integrated Circuits for Signal Processing*. Wiley–Interscience. ISBN: 978-0-471-09797-6.

Gubbi, J. et al. (2013). “Internet of Things (IoT). A vision, architectural elements, and future directions”. In: *Future Generation Computer Systems* 29.7, pp. 1645–1660. ISSN: 0167-739X. DOI: 10.1016/j.future.2013.01.010.

Hetland, M. L. (2017). *Beginning Python: from novice to professional*. 1st ed. Apress. ISBN: 978-1590595190. DOI: 10.1007/978-1-4302-0072-7.

Hosmer Jr., D. W., S. Lemeshow, and R. X. Sturdivant (2013). *Applied logistic regression*. 3rd ed. Vol. 398. John Wiley & Sons. ISBN: 978-0470582473. DOI: 10.1002/9781118548387.

Janai, J. et al. (2020). “Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art”. In: *Foundations and Trends® in Computer Graphics and Vision* 12.1–3, pp. 1–308. ISSN: 1572-2740. DOI: 10.1561/0600000079.

Jazdi, N. (May 2014). “Cyber physical systems in the context of Industry 4.0”. In: *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*. 2014 IEEE International Conference on Automation, Quality and Testing, Robotics, pp. 1–4. DOI: 10.1109/AQTR.2014.6857843.

Kelly, S. and J.-P. Tolvanen (2008). *Domain-specific modeling: enabling full code generation*. John Wiley & Sons. ISBN: 978-0-470-03666-2. DOI: 10.1002/9780470249260.

Kernighan, B. W. et al. (1988). *The C Programming Language*. 2nd ed. Prentice Hall Englewood Cliffs. ISBN: 978-0131103627.

Kim, K.-D. and P. R. Kumar (2012). “Cyber–Physical Systems: A perspective at the Centennial”. In: *Proceedings of the IEEE* 100.Special Centennial Issue, pp. 1287–1308. DOI: 10.1109/JPROC.2012.2189792.

Kinsman, T. et al. (2021). “How Do Software Developers Use GitHub Actions to Automate Their Workflows?” In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 420–431. DOI: 11.1109/MSR52588.2021.00054.

Knight, J. C. (May 19, 2002). “Safety critical systems: challenges and directions”. In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE ’02. New York, NY, USA: Association for Computing Machinery, pp. 547–550. ISBN: 978-1-58113-472-8. DOI: 10.1145/581339.581406.

Larsen, P. G., N. Battle, et al. (2010). “The overture initiative integrating tools for VDM”. In: *ACM SIGSOFT Software Engineering Notes* 35.1, pp. 1–6. ISSN: 0163-5948. DOI: 10.1145/1668862.1668864.

Larsen, P. G., J. Fitzgerald, et al. (2016). “Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project”. In: *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*. IEEE, pp. 1–6. DOI: 10.1109/CPSData.2016.7496424.

Lausdahl, K. et al. (May 25, 2016). *INTO-CPS FMU Builder. Web service for cross-compilation of FMUs*. The INTO-CPS Association. URL: https://sweng.au.dk/fmubuilder (visited on 12/15/2021).

Lee, E. A. (May 2008). “Cyber Physical Systems: Design Challenges”. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC). ISSN: 2375-5261, pp. 363–369. DOI: 10.1109/ISORC.2008.25.

Legaard, C. M., T. Schranz, et al. (2021). “Constructing Neural Network-Based Models for Simulating Dynamical Systems”. In: *ACM Computer Survey* 1.1. DOI: 10.1145/1122445.1122456.

Legaard, C. M., D. Tola, et al. (2021). “A Universal Mechanism for Implementing Functional Mock-up Units”. English. In: *Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications, SIMULTECH 2021, Online Streaming, July 7-9, 2021*. Ed. by G. Wagner et al. 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications : SIMULTECH 2021 ; Conference date: 07-07-2021 Through 09-07-2021. SCITEPRESS Digital Library, pp. 121–129. DOI: 10.5220/0010577601210129.

Lehtosalo, J. et al. (2014). *Mypy. Optional Static Typing for Python*. URL: http://mypy-lang.org/ (visited on 11/18/2021).

Lethbridge, T. C., J. Singer, and A. Forward (2003). “How software engineers use documentation. The state of the practice”. In: *IEEE Software* 20.6, pp. 35–39. DOI: 10.1109/MS.2003.1241364.

Marick, B. et al. (1999). “How to Misuse Code Coverage”. In: *Proceedings of the 16th Interational Conference on Testing Computer Software*, pp. 16–18.

Martin, K. and B. Hoffman (2008). *Mastering CMake. A Cross-Platform Build System*. 4th ed. Kitware. ISBN: 978-1930934207.

Martin, R. C. (2009). *Clean Code. A Handbook of Agile Software Craftsmanship*. Pearson Education. ISBN: 978-0132350884.

Marwedel, P. (2021). *Embedded System Design. Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. 4th ed. Springer Nature. ISBN: 978-3-030-60909-2. DOI: 10.1007/978-3-030-60910-8.

McCarthy, J. and E. A. Feigenbaum (1990). “In Memoriam: Arthur Samuel: Pioneer in Machine Learning”. In: *AI Magazine* 11.3, pp. 10–10. DOI: 10.1609/aimag.v11i3.840.

Mecklenburg, R. (2004). *Managing Projects with GNU Make. The Power of GNU Make for Building Anything*. 3rd ed. O'Reilly Media, Inc. ISBN: 978-0596006105.

Meijer, E. and P. Drayton (2004). "Static Typing Where Possible, Dynamic Typing When Needed. The End of the Cold War Between Programming Languages". In: *Revival of Dynamic Languages*. Citeseer.

Mohri, M., A. Rostamizadeh, and A. Talwalkar (Dec. 25, 2018). *Foundations of Machine Learning, second edition*. MIT Press. 505 pp. ISBN: 978-0-262-35136-2.

Monostori, L. et al. (2016). "Cyber–physical systems in manufacturing". In: *CIRP Annals* 65.2, pp. 621–641. ISSN: 0007-8506. DOI: 10.1016/j.cirp.2016.06.005.

Nakhimovski, I., E. Fredriksson, et al. (May 23, 2012). *FMI Compliance Checker. FMI Compliance Checker for validation of FMUs 1.0 and 2.0*. Modelica. URL: https://github.com/modelica-tools/FMUComplianceChecker (visited on 12/15/2021).

Neves, F. S. P. d. S. (Mar. 2021). "Reinforcement Learning for Robotic Navigation in Obstacle Scattered Environments". MA thesis. Faculdade de Engenharia da Universidade do Porto. URL: https://hdl.handle.net/10216/133320.

O'Brien, F. (2010). *The Apollo Guidance Computer. Architecture and Operation*. 1st ed. Springer Science & Business Media. ISBN: 978-1-4419-0876-6. DOI: 10.1007/978-1-4419-0877-3.

Olsson, F. (2009). "A literature survey of active machine learning in the context of natural language processing". In: 2009:06. ISSN: 1100-3154.

Otterlo, M. van and M. Wiering (2012). "Reinforcement Learning and Markov Decision Processes". In: *Reinforcement Learning*. Springer, pp. 3–42. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_1.

Pakdaman, M. and M. M. Sanaatiyan (2009). "Design and Implementation of Line Follower Robot". In: *2009 Second International Conference on Computer and Electrical Engineering*. Vol. 2. IEEE, pp. 585–590. DOI: 10.1109/ICCEE.2009.43.

Parr, T. J. (2004). "Enforcing Strict Model-View Separation in Template Engines". In: *Proceedings of the 13th international conference on World Wide Web*. Association for Computing Machinery, pp. 224–233. ISBN: 158113844X. DOI: 10.1145/988672.988703.

Payne, R. et al. (2016). *INTO–CPS Examples Compendium*. Version Version 1.0. INTO–CPS Association. URL: https://projects.au.dk/fileadmin/D3_4_Examples_Compendium_1.pdf (visited on 11/18/2021).

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830. URL: https://hal.inria.fr/hal-00650905v2 (visited on 11/18/2021).

Percival, H. J. W. (2014). *Test-Driven Development with Python. Obey the Testing Goat: Using Django, Selenium, and JavaScript*. O'Reilly Media, Inc. ISBN: 978-1-449-36482-3. URL: https://www.obeythetestinggoat.com/ (visited on 11/18/2021).

Punetha, D., N. Kumar, and V. Mehta (2013). "Development and applications of line following robot based health care management system". In: *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)* 2.8, pp. 2446–2450. ISSN: 2278–1323.

Rajkumar, R. et al. (2010). “Cyber-physical systems: the next computing revolution”. In: *Design Automation Conference*. IEEE, pp. 731–736. DOI: 10.1145/1837274.1837461.

Raymond, E. S. (2003). *The Art of Unix Programming*. Addison-Wesley Professional. ISBN: 978-0131429017.

Ronacher, A. (2008). “Jinja2 Documentation”. In: *Welcome to Jinja2—Jinja2 Documentation (2.8-dev)*.

Russell, S. J., P. Norvig, and E. Davis (2010). *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall Series in Artificial Intelligence. Upper Saddle River: Prentice Hall. 1132 pp. ISBN: 978-0-13-604259-4.

Rütz, S., M. Sjölund, T. Beutlich, et al. (Oct. 13, 2013). *CSV Compare. Tool to compare curves from one csv files with curves from other csv files using an adjustable tolerance*. Modelica. URL: https://github.com/modelica-tools/csv-compare (visited on 12/15/2021).

Samuel, A. L. (1959). “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3, pp. 210–229. DOI: 10.1147/rd.33.0210.

Samuelson, P. and S. Scotchmer (May 2002). “The Law and Economics of Reverse Engineering”. In: *The Yale Law Journal* 111.7, p. 1575. ISSN: 00440094. DOI: 10.2307/797533.

Schwab, K. (2017). *The Fourth Industrial Revolution*. Currency. ISBN: 978-1524758868.

Shi, J. et al. (2011). “A survey of Cyber-Physical Systems”. In: *2011 international conference on wireless communications and signal processing (WCSP)*. IEEE, pp. 1–6. DOI: 10.1109/WCSP.2011.6096958.

Snijders, T. A. B. and R. J. Bosker (1994). “Modeled Variance in Two-Level Models”. In: *Sociological Methods & Research* 22.3, pp. 342–363. DOI: 10.1177/0049124194022003004.

Stallman, R. M. et al. (1999). *Using and Porting the GNU Compiler Collection*. Free Software Foundation. ISBN: 1-882114-37-X.

Stallman, R. M. et al. (2003). *Using the GNU compiler collection. For gcc version 9.2.0*. GNU Press Website: http://www.gnupress.org. URL: https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc.pdf#page=229.

Stephenson, N. (1999). *In the beginning... was the command line*. Avon Books New York. ISBN: 978-0380815937.

The INTO–CPS Association (2016). *Line Follower Robot*. URL: https://github.com/INTO-CPS-Association/example-line_follower_robot (visited on 11/18/2021).

Thule, C. et al. (2019). “Maestro: The INTO-CPS co-simulation framework”. In: *Simulation Modelling Practice and Theory* 92, pp. 45–61. ISSN: 1569-190X. DOI: 10.1016/j.simpat.2018.12.005.

Tukey, J. W. (1962). “The Future of Data Analysis”. In: *The Annals of Mathematical Statistics* 33.1, pp. 1–67. ISSN: 00034851. DOI: 10.1214/aoms/1177704711.

Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization*. James Turnbull. URL: https://dockerbook.com/.

Van Rossum, G., J. Lehtosalo, and Ł. Langa (2014). *PEP 484 – Type Hints*. URL: https://www.python.org/dev/peps/pep-0484/ (visited on 11/17/2021).

Van Rossum, G., B. Warsaw, and N. Coghlan (2001). *PEP 8 – Style Guide for Python Code*. URL: https://www.python.org/dev/peps/pep-0008/ (visited on 11/17/2021).

Visual C++ Documentation (2019). Microsoft. URL: https://docs.microsoft.com/en-us/cpp/build/reference/md-mt-ld-use-run-time-library/ (visited on 07/19/2019).

Wing, J. M. (1990). "A specifier's introduction to formal methods". In: *Computer* 23.9, pp. 8–22. DOI: 10.1109/2.58215.

Wolf, W. (2009). "Cyber-physical Systems". In: *Computer* 42.03, pp. 88–89. ISSN: 1558-0814. DOI: 10.1109/MC.2009.81.

Part I

APPENDICES

A

AUTOFMU REFERENCE MANUAL

This chapter contains the reference manual for AutoFMU program, built from the documentation distributed with the source code. This online version of this manual is available at `autofmu.readthedocs.io`.

A.1 USER GUIDE

A.1.1 *Installation*

Install with pip from PyPI:

```
$ pip install autofmu --user
```

Or download the source code with Git and install locally:

```
$ git clone https://github.com/ajcerejeira/autofmu.git
$ cd autofmu/
$ pip install . --user
```

Compilers

To correctly build an FMU this program needs to compile the generated C source into a shared library, therefore it requires the installation of C compilers.

If you are using the provided Docker image to run the program then you are already able to cross compile the generated FMUs to `linux32`, `linux64`, `win32` and `win64` platforms.

Otherwise if you are using a Linux distribution, you probably already have GCC installed, so you should be able to compile FMUs for your system. If you want to share the generated FMU it is advisable to also install a cross compiler to produce the binaries for Windows platforms (like MinGW). Below are the instructions to install with `apt` and `dnf`:

DEBIAN/UBUNTU

```
$ sudo apt install gcc-x86-64-linux-gnu gcc-i686-linux-gnu gcc-mingw-w64 gcc-mingw-w64-i686
```

FEDORA

```
$ sudo dnf install gcc-x86_64-linux-gnu mingw64-gcc mingw32-gcc
```

A.1.2 *Usage*

```
$ autofmu "dataset.csv" --inputs "x" "y" --outputs "z" -o "My_Model.fmu"
```

This will read the `dataset.csv` file, select the `x`, `y` and `z` columns and find an approximation of the relation between the inputs and the outputs. Based on this relation, the sources files for the FMU will be generated and compiled, resulting in the `My Model.fmu` file ready to be used for simulations.

A.2 API REFERENCE

A.2.1 *autofmu*

Automatic FMU approximation tool

autofmu.main

Main entry point for running the program from the command line.

`main(args=none)`

Execute the program in a command line environment.

PARAMETERS *args* – sequence of command line arguments

autofmu.cli

Utilities for exposing a command line interface of the program.

`create_argument_parser()`

Create an argument parser object to process command line arguments.

RETURNS An argument parser object

autofmu.generator

Utilities for generating valid FMUs.

`generate_fmu(dataframe, name, inputs, outputs, outfile, strategy)`

Generate a valid FMU model.

PARAMETERS

- *dataframe* – dataframe that contains the data used for the approximation
- *name* – name of the model as used in the modeling environment
- *inputs* – variable input names
- *outputs* – variable output names
- *outfile* – path to the file to write the FMU
- *strategy* – strategy to use to find the approximation (e.g, “linear”)

`generate_model_description(name, identifier, guid, inputs, outputs)`

Generate a valid FMI 2.0 model description Extensible Markup Language (XML) document.

PARAMETERS

- *name* – name of the model as used in the modeling environment
- *identifier* – short class name according to C syntax, for example, “A_B_C”
- *guid* – globaly unique identifier that identifies this model
- *inputs* – variable input names
- *outputs* – variable output names

RETURNS Valid FMI 2.0 model description XML document

`generate_model_source(guid, inputs, outputs, strategy, result)`

Generate a valid FMI 2.0 C source code implementation.

PARAMETERS

- *guid* – globaly unique identifier that identifies this model
- *inputs* – variable input names
- *outputs* – variable output names
- *result* – a result from an approximation calculation

RETURNS Valid C source code that implements the FMI

autofmu.utils

General utilities

`compile_fmu(model_identifier, fmu_path)`

Compile the C sources files of an FMU.

Extracts the FMU into a temporary directory, calling CMake to build the FMU, copying the generated library back into the FMU file. If MinGW is installed, it also cross compiles the FMU for Linux and Windows.

PARAMETERS

- *model_identifier* – short class name according to C syntax, for example, “A_B_C”

- *fmu_path* – path to the FMU file

`run_cmake(source_dir, build_dir, variables=none)`

Run CMake command and build the targets.

Roughly equivalent to running the following two commands:

```
cmake -S source_dir -B build_dir
cmake --build build_dir
```

PARAMETERS

- *source_dir* – path to source directory
- *fmu_path* – path to build directory
- *variables* – a mapping between variable names and their values, e.g, `"CMAKE_PROJECT_NAME": "Unicorn"` would be passed as `DCMAKE_PROJECT_NAME=Unicorn` in the command line

`slugify(value, allow_unicode=false)`

Convert a string to a URL slug.

Convert to ASCII if 'allow_unicode' is False. Convert spaces or repeated dashes to single dashes. Remove characters that aren't alphanumerics, underscores, or hyphens. Convert to lowercase. Also strip leading and trailing whitespace, dashes, and underscores.

A.3 COMMAND LINE REFERENCE

```
usage: autofmu [-h] [-o FILE] [-v] [-V] --inputs VARIABLE [VARIABLE ...]
               --outputs VARIABLE [VARIABLE ...] [-s {linear,logistic}]
               FILE
```

A.3.1 *Positional Arguments*

`file` CSV files that contain the datasets for training the FMU model.

A.3.2 *Named Arguments*

`-o, --outfile` file to output the generated FMU model. Default: "model.fmu".

`-v, --verbose` run the program in verbose mode. Default: False.

`-v, --version` show program's version number and exit.

`--input` list of names of the model input variables.

`--outputs` list of names of the model output variables.

`-s, --strategy` strategy to use to deduce the approximation. Possible choices: “linear”, “logistic”. Default: “linear".

A.4 LICENSE

MIT License

This work was developed in the context of the Erasmus+ exchange program, under the component code H509A1 of the learning agreement, in coordination with Aarhus University in Denmark.