

**Hamburg University of Applied Sciences
Faculty Life Sciences**

**Development of a digital assistant to facilitate the use of model
checking in Product Engineering Processes**

Master Thesis

Process Engineering, Master of Science (M.Sc.)

submitted by

Ahmad Nasruddin, Bin Mohd Azlan



Hamburg

07. October 2022

1st Supervisor:	Prof. Dr. Margret Bauer	(HAW Hamburg)
2nd Supervisor:	M.Sc. Hassan Hage	(Helmut-Schmidt-University)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources or resources, and that I have explicitly marked all material, which has been quoted either literally or by content from the used sources.

.....

Date

.....

Signature

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen bzw. Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Hamburg, am

.....

Unterschrift

Abstract

With increasing computing power and advances in electronic technology, embedded systems which consist of a combination of hardware and software to perform a certain function, can nowadays execute more sophisticated software, leading to the implementation of more functions and capabilities. Systems with high functionality, such as autonomous systems, are unfortunately often also safety-critical systems. In safety-critical systems, failures can cause high costs and even endanger human lives. Experience has shown that the impact of software failures can range from minor inconvenience to catastrophic consequences, such as the explosion of the Ariane 5 heavy lift launch system. The high complexity of the software system makes it extremely difficult to detect subtle errors or to replicate them during testing or prior to deployment. The concept of verifying the correctness of software or hardware through testing has often proven inadequate for these complex systems.

In this paper, the focus is placed on the Product Engineering Processes (PEP) of the automotive industry, in particular on discrete manufacturing, in which goods, e.g. automobiles, are produced that consist of many individual parts that must be assembled along a production line. The need to ensure the reliability and integrity of the embedded software systems in the automotive industry has led to the introduction of the Automotive Software Performance Improvement and Capability dEtermination (A-SPICE) standard, which supervises the manufacturing and operation of software systems and requires verification of the correctness of these systems. Essentially, A-SPICE provides guidelines and best practices for automotive suppliers to ensure safety and thus prevent the occurrence of catastrophic events.

One method of ensuring safety of embedded systems, preventing errors and defects in software, and thus, complying with the A-SPICE standard, is the formal verification technique such as model checking. Model checking is a method of verifying the correctness of a system by considering all possible behaviours of a system. Probabilistic model checking, a mature technique that evolved from model checking, extends conventional model checking with tools and techniques for analysing systems that exhibit random or stochastic phenomena. Unfortunately, although model checking is a powerful tool for verifying the correctness of a system, its complex language and unfamiliar specification methods make it difficult for an organization to fully implement it or for auditors to assess compliance of the systems with A-SPICE.

Therefore, this work is concerned with the development of a digital assistant that reduces the difficulty and effort to perform model checking and thus facilitates model checking for everyone. In its development, the digital assistant applies state-of-the-art methods of Natural Language Processing (NLP), which is part of Artificial Intelligence that makes natural texts in human language understandable to computers and machines. The process begins with the user asking the digital assistant for a specific model checking result such as probability, duration or cost. The digital assistant then sends the data to the Probabilistic Symbolic Model Checker (PRISM) on the backend to compute the results based on the user's request. The digital assistant then retrieves the results from PRISM and sends the requested answer back to the user.

Acknowledgments

I would like to express my gratitude to my two supervisors, Mr. Hassan Hage and Prof. Dr. Margret Bauer, for their guidance and knowledge throughout the completion of this thesis. I would also like to thank my friends and family who supported me and offered deep insight into the work. Last but not least, I wish to acknowledge the help provided by the technical and support staff in the Life Sciences department of the Hamburg University of Applied Sciences.

Table of Contents

Abstract	i
Acknowledgments.....	ii
List of Abbreviations.....	v
1. Introduction.....	1
1.1 Motivation.....	1
1.2 Related work	5
1.3 Thesis Objective.....	7
1.4 Thesis Structure	9
2. Theoretical Foundations	10
2.1 Business Process.....	10
2.1.1 Introduction.....	10
2.1.2 Modelling Business Processes	13
2.1.3 Automotive-SPICE Standard.....	18
2.2 Model Checking.....	22
2.2.1 Formal Verification Method.....	22
2.2.2 Model Checking.....	22
2.2.3 Probabilistic Model Checking.....	24
2.3 Natural Language Processing (NLP)	34
2.3.1 Introduction.....	34
2.3.2 Overview of NLP methods	36
2.4 Artificial Intelligence: Deep Learning	41
2.4.1 Introduction.....	41
2.4.2 Artificial Neural Networks	42
2.4.3 Training of Neural Networks	45
2.4.4 Technologies and Framework.....	49
2.5 Application Programming Interface (API).....	51
3. Methodology	53
3.1 Software and Hardware	53
3.1.1 Software	53
3.1.2 Hardware	54
3.2 Technology and Framework.....	55
3.3 Program Development.....	55
3.3.1 Business Understanding.....	56
3.3.2 Model Examination	58

3.3.3	Property Specification	63
3.3.4	Modelling.....	64
3.3.5	Integration of Programming Languages.....	86
4.	Results and Discussion.....	90
4.1	User Interface.....	90
4.2	Questions and Answers	91
5.	Conclusion and Outlook	107
	References	110
	Table of Figure	117
	List of Tables	120

List of Abbreviations

AdaGrad	Adaptive Gradient Algorithm
Adam	Adaptive Moment Estimation
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
A-SPICE	Automotive Software Performance Improvement and Capability dEtermination
BoW	Bag-of-Words
BP	Basic Practices
BPMI	Business Process Management Initiative
BPMN	Business Process Model and Notation
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CORS	Cross-Origin Resource Sharing
CSL	Continuous Stochastic Logic
CSS	Cascading Style Sheets
CTL	Computation Tree Logic
CUDA	Compute Unified Device Architecture
DL	Deep Learning
DNN	Deep Neural Network
DTMC	Discrete-Time Markov Chain
ePMC	Extendible Probabilistic Model Checker
FNN	Feed-Forward Neural Network
GP	Generic Practices
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HIS	Hersteller Initiative Software
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JVM	Java Virtual Machine
LTL	Linear Temporal Logic
MA	Markov Automata
MC	Markov Chain
MCSTA	Model Checking Stochastic Timed Automata (Modest Toolset)
MDP	Markov Decision Process
MSE	Mean Square Error
NLP	Natural Language Processing
NLTK	Natural Language Toolkit
NN	Neural Network
OEM	Original Equipment Manufacturer
OMG	Object Management Group
PA	Probabilistic Automata
PARAM	Parametric Markov Models
PCTL	Probabilistic Computation Tree Logic
PEP	Product Engineering Processes
POS	Parts-of-Speech

PRCTL	Probabilistic Reward Computation Tree Logic
PRISM	Probabilistic Symbolic Model Checker
RAM	Random-Access Memory
ReLU	Rectified Linear Unit
REST	Representational State Transfer
RMSProp	Root Mean Squared Propagation
RPC	Remote Procedure Call
SMC₄PEP	Stochastic Model Checking of Product Engineering Processes
SOAP	Simple Object Access Protocol
SQuAD	Stanford Question Answering Dataset
UTF	Unicode Transformation Format
USA	United States of America
VDA	Verband der Automobilindustrie
VM	Virtual Machine
Wi-Fi	Wireless Local Area Network

1. Introduction

This chapter first presents the motivation and problem statement of this thesis, then describes the related previous work that preceded this thesis, followed by the objectives of this work, and concludes with an overview of the structure of the entire thesis.

1.1 Motivation

In 1965, Intel co-founder and chairman Gordon Moore discovered that the number of transistors in a dense integrated circuit doubled every eighteen months. This meant that computers were becoming twice as powerful every year and a half, but their cost remained the same, which indicates an exponential rate of growth. Moore found this quite astonishing and predicted that this trend might continue for a few more years, maybe five, maybe ten. Well, that was twenty, forty, soon to be sixty years ago. Moore's Law is the reason why the smartphone in our pockets is a thousand times smaller, a thousand times cheaper, and a million times more powerful than a supercomputer from the 1970s. However, Moore's Law has been slowing down over the past recent years and its exponential growth has decelerated mainly due to the physical limits of transistor growth. In particular, the number of transistors that can be integrated per unit area of a chip has reached its limit. While it is true that the number of transistors per chip no longer doubles every eighteen months, Moore's Law has led to tremendous advances in technological improvements and continues to drive innovation and progress to this day.

One of the alternatives to Moore's Law that has emerged in recent years is quantum computing. Quantum computers harness the laws of quantum mechanics, e.g. superposition, entanglement, and quantum interference, to solve problems that are too complex for classical computers, which includes performing complex computations, processing multiple data sets, and handling numerous variables. In 2002, Geordie Rose, founder of the early quantum computing company D-Wave, introduced the quantum computing version of Moore's Law, now known as Rose's Law. The idea is similar: the number of qubits in a quantum computer doubles every year. Yet, Rose's Law is considered far more impactful than Moore's Law because qubits in superposition are far more powerful than binary bits in transistors (Diamandis & Kotler, 2020). To put this in perspective, if the development of classical computers had followed Rose's Law (rather than Moore's Law), today's laptops and smartphones would have been available by 1975. Quantum computing is gaining momentum over the years, and the Figure 1 illustrates the growth in processing power of quantum computers between 1998 and 2020 (Mazhandu, 2021).

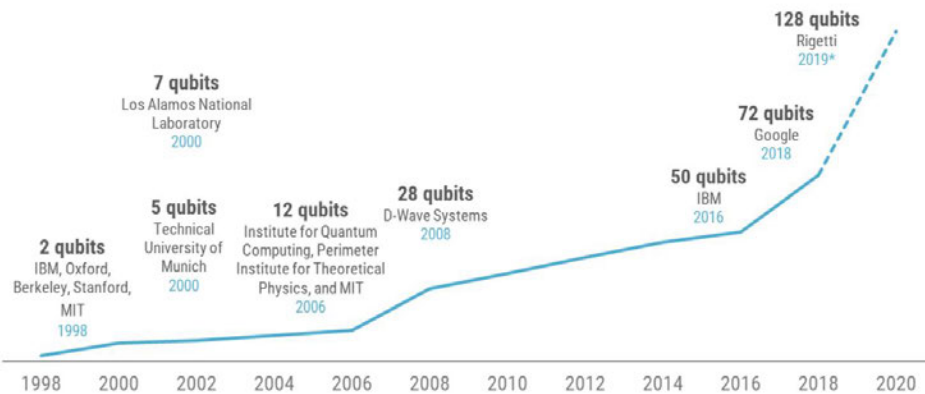


Figure 1: The growth in processing power of quantum computers between 1998 and 2020 (Mazhandu, 2021)

As shown in Figure 1, the first quantum computer tested in 1998 by the researchers from Oxford University has grown from a computational power of 2 qubits to a computational power of 128 qubits, developed in 2019 by Rigetti Computing, a start-up company in Berkeley, California, whose mission is to build the world's most powerful quantum computer in order to solve humanity's biggest problems. With the rapid increase in computing power and advances in electronic technology, the embedded system, which can be broadly defined as a device consisting of a combination of computer hardware and software designed for a specific function, is able to execute more sophisticated software, leading to the implementation of more features and capabilities. However, there is a downside to all of this. Systems with high functionalities are unfortunately also often safety-critical system. In safety-critical systems, failures can cause high costs and even endanger human lives (Drechsler, 2017).

Consider autonomous system, e.g. driverless cars and autonomous mobile robots, as an example of embedded system with high functionalities. Autonomous systems are a class of software capable of sensing the environment and responding automatically to real-world conditions without human intervention. They operate in complex and open environments with a high degree of independence and self-determination. They perceive, learn, think and act with self-awareness and respond intelligently to unforeseen changes in the environment. This differs from traditional automation, which requires predictable conditions to function (Choi, 2013). Observing the development of self-driving cars over the past decades, the performance doubles roughly every 16 months, which also indicates an exponential growth. It is therefore safe to suggest that the development trends for most embedded systems are moving from automated to autonomous systems (Zhao, 2016).

Problem Statement 1: Safety Concerns

As various tasks of autonomous systems get introduced, safety concerns to the operations have been raised because autonomous systems can be out of control enough to cause loss, injury, or death to persons or property (Choi, 2013). Safety cannot be guaranteed simply by good design because any behaviour of the system may be upset by mistakes made during its production, installation or use. Furthermore, to apply autonomous systems to execute more complex missions, the structure of systems has to become more complicated, which makes it more difficult to detect errors during testing phase, thus increases the possibility of malfunctions in the real-world.

In general, (Zhao, 2016) stated that a (distributed) computer system can fail for external and/or internal reasons. External reasons are related to the system specification itself or to the operating environment, e.g., mechanical stress, incorrect inputs, temperature, and so on. The main internal reasons for failure can be (i) random physical errors in hardware; (ii) design errors in hardware and/or software; or (iii) communication errors in a distributed environment. Software failures are usually caused by the unmanaged complexity of the system design. The increasing complexity of embedded software makes it extremely difficult to discover subtle faults or to reproduce them in a laboratory environment. Although safety-critical systems are usually designed to be fault-tolerant, experience shows that software errors are nevertheless unavoidable.

System failures can jeopardize human lives or result in high financial losses. The Ariane 5 heavy-lift space launch vehicle, which caused a financial loss of more than \$370 million, was one of the famous fatal failures and is still remembered today. At an altitude of about 3700 m, the launcher deviated from its trajectory, began to disintegrate under the high aerodynamic forces, and eventually went into self-destruct mode and exploded. The failure was caused by a software error in the inertial reference system: a 64-bit floating-point number (representing horizontal velocity) was converted to a 16-bit signed integer. As a result, the conversion failed and the information about flight direction and altitude was lost. The program was the same one that had worked perfectly in Ariane 4, while the continuous dynamic systems around it had changed. Unfortunately, in the new physical environment, the trusted code led to disaster (Zhao, 2016).

Another example, considered one of the most famous fatal software errors in history, is an error in the control system of the Therac 25 radiation therapy machine. The accident caused the deaths of six cancer patients between 1985 and 1987 due to their exposure to massive overdoses of radiation. During the two-year period, the machine occasionally gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury. During the investigation, it was determined that the root cause of the problem was twofold. Firstly, the software controlling the machine contained errors which proved to be fatal. Secondly, the design of the machine relied on the controlling computer alone for safety. There were no hardware interlocks or monitoring circuits to ensure that software errors could not cause catastrophic failures (Leveson, 1993).

These accidents have highlighted the dangers of software control of safety-critical systems and have become a standard case study in the fields of health informatics, software engineering, and computer ethics. Ensuring that the complex hardware and software components of these systems are developed with an appropriately high level of confidence in their correctness and integrity is in this respect a particularly important task (Drechsler, 2017). Therefore, communities in most countries have focused on regulatory authorities, certifications and standards to supervise the manufacturing and operation of embedded software systems. This has led to the introduction of a standard, Automotive Software Process Improvement and Capability Determination (A-SPICE), which evaluates development factors that enable assessors to determine an organization's ability to effectively and reliably deliver software products for the automotive industry (Automotive SIG, 2015). Essentially, A-SPICE provides guidelines and best practices for automotive suppliers to ensure safety and thus prevent the occurrence of catastrophic events.

To ensure the correctness of the system, classical testing approaches based on simulation or emulation have been used in the past. In some cases, it is still common practice to write software first and test it (perhaps) later. But these techniques do not scale well and reach their limits (Drechsler, 2017). The problem with these methods is that there are still undetected scenarios that can show up as errors later in practice. Some undiscovered errors at the deepest corners don't show up during testing, but can show up during system execution. Even if an error is found during testing, it is usually difficult to figure out the cause(s) of the error (Zhao, 2016). The formal verification technique may be a more suitable solution to ensure the correctness and integrity of systems. It provides techniques to verify that a system conforms to its specification and that it satisfies safety properties. Unlike testing, formal verification considers all possible behaviours of a system. The state-of-the-art formal verification method is model checking.

Model checking is a mathematically based technique that helps in ensuring the correctness of systems and is capable of analysing the quality of service properties of such systems. It is an automated verification method that systematically checks whether concurrent systems meet the specified properties (Drechsler, 2017). The model checking technique evaluates safety properties that guarantee that a particular behaviour can always or never occur under all possible situations. This is done in such a way that a model checker accepts the formal system model and the formal property as inputs and reports that the property is either satisfied or violated. This could therefore play a major role in ensuring the safety of large complex systems and help prevent accidents and catastrophic events due to software malfunction. Probabilistic model checking, a mature technique that evolved from model checking, extends conventional model checking with tools and techniques for analysing systems that exhibit random or stochastic phenomena such as message loss, unpredictable environments, errors, and delays.

Problem Statement 2: Language Barrier

However, the complex language of model checking is usually not easily understood by people who are not familiar with this method. In order to perform model checking, one must be familiar with the concepts of model checking functions and know how to properly specify a property function. Usually, only developers or people familiar with model checking have the knowledge and experience to perform such tasks. This is unfortunate since model checking is such a powerful tool for verifying the correctness of models, which can ultimately reduce safety-related accidents and catastrophes due to system failures. At the very least, every member of an organization, as well as the auditors responsible for evaluating the correctness of a system based on the guidelines of the A-SPICE standard, should be able to easily verify a process of a software system. Therefore, a solution is needed to facilitate the process of model checking and thus encourage its implementation and dissemination for everyone.

In order to make human languages understandable to machines for performing a specific functionality, research in the field of Natural Language Processing (NLP) has been conducted since the late 1940s. In the early days of NLP, machine translation was the focus of research. Nowadays, more and more research is being conducted in the areas of NLP and machine learning due to the boom in Deep Learning, which is driven by improvements in the performance of computing power, the increasing availability of data, and advances in research. NLP enables the development of technologies such as virtual assistants, e.g., Apple's Siri, Google Assistant or Amazon's Alexa, which are driven by innovative trends. Many currently emerging technologies are impossible to use without the application of NLP and almost every interaction between humans and machines has some relation to NLP (Blanck, 2019).

In this work, a new method is proposed using a digital assistant, which is developed with the help of state-of-the-art NLP and Deep Learning techniques to facilitate the model checking process and enable anyone, regardless of experience, to perform model checking. The process begins with the user asking the digital assistant for a specific model checking result, such as "what is the probability that a process will reach the final state without delay?" or "what is the total duration for the processes to reach the final state?". The digital assistant then sends the questions as data to the probabilistic model checking tool, PRISM at the backend, which performs model checking based on the user's request. The digital assistant then retrieves the result from the PRISM tool and returns it back to the user.

1.2 Related work

This section describes the previous work conducted by researchers from studies (Hage, et al., 2020) and (Hage, et al., 2022), which preceded this thesis. The work focuses on the Product Engineering Processes (PEP) of the automotive industry, in particular on discrete manufacturing in which goods, e.g. automobiles, are produced that consists of many individual parts that must be assembled along a production line. The processes in discrete manufacturing are not continuous in nature, in the sense that each process can be started or stopped individually and can run at different production rates.

PEP typically involve the processes of product development and production control. The processes can be modelled according to the Business Process Model and Notation (BPMN) standard, which is a well-known and widely used modelling language for processes in industrial PEP. This allows organizations to understand their internal process workflows in a graphical notation and to visualize the communication and dependency between different departments within the PEP (Allweyer, 2016). Essentially, BPMN enables organizations to examine the processes in their current state and determine what needs to be improved. Figure 2 shows an example of the two BPMN process models, namely product development and production control, and illustrates the interaction between these two departments and the parties involved starting from product specification up to product delivery.

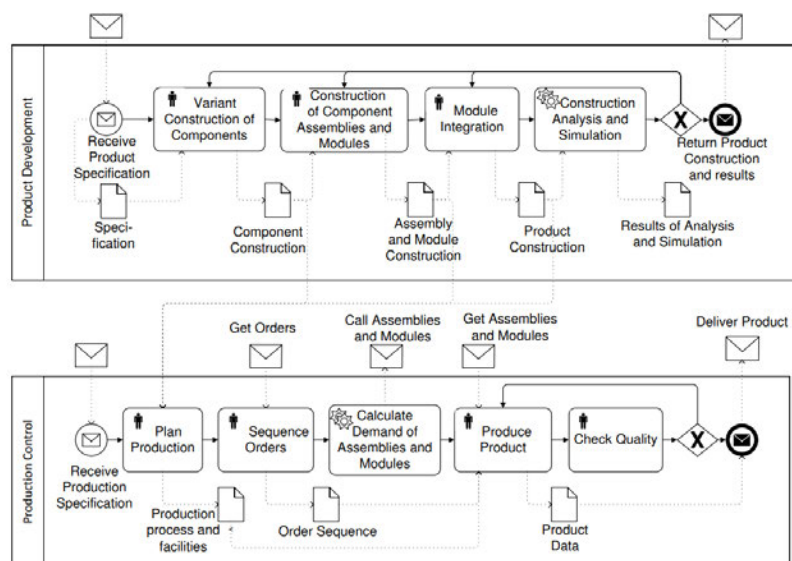


Figure 2: Process model of Product Development and Production Control (Roller & Engesser, 2014)

The first pool shown in Figure 2 shows the BPMN process model for product development. Firstly, a process request message with the product specification is sent. The first human task starts once the product specification is received and makes the variant construction of components by Computer-aided design (CAD). The data object of product specification is an input which describes, e.g. product forms, sizes, functionalities, and propositions. The designed component construction is the input for the human task construction of component assemblies and modules. The next human task integrates the modules into the product construction. A service task follows, which analyses and simulates the product construction. If the result returns any errors, the previous tasks were informed and corrected. Finally, the data objects of product construction and analysis results are sent back as response message.

The second pool in Figure 2 shows the BPMN process model of a production control workflow. The part, assembly, module, and product design are sent from product development to the production control workflow. The first human task is the production planning. Production processes and production equipment such as robots and assembly lines are planned. Computer-aided manufacturing (CAM) systems are used for this purpose. Order requests are received from distributors. The human task order sequencing creates the sequence of production orders. The next service task calculates component demand and sends a request to suppliers. After receiving the components, the next task produces the product. The next task checks the product quality. If there are any quality defects, rework has to be performed. Finally, the produced product is delivered to the distributor (Roller & Engesser, 2014).

Overview of previous work

This paper is an extension of a series of work conducted in the past. A summary of this work is shown in Figure 3.

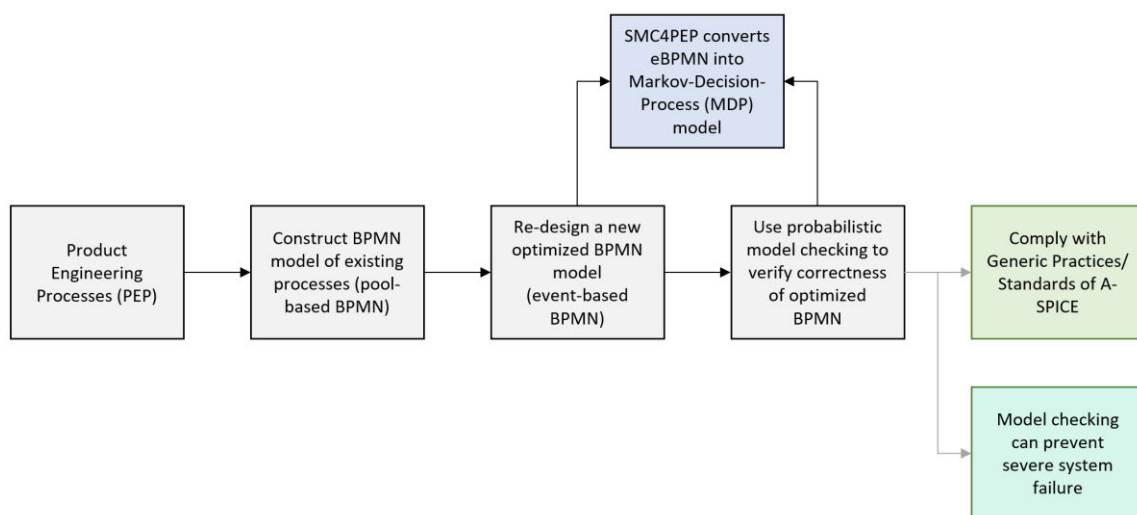


Figure 3: An overview of the previous work

In previous works, as can be seen in Figure 3, PEP of an automotive manufacturer were initially modelled according to the BPMN standard. This constructed BPMN model of the existing processes can also be referred to as the pool-based BPMN. The pool-based BPMN is subsequently re-designed and converted into an optimized BPMN process model (also referred to as event-based BPMN). This step removes redundant processes in the PEP and enables interchange between different levels of process abstraction to ensure consistency and traceability of processes in the PEP, as specified in the A-SPICE standard (Hage, et al., 2022).

Using the Stochastic Model Checking of Product Engineering Processes (SMC4PEP) tool, the revised BPMN process model or the event-based BPMN is then converted into a stochastic control process called the Markov Decision Process (MDP) model, described by the syntax of the PRISM language, in order to capture the stochastic properties of the PEP, and consequently ensure consistency and enable automated verification of the correctness of the processes using the probabilistic model checking tool, PRISM with respect to the requirements of the A-SPICE standard. With the help of the PRISM tool, processes can be simulated and analysed to determine, among other parameters, the probability of reaching the final state of the process, the duration of the processes, and the total number of process transitions and states.

1.3 Thesis Objective

The main objective of this work is to use a digital assistant with the help of NLP to reduce the difficulty and effort in applying the model checking method and to enable anyone, regardless of experience, to perform the method. This could encourage the implementation and dissemination of model checking, reduce software system failure and thus help in compliance with the A-SPICE standard.

Further objectives

In addition to the main objective, this work aims to achieve other side benefits by implementing the digital assistant in an organization. Firstly, the digital assistant can help to perform model checking at any time of the day or night, 24/7 (Han, 2019). The user is able to pose a model checking query to the digital assistant and receive instant results. Therefore, any member in the organization, including the assessor of the A-SPICE standard, who wishes to verify that a process has no deadlocks or delays, can use the digital assistant for this purpose at any time without having to wait for the next responsible developer to be available.

This in turn can lead to another advantage when using the digital assistant. Since the program is available at any time of the day, developers who are normally responsible for verifying processes by performing model checking do not have to be present at all times. Therefore, the time a developer normally spends in an organization verifying processes can be shortened and utilized for other more business-critical tasks. The extra cost of hiring or assigning new developers to perform such tasks can also be reduced (Han, 2019).

Furthermore, provided that the digital assistant program is well developed and works exactly as it is intended, mistakes that could potentially be committed by the developers can be avoided. Computers are inherently more precise than humans, as they work exactly as they are programmed, assuming there are no errors in the code. They tend to perform tasks more efficiently and with consistent accuracy, unlike humans who tend to make mistakes and errors (Nocks, 2008). This is especially detrimental when it comes to verifying processes that deal with safety-critical systems, where a small error can have major consequences. The use of digital assistants can therefore help to solve this issue.

The following section describes what the program should ideally do to achieve the objectives of this thesis. Ideally, the program should:

- Flawlessly understand all the intentions of the questions posed by the user, provided that the question falls within the scope of the specified and trained program parameters.

- Successfully exchange data and information with the probabilistic model checking tool, PRISM.

- Load and read MDP model file uploaded by the user and display error if there is no MDP model uploaded.

- Be visualized on a web-designed front-end with user-interface.

- Automatically extract all states and transitions and determine the final state of each variable from the uploaded MDP model.

- Perform all model checking functions based on properties specified by the A-SPICE standard and return accurate results back to users in the shortest amount of time.

- Share list of Generic Practices of the highest achievable process capability level of A-SPICE.

- List all variables and modules names of the MDP model, build the model and provide the model construction time.

- Be designed such that model checking can be performed with minimal effort, with additional help from a template of question examples provided on the home screen to avoid errors in predicting the answer to the question asked.

- Be designed to be deployable on multiple platforms and operating system with minimal effort.

- Process the extracted data into a suitable format for machine learning solutions.

- Provide a scalable machine learning pipeline for the system's models, starting from the raw intent data.

1.4 Thesis Structure

This section describes the structure and organization of the thesis. The thesis is structured as follows: The first chapter deals with the motivation and problem statement of this work, previous related works, and the objectives of the work. Chapter 2 deals with the theoretical foundations and general theory of the various concepts, methods and models used in this work. This includes chapters on Business Processes, Model Checking, Natural Language Processing (NLP), Deep Learning and Application Programming Interface (API). Chapter 3 deals with the methodology used in the development of the program. This includes the software and hardware, as well as the technologies and frameworks that were used in the construction of the program, including the connection architecture of the different programming languages. Chapter 4 presents the final outcome of the program and discusses the results of the message predictions from the program. Chapter 5 provides a conclusion and outlook that can be drawn from the results and the ability to achieve the objectives of the thesis.

2. Theoretical Foundations

This chapter describes the theoretical foundations and general theory of the various concepts relevant in this thesis. The chapters start with Business Processes, followed by Model Checking, Natural Language Processing (NLP), Deep Learning and conclude with Application Programming Interface (API).

2.1 Business Process

The goal of this section is to provide an overall understanding of the concepts and technologies that are relevant in business processes. The section begins with the definition and examples of business processes, then it briefly outlines the business process lifecycle, subsequently describes the modelling techniques of business processes, e.g. BPMN, the concepts of BPMN, its symbols and elements, as well as examples and descriptions of simple and multiple process implementations. This section also provides an overview of the business process applied as a use case in this thesis, namely the Product Engineering Processes (PEP). It then discusses the challenges of business processes and concludes by describing the A-SPICE standard, which was introduced primarily to address these challenges.

2.1.1 Introduction

“A business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal (Weske, 2012).”

A business process can be described a set of related, structured activities and steps performed by the people or equipment in an organization in order to achieve the basic organizational goals such as profit maximization and customer satisfaction.

Business Process Example

Business processes occur at all organizational levels, and they differ from one industry to another. For example, business processes in mining differ from those in finance, healthcare, aviation, security, etc., due to the varying nature of these industries. Examples of business processes in some of the well-known industries are presented in Table 1.

Table 1: Business Process Examples (Comindware, 2019)

Industry	Business Process Examples
Manufacturing	Order processing, engineering change control, product development and assembly, product line process, quality assurance, maintenance
Finance	Invoicing process, risk management process, billing process
Health	Medical assessment process, drug approval process, financial process
Banking	Customer onboarding process, credit check process, risk assessment process
Travel	Agent billing, trip booking, leave management process, business travel management process
Procurement	Purchasing, invoice reconciliation, account receivable
Advertising	Cost estimating, cost approval, cost reviewing
Sales and Marketing	Product delivery process, product development process, marketing research process

The next section will consider the development of business processes by introducing the business process lifecycle.

Business Process Lifecycle

Every organization in every industry follows a cycle of business activities, all focused on achieving a specific business outcome. The effectiveness of these activities has a direct and significant impact on the performance of the business. The business process life cycle consists of phases that are interrelated. The phases are organized in a cyclical structure that shows their logical dependencies. However, it is to note that these dependencies do not imply a strict chronological order in which the phases must be executed. Numerous design and development activities are performed during each of the phases, and incremental and evolutionary approaches with simultaneous activities in multiple phases are not uncommon (Weske, 2012). The business process lifecycle is shown in Figure 4.

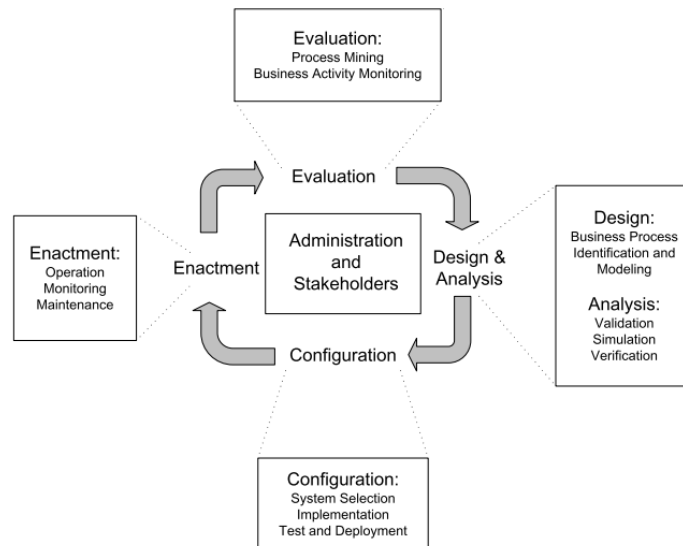


Figure 4: Business Process Lifecycle (Weske, 2012)

Design and Analysis

The lifecycle begins with the design and analysis phase, in which surveys of the business processes and their organizational and technical environment are conducted. Based on these surveys, the business processes are identified, reviewed, validated and represented by business process models. The business process is designed as it ideally should be, and the process is examined in its current state to determine what needs to be improved. Business process modelling techniques as well as validation, simulation, and verification techniques are employed in this phase. Once an initial design of a business process is developed, it must be validated. Simulation techniques can be used to support validation because certain undesirable execution sequences can be simulated that reveal deficiencies in the process model. Business process simulation also enables stakeholders to walk through the process step by step and verify that the process actually exhibits the desired behaviour (Weske, 2012).

Configuration

Once the business process model has been designed and reviewed, the business process must be implemented. It can be implemented through a set of policies and procedures that must be obeyed by the company's employees. The implementation of the business process must be tested once the system is configured. Traditional testing techniques from the software engineering field are used at the process activity level to check, for example, whether a software system exhibits the expected behaviour (Weske, 2012).

Enactment

Once the system configuration phase is complete, the business process instances can be put into effect. The process implementation phase comprises the actual runtime of the business process. Business process instances are initiated to meet an organization's business objectives, which typically occurs after a defined event, such as the receipt of an order from a customer. A monitoring component of a business process management system visualizes the status of business process instances. Process monitoring is an important mechanism for getting accurate information about the status of business process instances. During the execution of business processes, valuable execution data is collected, usually in the form of log files. These log files consist of ordered sets of log entries that indicate events that occurred during business processes. The beginning and end of an activity are typical information stored in execution logs. The log information is the basis for evaluating processes in the next phase of the business process lifecycle (Weske, 2012).

Evaluation

In the evaluation phase, available information is used to assess and improve business process models and their implementations. Execution logs are evaluated using business activity monitoring and process mining techniques. These techniques aim to determine the quality of business process models and the suitability of the execution environment. Similar considerations apply to process mining, which has several possible applications. When the execution logs of traditional information systems are generated, they can be used collectively as a starting point for developing business process models. Evaluation of existing business process models is another application area of process mining (Weske, 2012).

2.1.2 Modelling Business Processes

This section introduces the foundation of business process modelling concepts specifically, the BPMN model and introduces the subdomains of BPMN, namely modelling functions, processes, core elements, and operation.

Process modelling is the graphical representation of business processes or workflows. Similar to a flowchart, the individual steps of the process are drawn so that a continuous overview of the tasks of the process in the context of the business environment is created (IBM, 2022). A process model enables the visualization of business processes so that organizations can gain a better understanding of their internal business processes for management and efficiency. Organizations are able to understand how processes work in their current state and how they can be improved (Allweyer, 2016). The Business Process Model and Notation (BPMN) has been developed as a dedicated language for the modelling of business processes and is also used as a process modelling language in related previous work.

Business Process Model and Notation (BPMN)

“The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial draft of the processes, to the technical developers responsible for implementing the technology that will perform these processes, and finally to the business people who will manage and monitor those processes (Rosemann & vom Brocke, 2010).”

BPMN is originally developed by the Business Process Management Initiative (BPMI), which published BPMN 1.0 in May 2004. In 2005, BPMI merged with Object Management Group (OMG), which has since continued to develop BPMN and released the current version 2.0 of BPMN in 2011. The objective of BPMN is to support business process management for both technical and business users by providing a notation that is intuitive to business users and yet can represent complex process interpretation. The main goal of BPMN is to provide a standard notation that can be easily understood by all business stakeholders (Allweyer, 2016).

Core elements of BPMN

The BPMN language is based on flowcharts and graphical notations. Different types of BPMN flow object symbols and connecting objects are visualized in Figure 5.

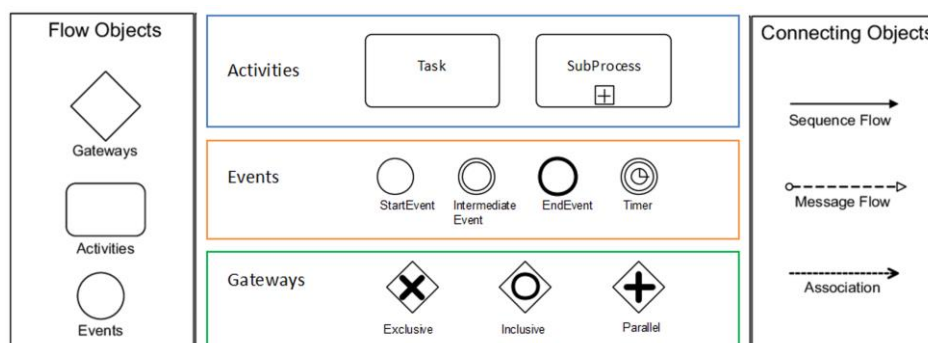


Figure 5: BPMN flow object symbols and connecting objects (Suchenia, et al., 2017)

Essentially, a process model is a graph with nodes and edges, where the nodes can be activities representing a task, the events representing relevant events that influence the process, and the gateways representing branches in the process execution (Weske, 2012). Typically, processes are triggered by a start event, have activities or tasks and gateways (decision points) in the middle, and finish with an end event. Complex processes also contain sub-processes and intermediate events, as well as different types of gateways to show how the workflow progresses through the diagram. For example, an exclusive gateway has only one option for movement, an inclusive gateway has options based on the decision made at the gateway, and parallel gateways represent two simultaneous tasks in a workflow. Symbols are used to connect flow objects such as sequence flows, message flows, and associations. Flows are represented by dashed or straight lines with arrows, while associations use a dotted line to show that certain documents or artifacts are associated with an event or gateway (IBM, 2022).

Process Model

A process model is used as a blueprint for a set of process instances that represent the individual executions of that process. Each process instance consists of multiple activity instances. These activity instances go through different lifecycle states. Once the process is instantiated, each activity is initialized. This puts the activity instances into the “*init*” state. Once the incoming flow of an activity is triggered, it activates the instance and changes the state to ready. When the execution of the activity starts, the state changes to activity life cycle running. Finally, the activity instance is terminated once execution is complete. In some cases, the process execution chooses another path while the activity has not started yet. This causes the activity to enter the “*skipped*” state. Also, a running activity may enter the terminated state due to an exceptional situation. During the execution of a process instance, an activity can be instantiated again after it has been terminated or cancelled (Weske, 2012).

Single process implementation

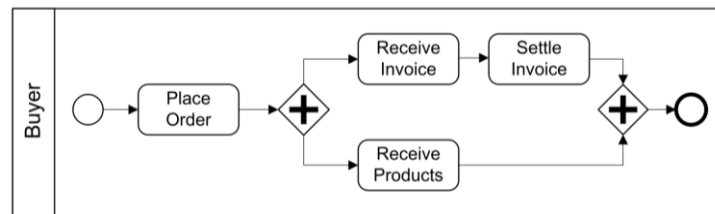


Figure 6: Process of placing an order of a buyer (Weske, 2012)

An illustration of a business process of a buyer placing an order is shown in Figure 6. The process starts with the placement of an order from the buyer, before two parallel branches are opened. In one branch, the invoice is received and settled. In the other branch, the product is received. When both branches are completed, the buyer's business process is complete.

Multiple process implementations

The reseller's and the buyer's business processes can possibly interact in the following ways. First, the buyer sends an order message to the reseller. Second, the reseller receives this message in a start event. The order information is then extracted from the message, and order processing begins. Third, the reseller sends an invoice and ships the ordered products. Forth, the buyer receives the invoice. Fifth, the buyer pays and settles the invoice. Finally, the buyer receives the ordered products. The interacting business processes are shown in Figure 7.

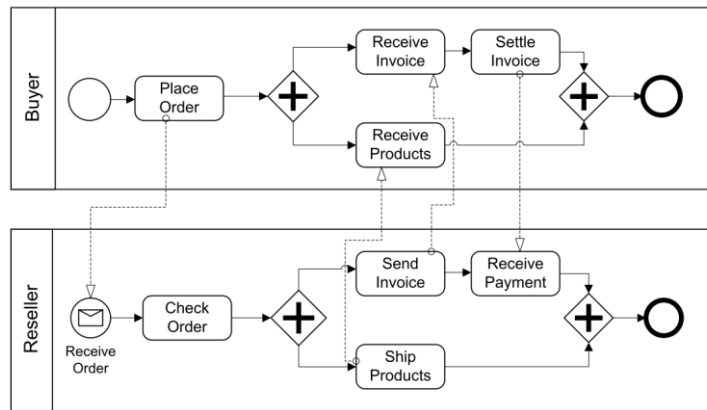


Figure 7: Interacting multiple business processes (Weske, 2012)

The interacting activities of the reseller's business process and the buyer's business process are connected by dashed arcs representing the message flow. The message flow can represent electronic messages sent and received as well as the transport of physical objects, such as ordered products. The interactions of a set of business processes are described in a process choreography, which indicates that there is no central agent controlling the activities in the involved business processes. Interaction is achieved only by sending and receiving messages. In order to achieve correct interactions, the interacting business processes must agree on a common choreography before they start to interact (Weske, 2012).

Use Case: Product Engineering Process (PEP)

The use case for this work stems from a problem in the automotive industry of Product Engineering Processes (PEP). PEP, a term which is fairly common in the automotive and avionics industries, is used to describe a business process that involves the design, development, and transition to manufacturing of products such as devices, assemblies, or systems (Feldhusen & Grote, 2013).

Phases of Product Engineering Processes (PEP)

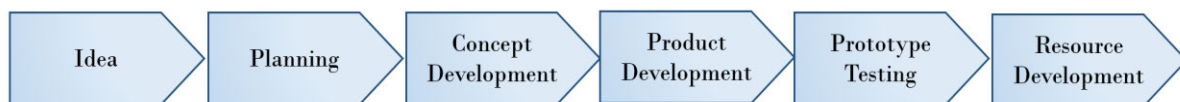


Figure 8: Phases of Product Engineering Processes (PEP) (Grochowski, et al., 2015)

Various PEP models exist in the literature. These are mainly subdivided into the idea, product planning, concept development, product development, prototype testing and production resource development phases as can be seen in Figure 8. In the first phase, ideas are generated and new approaches are developed. The focus of this phase is on idea generation and evaluation. It serves to define the task and to determine the product function, basic design principles and manufacturing processes. In the planning phase, a specification and a product design are developed and product planning, detailing and design are carried out. During concept development, the product concept is defined on the basis of the selected product ideas. Here, the focus is placed particularly on the product features and characteristics that are valuable to customers. Tools for evaluating the concepts are sampling and competitor comparisons. This phase also includes the drafting of a requirements specification. The subsequent product development phase influences the structure, design and manufacture of the final product. It includes development, design and the formation of multidisciplinary working groups. In order to validate product development processes and to verify the results of the phases with regard to their quality and integrity, the department of product control defines several product quality criteria that the product must meet before a new process step can be started (Grochowski, et al., 2015).

Challenges of Business Processes

PEP are usually involved in developing complex products or providing complex services, such as manufacturing highly engineered products or fulfilling extremely dynamic customer needs. The business processes and consequently, embedded software system of these organizations often exhibit a high degree of complexity. In these environments, creating safe, reliable yet efficient processes is a major challenge. The complexity makes traditional manual validation difficult and error-prone. Real-world processes are even more complex because they can interact with elements that are beyond the control of the processes and whose behaviour is unknown. This motivates the need for probabilistic decision points when modelling business processes, since the outcomes of complex decisions within a process can be random and/or unpredictable. They can usually be determined using probabilities (Herbert, et al., 2014).

The prevailing approach to develop business processes is to construct conceptual process models, which then lead to a design where the process is analysed to see if it meets its objectives and where any necessary adjustments can be made before a costly implementation (Herbert, et al., 2014). Typically, the design should satisfy complex categories of properties, such as:

- **Validation properties:** testing whether the business process behaves as expected in a particular context; e.g., does the traceability of process comply with A-SPICE?
- **Performance characteristics:** Assessing the ability to meet requirements in terms of processing times, service levels, and resource utilization or other quantitative factors; e.g., how long does it take to deliver a product?
- **Stochastic properties:** Assessment of the probability of behaviour within the system; e.g., is there a greater than 98% probability that the product will be delivered in 2 weeks?

The need for safety and reliability of business processes specifically in automotive industry has led to the introduction of the A-SPICE standard, which is explained in more detail in the next section.

2.1.3 Automotive-SPICE Standard

Automotive Software Performance Improvement and Capability dEtermination (**A-SPICE**) was developed on the basis of ISO 15504 by the Automotive Special Interest Group, an organization consisting of Audi AG, BMW Group, Daimler AG, Fiat Auto S.p.A., Ford Werke GmbH, Jaguar, Land Rover, Dr.Ing.h.c. F. Porsche AG, Volkswagen AG and Volvo Car Corporation (Automotive SIG, 2015). The first draft was published in 2005, with final approval in 2008 as a “*Verband der Automobilindustrie*” (VDA) standard. The major German Original Equipment Manufacturers (OEMs) Audi AG, BMW Group, Daimler AG, Porsche AG and VW AG formed an additional group called “*Hersteller Initiative Software*” (HIS). SPICE, and later Automotive SPICE, has been a mandatory requirement of HIS for mechatronics suppliers since about 2005, which led to a rapid expansion in their supplier base (Spork, 2011).

A-SPICE is used as a framework for process evaluation to determine an organization's ability to effectively and reliably deliver software products for the automotive industry. The standard can be considered a representative model for software process assessment, as assessors evaluate indicators and metrics that measure software process performance. It specifies requirements for process reference models and process assessment models. The process reference model includes several key components, namely: lifecycle processes from several process categories for the process dimension and six competence levels for the process capability dimension (Automotive SIG, 2015).

Process Capability Level

The process capability level is a central idea of the A-SPICE standard. The levels enable the user to evaluate the process capability and improve it accordingly. The suppliers attest to the customer a certain degree of process capability or maturity from A-SPICE. This process capability level is determined in supplier assessments and put to the test. In this context, compliance with the selected processes is decisive. In order to achieve a certain level of process capability, defined model elements must be implemented in the own process environment. A-SPICE defines six levels of process capability from level 0, being the lowest, to level 5, being the highest (Wilfinger, 2017).

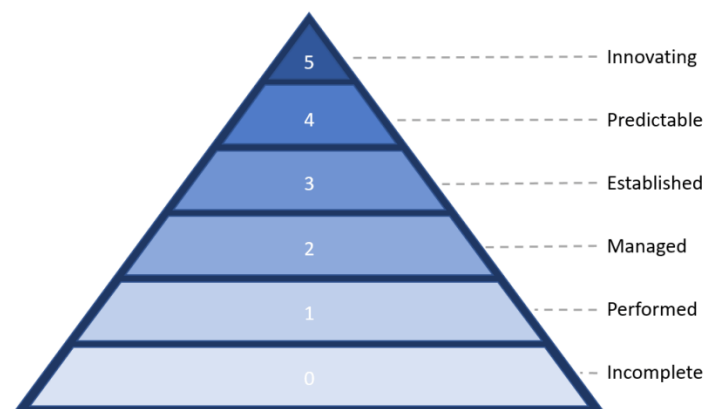


Figure 9: Process Capability Level according to A-SPICE (CiGi Technology Limited, 2022)

Since A-SPICE is not only concerned with compliance with the required processes, but also with maturity and process capability for improvements and process innovations, the degree of compliance is described analogously to the Figure 9:

Level 0 (Incomplete) describes an incomplete process. The process is not implemented, or the purpose of the process is not fulfilled. Project successes are quite possible, but are a result of coincidence.

Level 1 (Performed) is defined as a state where everyone is clear on what needs to be done. This can be described in terms of Basic Practices. Each process has its own set of Basic Practices that should be mastered at Level 1. The implementation of these practices leads to intermediate deliverables, the so-called work products.

Level 2 (Managed) signifies that everyone understands what is good and what is not good. In particular, there are document templates and checklists for the work products to be produced in the project. Responsibilities are clarified and team members receive proper training regarding their role in the project. All actions and work steps are planned and their execution is verified. It is no longer a matter of individual Basic Practices of the individual processes, since from stage 2 onwards a large number of Generic Practices must be fulfilled.

Level 3 (Established) describes the realization that it is good to record the positive experiences made. This is done in the form of process definitions. In this way, improvements can be made across project boundaries. The process definitions are continuously adapted for use in projects, so that there is also room for improvement. In particular, experience from past projects is factored in.

Level 4 (Predictable) assumes that the defined process has already been carried out several times and that figures and data are therefore available to describe process performance. Statistical observations can be conducted on the processes carried out and defined upper and lower limits exist for the individual process parameters. This enables preventive process maintenance.

Level 5 (Optimizing) describes the continuous improvement of processes. Technological innovations are researched, their usability with the applied process is evaluated and implemented for optimization. This level has the highest requirements and is rarely certified.

Generic Practices

A-SPICE outlines Basic Practices (BP) and Generic Practices (GP) as evaluation benchmarks for an organization's compliance with the standard. The team of auditors documents and evaluates the processes according to the BP and GP (Automotive SIG, 2015). Currently, the highest achievable process capability level of the use case in this work is level 2. For this reason, only GP level 2 processes are described in this section.

Part 2.1: Performance management process attribute

- **GP 2.1.1 Identify the objectives for the performance of the process.**

Performance objectives are identified based on process requirements. The scope of the process performance is defined and assumptions and constraints are considered when identifying the performance objectives.

- **GP 2.1.2 Plan the performance of the process to fulfil the identified objectives.**

Plan(s) for the performance of the process are developed. The process performance cycle, key milestones for the performance of the process and process activities and tasks are defined. Schedule is also defined and aligned with the approach to performing the process.

- **GP 2.1.3 Monitor the performance of the process against the plans.**

The process is performed according to the plan(s). Process performance is monitored to ensure planned results are achieved and to identify possible deviations.

- **GP 2.1.4 Adjust the performance of the process.**

Process performance issues are identified. Appropriate actions are taken when planned results and objectives are not achieved. The plan(s) are adjusted, as necessary. Rescheduling is performed as necessary.

- **GP 2.1.5 Define responsibilities and authorities for performing the process.**

Responsibilities, commitments and authorities to perform the process are defined, assigned and communicated. Responsibilities and authorities to verify process work products are defined and assigned. The needs for process performance experience, knowledge and skills are defined.

- **GP 2.1.6 Identify, prepare, and make available resources to perform the process according to plan.**

The human and infrastructure resources necessary for performing the process are identified made available, allocated and used. The individuals performing and managing the process are prepared by training, mentoring, or coaching to execute their responsibilities.

- **GP 2.1.7 Manage the interfaces between involved parties.**

The individuals and groups involved in the process performance are determined. Responsibilities of the involved parties are assigned and the interfaces are managed. Communication is assured between the involved parties. Communication between the involved parties is effective.

Part 2.2: Work product management process attribute

- **GP 2.2.1 Define the requirements for the work products.**

The requirements for the work products to be produced are defined. Requirements may include defining contents and structure. Quality criteria of the work products are identified. Appropriate review and approval criteria for the work products are defined.

- **GP 2.2.2 Define the requirements for documentation and control of the work products.**

Requirements for the documentation and control of the work products are defined. Such requirements may include requirements for (1) distribution, (2) identification of work products and their components and (3) traceability.

- **GP 2.2.3 Identify, document and control the work products.**

The work products to be controlled are identified. The work products are documented and controlled in accordance with requirements. The work products are made available through appropriate access mechanisms. The revision status of the work products may readily be ascertained.

- **GP 2.2.4 Review and adjust work products to meet the defined requirements.**

Work products are reviewed against the defined requirements in accordance with planned arrangements. Issues arising from work product reviews are resolved.

The next subchapter discusses methods to validate the correctness and integrity of the software systems and thus, comply with the GP outlined in A-SPICE.

2.2 Model Checking

2.2.1 Formal Verification Method

Formal verification techniques, in particular model checking, provide a powerful and rigorous approach to establish the correctness of complex systems. Improvements in the efficiency and usability of this technology mean that it can now be applied to the design phase of a wide range of computer systems, from microchips to device drivers. In the past, classical approaches based on testing with simulation or emulation have been used. However, these techniques do not scale well and reach their limits. Correctness can only be guaranteed by using formal methods (Drechsler, 2017).

Formal Verification vs. Statistical Simulation

One of the approaches to testing systems is to predict their behaviour through statistical simulations, such as a Monte Carlo simulation. A statistical simulation typically uses statistical sampling, where the problem is analysed using a set of randomly generated samples and measuring what fraction of the random set satisfies a property to determine the probability of a property. When analysing a business process, a simulation approach involves developing a model that reflects the behaviour of a process, including data and resource information, and then conducting simulation experiments to better understand the effects of running that process. The main difference between simulation-based verification and formal verification is that the former requires input vectors while the latter does not. The approach in simulation-based verification is to first generate input vectors and then derive reference outputs. In formal verification, the thought process is reversed. The user first specifies what output behaviour is desired, and then leaves it to the formal verifier to prove or disprove it. The user is not concerned with the input specification at all (Herbert, et al., 2014).

Although statistical simulation can be useful in some cases, only formal verification is complete, in the sense that it does not exclude any point in the input space of a problem. Statistical simulation can examine some situations but cannot observe all behaviours. Safety properties that guarantee that a particular behaviour can always or never occur must be evaluated under all possible situations, which is simply not possible with the simulation method (Herbert, et al., 2014).

2.2.2 Model Checking

Model checking is a technique used to prove the correctness of hardware or software systems in an automated way. The goal of this technique is to predict the system behaviour, or more precisely, to formally prove that all possible executions of the system meet specified requirements. In contrast to testing, model checking examines all possible behaviours of a system. While testing can only find errors, model checking can also prove their absence. It allows to express properties to which the answer is "yes" or "no", such as: It is never the case that, traffic lights "A" and "B" are red at the same time (Oldenkamp, 2007).

Model Checking Approach

An overview of the model checking approach can be seen in Figure 10. A model checker accepts the formal system model and the formal property as inputs and returns, slightly simplified, one of these three results. It either reports that the property is satisfied or violated, and these reports are guaranteed to be correct, provided the implementation is done correctly, or reports the third result, which is that the model checker has run out of computing resources (Hensel, et al., 2022).

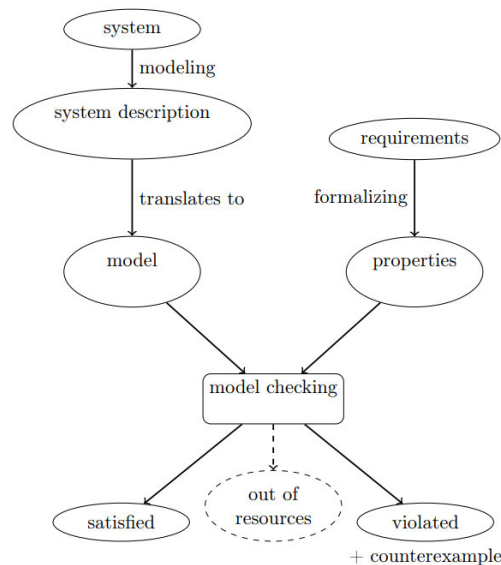


Figure 10: Overview of the model checking approach (Hensel, et al., 2022)

Applications of model checking

Typically in model checking, the structures to be checked are hardware or software systems, and the specification includes safety requirements such as the absence of deadlocks and similar critical conditions that might lead to the system crashing. However, the flexibility of the general approach has resulted in model checking being successfully applied to systems as diverse as (Herbert, et al., 2014):

- **Biological systems** where the specification may express required physical properties of chemical interaction and these are found to agree with observed biological data.
- **Robotics** where foraging robot swarms are analysed using probabilistic model checking to ensure properties, e.g. collision avoidance, timing of operations, and power consumption, are verified and that global swarm behaviour will indeed function as required.
- **Sociology** where assumptions used in models of the voting patterns for different ethnic groups in New York City, and of the effect of radon on lung cancer in the United States, are checked to see if they can be falsified by the data on which they are based.

Challenges for verification

While model checking focuses on the absolute correctness of systems, which allows the expression of properties to which the answer is "yes" or "no," such inflexible requirements are difficult or impossible to guarantee in practice. Instead, systems are subject to various stochastic phenomena such as message loss, unpredictable environments, errors, and delays. Correctness of systems is therefore less absolute. Therefore, instead of checking whether system failures are possible or impossible, it is more realistic to determine, for example, whether "*the probability of failure is at most 0.01%*" (Oldenkamp, 2007).

2.2.3 Probabilistic Model Checking

The verification of systems involving stochastic uncertainty is a prominent research challenge. Among the many techniques researched is the probabilistic model checking, a mature technique that evolved from model checking. Probabilistic model checking extends conventional model checking with tools and techniques for analysing systems that exhibit random or stochastic phenomena, or other forms of behaviour that can be approximated by randomization. Distributed algorithms and communication protocols are main examples, as they often use randomization to efficiently break symmetry. Another example is cyber-physical systems that tightly integrate software and hardware such as sensors, actuators, and microcontrollers. In particular, sensor readings can be noisy, actors may not always have the same effects, and physical components can fail. Other areas that lead to models with probabilistic aspects include randomized safety protocols and systems biology processes (Hensel, et al., 2022).

Unlike model checking, probabilistic model checking allows quantitative requirements to be specified, which enables the determination of quantitative outcomes, such as "*how reliable is my car's Bluetooth network?*", "*what is the probability that the process will complete without delay?*", or "*how secure is my bank's web service?*" (Parker, 2011).

Overview of Probabilistic Model Checking

An overview of probabilistic model checking process can be view in Figure 11.

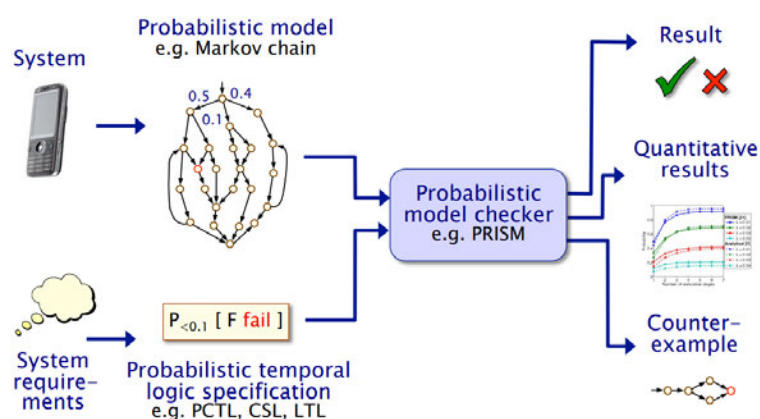


Figure 11: An overview of probabilistic model checking process (Parker, 2011)

Probabilistic model checker requires two inputs (Kwiatkowska, et al., 2009):

- **System model description of probabilistic model**, which is a description of the system to be analysed, typically given in some high-level modelling language e.g. Markov chain or Markov process.
- **Probabilistic temporal logic specification**, which is a formal specification of quantitative properties of the system that are to be analysed, usually expressed in variations of temporal logic such as:

Informally:

- “*probability of delivery within time deadline is ...*”
- “*expected time until message delivery is ...*”
- “*expected power consumption is ...*”

Formally:

- probabilistic temporal logics (PCTL, PRCTL, CSL, LTL, PCTL*, ...)
- e.g. $P_{0.1} [F \leq t \text{ reply_count} = k]$

From these two inputs, the probabilistic model checker will systematically test the validity of the property for the given model. The outcome can either be qualitative or quantitative results, or counter-example.

I. System model

The first input required for the probabilistic model checking process is a system description of a probabilistic model. A variety of formalisms have been developed for specifying probabilistic models. These include several types of Markov models, including (a) discrete-time models, that model the behaviour of a probabilistic system as a sequence of discrete time steps, e.g. Discrete-Time Markov Chains (DTMCs), Markov Decision Processes (MDPs), or Segala’s Probabilistic Automata (PA), or continuous-time models, which model the behaviour of a continuous stochastic process, e.g. continuous-time Markov Chain (MC), CTMDP, or Markov Automata (MA) (Katoen, 2016). An overview of the models is shown in Figure 12.

	Discrete time	Continuous time
Deterministic	discrete-time Markov chain (DTMC)	continuous-time MC
Nondeterministic	Markov decision process (MDP)	CTMDP
Compositional	Segala's Probabilistic Automata (PA)	Markov Automata (MA)

Figure 12: Overview of Markov models (Katoen, 2016)

In the following section, a brief introduction to the Markov Decision Processes (MDP) is provided.

Markov Decision Processes (MDPs)

MDP is a discrete-time model that allows for both non-deterministic and (discrete) probabilistic behaviour. Instead of a single distribution determining the successor states, the system can non-deterministically choose between multiple actions, each of which identifies a different distribution. After a choice is made, the successor states are resolved probabilistically and a new selection process is initiated in the successor state. Non-determinism is a valuable tool for a modeler and can be used to represent a variety of unknown aspects of a system's environment and to model the possible interaction with an adversarial environment (Kwiatkowska, et al., 2017).

A well-known example is the randomized consensus algorithm of Aspnes and Herlihy (Pogosyants, et al., 2000). In this protocol, participating processes repeatedly change a common global counter based on the outcome of a coin flip until the entire system agrees on one of two outcomes, i.e., consensus is reached. To accurately model the protocol, non-determinism can be used to consider the lack of information about the timing of simultaneous accesses to the counter (Hensel, et al., 2022).

MDP Example: Coin flip

An MDP model can also be described as a probabilistic version of a state-transition system: Each state represents a possible configuration of the modelled system, whereas each transition represents a possible progression of the system from one configuration to another across time. Transitions are labelled with quantitative information about the probability and/or timing of the transition's occurrence (e.g., 0.3, 0.5, 0.7, 1) (Kwiatkowska, et al., 2009).

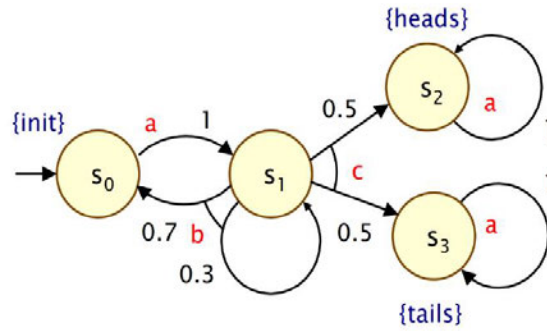


Figure 13: MDP model example of a coin flip (Parker, 2011)

Figure 13 shows an example of a simple MDP model. The model represents a process of a coin flip with four states (s_0, s_1, s_2, s_3). The process initially starts in state s_0 . After one step, the process is given only one option, to move directly to s_1 (action a). In state s_1 , a non-deterministic choice is offered between actions b and c; 1) action b gives a probabilistic choice of 0.3 (30%) to self-loop or 0.7 (70%) to return back to s_0 , and 2) action c gives a 0.5/0.5 random choice between heads and tails as an outcome.

MDP Example: Communication Protocol

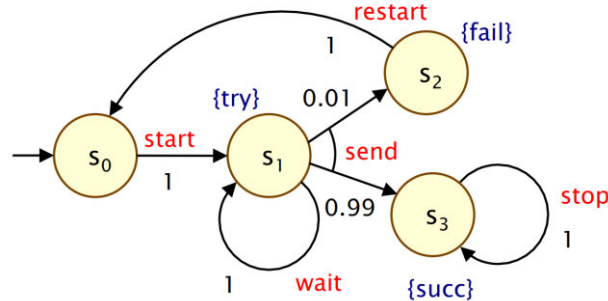


Figure 14: MDP model of a process of sending a message (Parker, 2011)

Figure 14 shows another example of a simple MDP model. The model represents a process of a communication protocol that is attempting to send a message with four states (s_0, s_1, s_2, s_3). The process starts in state s_0 . After one step, the process starts trying to send a message. A non-deterministic choice is then presented between: (a) wait one step because the channel is not ready or (b) send the message. If the latter is chosen, the message is sent successfully with a probability of 0.99 (99%) and eventually leads to the stopping of the process. On the other hand, sending the message fails with a probability of 0.01 (1%), whereupon the process restarts and returns to state s_0 .

Probabilistic modelling language

A probabilistic model consists of a set of modules representing different components of the system being modelled. The state of each module is represented by a set of variables with finite range (Kwiatkowska, et al., 2009).

```
// Component failure rates
// MTTFs: 1 month, 1 year, 1 day
const double λs = 1/(30 · 24 · 60 · 60);
const double λp = 1/(365 · 24 · 60 · 60);
const double δf = 1/(24 · 60 · 60);
// Rate for processor reboot
const double δr = 1/30;

module sensors

    // Number of sensors operational
    s : [0..3] init 3;

    // Failure of a single sensor
    [] s > 0 → s · λs : (s' = s - 1);

endmodule

module input_processor

    // State: 2=ok, 1=transient fault, 0=failed
    i : [0..2] init 2;

    // Failure of processor
    [] i > 0 → λp : (i' = 0);
    // Transient fault
    [] i = 2 → δf : (i' = 1);
    // Reboot after transient fault
    [input_reboot] i = 1 → δr : (i' = 2);

endmodule
```

Figure 15: Probabilistic model in PRISM modelling language (Kwiatkowska, et al., 2009)

Figure 15 shows an example of the PRISM modelling language: a fragment from the model description of a simple reliability case study. The model includes a set of sensors and actuators monitored and controlled by an input and output processor, respectively, and a main processor that communicates with the other components. In the model fragment, the modules representing the sensors and the input processor can be seen. As can be seen from the code marking comments, the state of the sensors is simply an integer value between 0 and 3, indicating the number of sensors currently operational, and the state of the processor is a value between 0 and 2, indicating whether it is operational or whether a transient or permanent error has occurred (Kwiatkowska, et al., 2009). The behaviour of a module, i.e., the possible changes in its state, is specified by a set of protected commands. These have the form:

$$[\text{act}] \text{guard} \rightarrow \text{rate} : \text{update};$$

where **act** is an (optional) action label, **guard** is a predicate over the variables of the model, **rate** is a (non-negative) real-valued expression, and **update** is of the form:

$$(x_1 = u_1) \& (x_2 = u_2) \& \dots \& (x_k = u_k)$$

where x_1, x_2, \dots, x_k are local variables of the module and u_1, u_2, \dots, u_k are expressions over all variables.

Rewards

This section discusses the introduction of rewards to a model, which are used to specify additional quantitative measures of interest. Rewards are simply labels of states and transitions with real values, referred to in PRISM as state rewards and transition rewards (Kwiatkowska, et al., 2009). Figure 16 illustrates the specification of rewards in the PRISM modelling language for the example model in Figure 15.

```
// Total elapsed time
rewards "time"
  true : 1;
endrewards

// Time operational
rewards "oper"
  (s ≥ 2 & i = 0) : 1;
endrewards

// Number of sensors currently operational
rewards "num_sensors"
  true : s;
endrewards

// Number of input processor reboots
rewards "num_reboots"
  [input_reboot] true : 1;
endrewards
```

Figure 16: Rewards specification in PRISM modelling language (Kwiatkowska, et al., 2009)

State rewards typically represent the rate at which rewards are accumulated. For example, the first two reward structures in Figure 16 associate a state reward of 1 for all states in which the system is operational. This can be used to determine the total accumulated time or the total operational time of the system. The third reward structure shows that state rewards can also represent an instantaneous measure of interest at a particular point in time. The final example in the figure specifies transition rewards that accumulate when a transition between states occurs. In this example, transitions marked with the action "reboots" from any state in the model receive a reward of 1 and could be used to count the number of entries (Kwiatkowska, et al., 2009).

II. Probabilistic temporal logic specification

In order to analyse a system model, a second input, which is the probabilistic temporal logic specification, is needed to specify one or more properties. The property specification language is based on temporal logic, which offers a flexible and unambiguous means of describing a wide range of properties (Kwiatkowska, et al., 2009).

In particular, the language contains operators from the Probabilistic Computation Tree Logic (PCTL), Continuous Stochastic Logic (CSL) and some of their extensions. These logics have already been shown to be capable of specifying quantitative properties and expressing a wide range of performance, reliability, and practicability properties. Other logics, such as Computation Tree Logic (CTL) and Linear Temporal Logic (LTL), are also currently being integrated into the property specification language (Kwiatkowska, et al., 2009).

Probability-based properties

PCTL and CSL allow the formulation of logical statements to specify properties such as the probability of a system reaching a particular state. For example, one can ask, "Is the probability of reaching an uncertain state of the system less than 0.1?" or "Is the probability of reaching a goal within 20 steps at least 0.9?". In addition, it allows the specification of quantitative properties such as the probability that an event will occur either within a specified time period or at any time during the lifetime of the system (Kwiatkowska, et al., 2009). Consider a system consisting of two components, A and B, each of which can fail independently. This can be expressed by the following specifications:

- $P_{=?} [F^{[0,600]} fail_A]$ - "the probability that component A fails within 10 minutes"
- $P_{=?} [F fail_A | fail_B]$ - "the probability that either component A or B fails at some point"
- $P_{=?} [fail_A U fail_B]$ - "the probability that component B fails before component A".

The P operator expresses the probability that the system is in a particular state of interest at a specific time instant (transient). The P operator is used with the $F^{[t,t]}$ operator to refer to a single instant in time.

Reward-based properties

The property specification language also includes an R operator which is used to refer to the expected value of a random variable defined in terms of a reward structure. Since a model will often be decorated with multiple reward structures, the R operator with a label of properties of the form $R_{\{time\}=?} [\dots]$ is augmented to the expected value of reward structure "time" (Kwiatkowska, et al., 2009).

The first example is the reward accumulated until a particular event occurs. Assume a simple reward structure time that assigns a reward of 1 to each state, and a Boolean state variable "fail" that is true if a system error has occurred. This can be specified as follows:

- $R_{\{time\}=?} [F fail]$ - "the mean-time-to-failure of the system, i.e. the expected amount of time that elapses before the first failure occurs".

Another operator to reason in terms of cumulative rewards is the S operator, which specifies the long-term expected rate of reward accumulation. For example, consider the model of a queue that stores jobs to be processed by a server. A reward structure "proc" that assigns a transition reward of 1 to each transition corresponding to a job processed by the server shall be assumed. Then this can be expressed as follows:

- $R_{\{proc\}=?} [S]$ - *"the throughput of the system, i.e. the expected steady-state rate of job completion"*

State-of-art Probabilistic Model Checking Tools

The state-of-art probabilistic model checking tools currently available to analyse and verify probabilistic models include PRISM, Storm, Modest Toolset (MCSTA), Model Checker for Parametric Markov Models (PARAM), and extendible Probabilistic Model Checker (ePMC) (Hage, et al., 2022). The choice of selecting the tool is motivated by the following considerations (Herbert, et al., 2014):

- I. **Support for quantitative and stochastic properties:** Model checking provides built-in native support for verification of quantitative and stochastic properties using the PCTL logic. In its extended form, PCTL*, support for verification of qualitative properties is also available.
- II. **Model size:** while models of business processes can have considerable complexity, their size is relatively modest compared to the complexity of semiconductors or biological systems for which model checking has been successfully used. Therefore, model checking is suitable for models that have much higher complexity than business processes.
- III. **Tool support:** A wide range of tools for model checking has been developed, with some of the best known tools having more than two decades of development time behind them.

The Probabilistic Symbolic Model Checker or PRISM is used as the probabilistic model checking tool in this work. Therefore, only this tool is discussed in the following section.

PRISM

PRISM is an open-source probabilistic model checking program developed first at the University of Birmingham and now at the University of Oxford. It provides support for building and analysing several types of probabilistic models: discrete and continuous-time Markov chains, MDPs, and extensions of these models with rewards (Kwiatkowska, et al., 2009). Figure 17 and Figure 18 visualize the user-interface and functionalities of the PRISM tool.

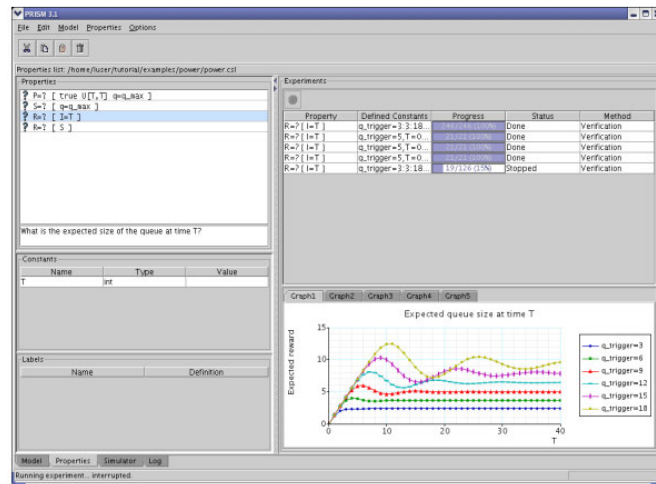


Figure 17: PRISM tool display (1) (Kwiatkowska, et al., 2009)

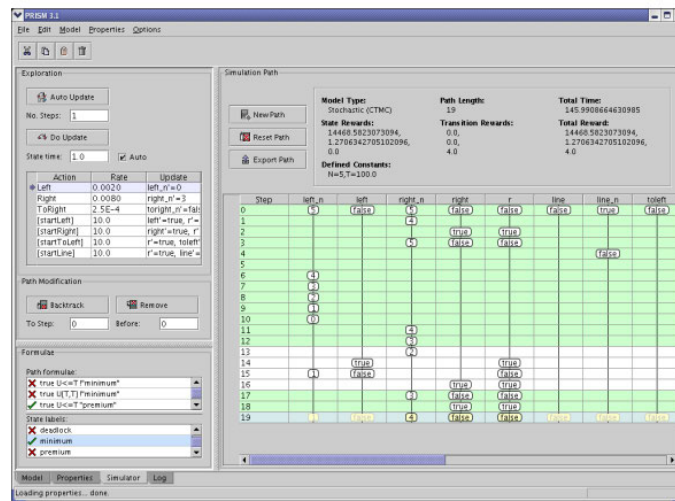


Figure 18: PRISM tool display (2) (Kwiatkowska, et al., 2017)

Functionality

User interfaces

All of PRISM's functionality is available either through a command-line version of the tool or through the Graphical User Interface (GUI). The GUI provides a more interactive entry point for newcomers to the tools, as well as invaluable functionality for more experienced users (Kwiatkowska, et al., 2009). The GUI provides:

- a model editor for the PRISM modelling language with syntax highlighting and error messages
- an editor for PRISM properties
- a simulator for exploration and debugging of PRISM models
- tools for graphical representation

Experiments

PRISM emphasizes the analysis of quantitative properties and able to examine trends in quantitative results or identifying abnormal behaviour of the system. Examples include "*the instantaneous availability of the system at time t* " for a range of time values t or "*the expected throughput of the system*" for a range of different component failure rates. The tool also includes an integrated graphing tool to visualize and display the results.

Discrete-event simulation

PRISM also contains a discrete-event simulation engine that serves two purposes. First, it forms the basis for a tool for debugging models. It can be used for either manual examination or random trace generation. Second, it enables the generation of approximate solutions to the numerical computations underlying the model checking process by applying Monte Carlo methods and sampling.

Challenges of probabilistic model checking

Probabilistic model checking extends the functionality of model checking by allowing the specification of quantitative properties such as "*the probability of delivery within the deadline*" or "*expected power consumption is ...*", instead of properties that only check the absolute correctness of systems to which the answer is either "yes" or "no". However, specifying these properties in a probabilistic model checking tool requires knowledge of the probabilistic temporal logic language and steps to specify them in the proper way. For example, to specify the probability of a process reaching the final state ("*end_state*"), the following PCTL specification must be provided for the property in the PRISM tool:

- $P_{=?} [\text{"end_state"}]$

For the reward property, the following must be specified for the property to determine the minimum reward of a process reaching the end state ("*end_state*"):

- $R_{=?} [\text{"end_state"}]$

These property specifications should not be too challenging for developers or people familiar with probabilistic model checking. However, the proper specification method of temporal logic is usually unknown and not easily understood by the general public, and the language of the logic is considered to be relatively complex by individuals who have no prior experience with this method. In order to perform model checking correctly, one must be able to correctly specify a property function and understand the concepts of model checking functions. Usually, only developers or individuals familiar with model checking have the knowledge and experience to perform such tasks. This is unfortunate because model checking is such a powerful tool for verifying the correctness of models that can ultimately reduce safety-related accidents and disasters due to system failures.

Therefore, a solution that bridges the gap between the programming language of the model checking tool and the natural language of humans is needed, to facilitate the process of properties specification, thereby encouraging the implementation of model checking and its dissemination to everyone. A technique presented in the following subchapter could be helpful in achieving this goal.

2.3 Natural Language Processing (NLP)

The section covers the definition, historical development, recent achievements and advances, and several applications of Natural Language Processing (NLP) in everyday life. In addition, various techniques and methods of NLP are also described.

2.3.1 Introduction

Making machines capable of processing human natural languages effectively has been a research challenge for quite some time. In contrast to formal languages, natural languages include all languages that have evolved without formal constructs through daily writing and speaking. Natural languages are often not clearly understood because humans have optimized communication to use shorter words and to recognize the exact meaning depending on the context. This makes it a non-trivial task for machines to extract and process the meaning of a word or intent of a sentence from its context. In addition to ambiguous and multi-meaning words, linguistic styles such as irony, sarcasm, humour, and metaphors make it even more difficult to extract the intended meaning (Horvat, 2021).

Historic Development of NLP

The historic development of NLP can be described in terms of three major waves: Rationalism, Empiricism, and Deep Learning (Deng & Liu, 2018).

The First Wave: Rationalism

In the first wave, rationalist approaches supported the design of manualized rules to incorporate knowledge into NLP systems, based on the assumption that knowledge of language is pre-determined in the human mind by generic inheritance. The first wave of NLP research lasted a long time, back to the 1950s.

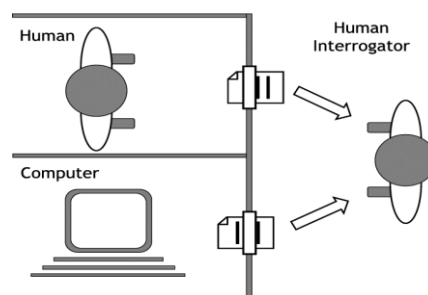


Figure 19: Turing Test (Kuipers & Prasad, 2022)

In 1950, Alan Turing proposed the Turing Test to evaluate the ability of a computer to exhibit intelligent behaviour indistinguishable from that of a human. As shown in Figure 19, this test is based on natural language conversations between a human and a computer designed to produce human-like responses and assessed by a human interrogator.

The Second Wave: Empiricism

In the second wave, empirical approaches assume that plentiful sensory input and observable surface language data are necessary and sufficient for the mind to learn the detailed structure of natural language. As a result, probabilistic models have been developed to discover the regularities of languages from a large collection. The main approaches developed in this era are called empirical or pragmatic approaches. With the increasing availability of machine-readable data and the steady increase in computing power, empirical approaches have dominated NLP since about 1990. However, the performance was far from human level.

The Third Wave: Deep Learning

In the third wave, Deep Learning uses hierarchical models of non-linear processing inspired by biological neural systems to learn intrinsic representations from language data in a way that aims to simulate human cognitive abilities. The machine learning models for NLP in this wave have sufficiently large capacity to handle the large amounts of training data. Moreover, the learning algorithms, methods, and infrastructures are also powerful enough. Recent successes in NLP have shown that it is now possible to successfully perform above the human level.

Recent successes in NLP

If standard benchmarks are to be believed, NLP systems are now remarkably good at a wide variety of language understanding tasks. The rise of Deep Learning, and in particular the success of unsupervised pre-training is driven mainly by the following two facts. On one hand, the computing power of processors has greatly increased, and on the other hand, by merging multiple computer systems, it is possible to further scale performance. The high computational effort involved in training artificial neural networks is no longer a problem due to the exponentially growth in the performance of computer chips. In addition, the database has grown massively due to the customer data collected from large corporations over the years. With the huge amount of data, the artificial neural networks could be trained for many more different types of input situations, including different writing styles of texts, intonation as well as pronunciation of words, which can vary greatly among humans (Horvat, 2021).

To illustrate the performance progress of NLP over the years, consider the following question answering dataset, the Stanford Question Answering Dataset (SQuAD). Given a paragraph from Wikipedia and a reading comprehension question, the system must find the word or phrase in the paragraph that answers the question (Jia, 2020). The outcome of the performance results are as follows:

- I. In the original SQuAD release in June 2016, the best baseline system achieved only 51.0% of the F₁ score.
- II. Crowdworkers asked to answer the same questions achieved 91.2% of the F₁ score, indicating significant room for improvement.
- III. In August 2020, the best system on the SQuAD leaderboard achieved 95.4% of the F₁ score, outperforming measured accuracy at the human level.

SQuAD is merely one NLP dataset that Deep Learning has solved. In the last few years, neural models have achieved near-human or super-human accuracy on many other well-studied NLP datasets (Jia, 2020).

Definition and Application

NLP, which aims to enable computers to intelligently process human languages, is an important interdisciplinary field that combines computer science, Artificial Intelligence (AI), and human language (linguistics) (Deng & Liu, 2018). The intersection of these various fields are shown in Figure 20.

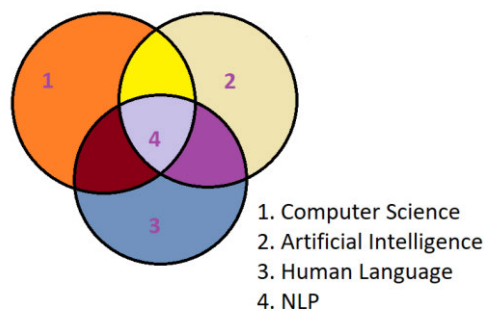


Figure 20: Venn diagram of intersection of NLP with various fields (Ranjan, 2020)

NLP applications such as speech recognition, dialog systems, information retrieval, question answering, and machine and language translation have begun to reshape the way people identify, acquire, and use information (Deng & Liu, 2018). NLP refers to a set of techniques that involve the application of statistical methods to understand text and use it to solve real-world tasks. The methods convert unstructured text and language data into structured data. When this machine-readable data is passed on, knowledge about the domain is obtained, and the semantics of the input is analysed. Based on this information, natural language answers can be generated that correspond to the input.

2.3.2 Overview of NLP methods

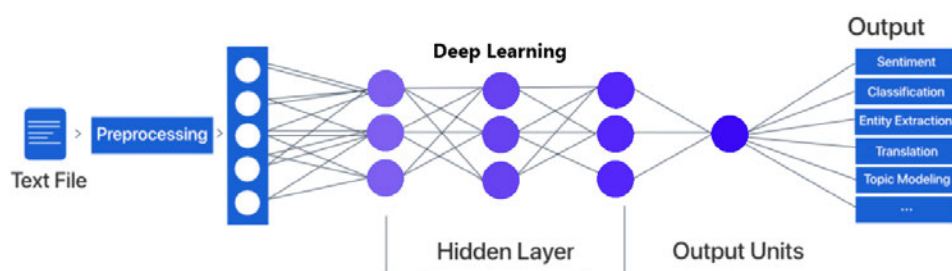


Figure 21: Overview of the NLP and Deep Learning process (Kaur, 2022)

The process of applying the NLP methods proceeds in the following steps. As shown in Figure 21, the input is first provided to the system in the form of a text file. Then it goes through a pre-processing step where various NLP methods are applied for feature selection. This is to clean the raw text data of unwanted or unimportant texts and characteristics that may affect the accuracy of the prediction performance. It then progresses to the Deep Learning phase, where it is trained using Deep Learning models to provide experience and learning to the network, and finally generates outputs based on the system requirements, where the performance of the models is evaluated using appropriate metrics. In this subchapter, only the fundamentals for the pre-processing phase are described. The method of model training with Deep Learning will be explained in more detail in the next subchapter.

Pre-processing

Pre-processing or feature selection is an important task in AI and mainly represent the step of data preparation. Especially in NLP, this task has a massive impact on the success of text analysis. This is mainly due to the unstructured and random nature of text data. In addition, machines need structure and numerical data. There are a number of approaches for this transformation task, such as the Bag-of-Words model or the vector space model (Blanck, 2019). The emphasis of this section is on the simplified methods of NLP pre-processing. In general, there are a number of additional pre-processing techniques that will not be discussed in detail in this paper.

Overview NLP Pipeline

The process of pre-processing input in the form of text using NLP methods is shown in the Figure 22. The figure is accompanied by the English phrase "is anyone there" as an example to illustrate the application of the NLP pipeline.

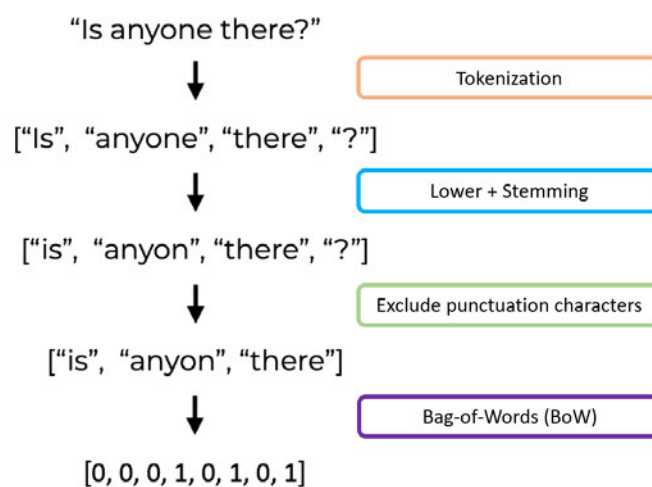


Figure 22: Overview of the NLP pipeline (Loeber, 2020)

A raw text phrase in the form of a string value is used as input. In the first step, the string is separated according to the sentence boundaries and stored in a list of strings. Then, the individual words of each sentence are extracted and tokenized and subsequently, stored sentence by sentence in lists. Next, the individual tokenized words are lowercased and stemmed to produce the root form of the words. Punctuation marks are then excluded from the word list in the next step. Finally, each sentence is transformed into a so-called Bag of Words (BoW) to convert the pattern strings into numerical data that the Deep Learning model can understand.

Tokenization

The first step in a typical NLP pipeline is the so-called tokenization. The goal of this step is to extract tokens from a given text, which can be a single word or as a complete sentence. All NLP methods start with a text dataset, also called a corpus. A corpus typically contains raw text in American Standard Code for Information Interchange (ASCII) or Unicode Transformation Format (UTF)-8 format and any metadata associated with the text. The raw text consists of a sequence of characters (bytes), but most often it is useful to group these characters into smaller units called tokens. In English, tokens correspond to words and numeric sequences separated by spaces or punctuation (Rao & McMahan, 2019).

The most common tokenization method is the whitespace/unigram tokenization. In this method, the entire text is split into words by separating them with whitespace. Essentially, tokenization involves breaking down a sentence, phrase, paragraph, or entire text document into smaller units, such as individual words or terms. (Khanna, 2022). As the example in the Figure 23 shows, the entire sentence "*Natural Language Processing*" is split into 3 unigrams or tokens, namely "*Natural*", "*Language*", and "*Processing*".

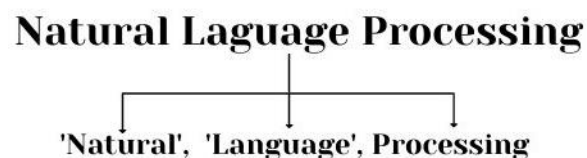


Figure 23: Tokenization of the sentence "*Natural Language Processing*" (Jain, 2021)

After tokenization, there are two possibilities for structural analysis of words represented by tokens, namely stemming and lemmatization (Horvat, 2021). Both methods pursue the same goal of identifying a base form of a modified word. In this procedure, the rules of the grammar of a language must be taken into account. For this reason, language-dependent algorithms exist for both procedures.

Lemmatization

Lemmas are root forms of words. Consider the verb *fly*. It can be applied to many different words, etc. *flow*, *flew*, *flies*, *flown*, *flowing*, and *fly* is the lemma for all these seemingly different words. Sometimes it may be useful to reduce the tokens to their lemmas in order to keep the dimensionality of the vector representation low. This reduction is called lemmatization (Rao & McMahan, 2019).

Stemming

Similar to lemmatization, stemming is a normalization technique in which a list of tokenized words is converted into shorter stems to remove redundancy. Stemming is the poor man's lemmatization. It involves using manually crafted rules to remove endings from words and reduce them to a common form, the root word (Rao & McMahan, 2019).

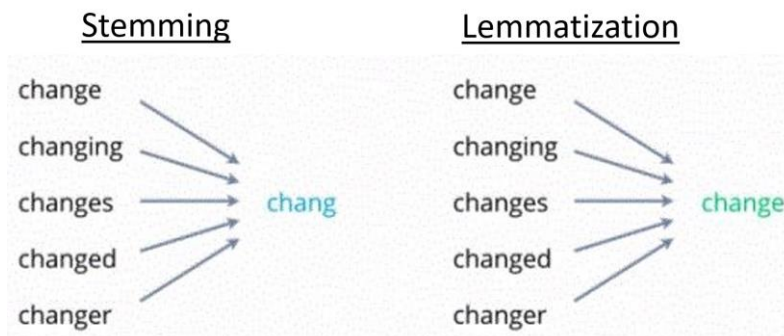


Figure 24: Comparison between stemming and lemmatization (Turing, 2022)

Both stemming and lemmatization produce the stemmed form of tokenized words. The difference is that the stem is not a real word, while the lemma is a real language word. As can be seen in Figure 24, the stem (root) is the part of the word from which suffixes or prefixes such as (-e, -ing, -es, -ed, -er) are removed. *Chang* is then the stem of the word. Thus, stems of a word or phrase can result in words that are not actual words. In lemmatization, *change*, *changing*, *changes*, *changed*, *changer* are all forms of the word *change*, so *change* is the lemma of all these words. Since lemmatization provides an actual word of the language, it is used where it is necessary to obtain valid words (Turing, 2022). Stemming follows an algorithm with steps performed on the words, which makes it faster. Lemmatization, on the other hand, uses a corpus library to create a lemma, which makes it slower than stemming (Narayan, 2020).

Bag-of-Words

It is not possible to simply provide the neural network with the input set in its current state for Deep Learning. The pattern strings must be converted into a number vector that the network can comprehend. For this purpose, each sentence is converted into the so-called Bag-of-Words (BoW). First, training words are collected, which represent all the words that the model has access to in the training data. Based on all these words, BoW is then computed for each new sentence. Each position in a BoW contains a 1 if the word occurs once in the new sentence and a 0 if none (Thanaki, 2017).

Consider the following sample document containing a list of words “Hi”, “How”, “are”, “you”, “bye”, “see”, “later” as shown in Figure 25. The BoW method is applied by deriving the term frequency of each word in the list, which can then be fed into a Deep Learning neural network algorithm. It should be noted that the grammar of the sentences and the order of the words are disregarded in this method.

	all words						
	["Hi", "How", "are", "you", "bye", "see", "later"]						
"Hi"	→	[1,	0,	0,	0,	0,	0]
"How are you?"	→	[0,	1,	1,	1,	0,	0]

Figure 25: Bag-of-Words (BoW) method (Loeber, 2020)

To derive the frequency count of the example sentence in Figure 25, the total number of occurrences of a given word in the word list is counted. First, the word, “Hi” is considered, which occurs once in the word list. Therefore, the frequency count for “Hi” in the document is given as 1. For the second entry, the word “How are you?”, contains words that all exist in the document, so the frequency count for all words - “How”, “are”, “you” is 1, respectively.

Natural Language Toolkit

The Natural Language Toolkit (NLTK) is a platform that supports the development of programs that deal with human language data. NLTK is one of the leading open source libraries for Python and is used for NLP. It offers various pre-processing methods such as stemming, tokenization or classification. The platform provides access to 50 different corpus such as “WordNet” as well as other lexical sources through simple interfaces. In linguistics, a corpus is a collection of written or spoken data. The data is usually stored digitally in a machine-readable form. Thus, a corpus consists of the text or audio data and optionally metadata (NLTK, 2022).

In the next subchapter, the fundamentals of Artificial Intelligence and Deep Learning are explained, and the step following the NLP pre-processing phase is described.

2.4 Artificial Intelligence: Deep Learning

There has been a lot of hype around Artificial Intelligence (AI) lately. Some express concern about the possibility that AI will one day take control of the human civilization, while others argue that AI will undoubtedly improve massively the life quality on this planet. In this section, some opportunities and risks of AI technology are presented.

2.4.1 Introduction

In 1956 at Dartmouth College in the United States of America (USA), the term AI was coined for the first time at a scientific conference. The initial meeting was organized by John McCarthy, then a mathematics professor at the college, who is known as one of the founding fathers of AI (Horvat, 2021). In his proposal, he stated the capability potential of AI as following:

"every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it (Nocks, 2008)."

There are now more than 60 years of research in the scientific field of AI. Thus, in theory, excellent algorithms and approaches to AI have been published for quite some time. In practice, the ideas were partly only suitable for thought experiments due to the low computing power of processors of past decades (Horvat, 2021). However, recent advances in computer processors are now powerful enough for the complex computations of the training phases of large neural networks.

Opportunities

With the accompanying performance improvements of the computer processors, AI methods are gaining practical application in industry. The potential benefits of using AI in the medical field are already being explored. The medical industry has a large amount of data that can be used to create predictive models for healthcare. AI has also been shown to be more effective than physicians in certain diagnostic contexts. In addition, it is already becoming apparent how AI will impact the world of transportation and automobiles with the rise of autonomous vehicles and autonomous navigation. AI will also have a major impact on manufacturing, including in the automotive sector. Furthermore, cyberattacks increased 600% during the pandemic as hackers took advantage of people working from home and using less secure tech systems and Wi-Fi networks. AI and machine learning will play a critical role in detecting and predicting cybersecurity threats. Similarly, in the financial world, AI will be a game changer for security, as it can process large amounts of data to predict and detect fraud (Stahl, 2021).

Risks

Attention should now be directed to the possible risks of AI. After all, it is penetrating more and more areas of the economy and human lives. One of the main concerns of AI is the development of autonomous weapons using AI systems programmed to cause destruction. In the hands of the wrong individual, these weapons could easily lead to mass death.

In addition, an AI arms race could unintentionally lead to an AI war, which would also result in many casualties. To prevent being disabled by the enemy, these weapons would be designed in such a way that they could not be easily "turned off," so humans could possibly lose control in such situation (FLI, 2022). Furthermore, some AI errors do not harm humans or the environment. However, if a tumour detection AI, for example, misclassifies a malignant tumour in an X-ray, chemotherapy could be initiated. The associated risk of such a form of therapy is severe organ damage and, in the worst case, death. In general, error rates have been greatly reduced over time by new algorithms and learning methods, but completely correct detection has not yet been achieved (Horvat, 2021).

2.4.2 Artificial Neural Networks

Artificial Neural Networks (ANN) or simply Neural Networks (NN), which is a branch of AI, are a class of pattern recognition models inspired by the human brain that occupy a large part of machine learning research and have been given their own branch in the field called "Deep Learning" due to their popularity in many applications (Nielsen, 2017). The biological neuron of the human brain and the ANN are shown side by side in Figure 26 to illustrate their resemblance.

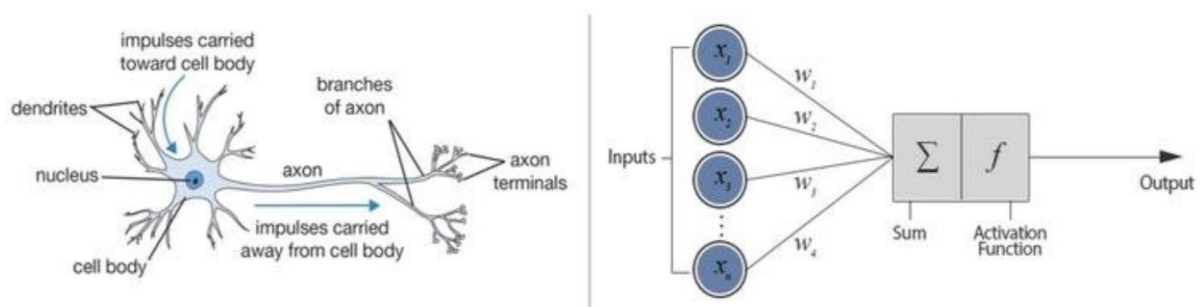


Figure 26: Biological Neuron in comparison to Artificial Neural Network (Kalla, 2021)

The layers in ANN are made up of nodes, which resembles biological neurons in the human brain and the connections resembling axons. A node is simply a location where computations take place, much like a neuron in human brain that fires when it encounters sufficient stimuli (Nielsen, 2017). A node combines the inputs $\{x_1, x_2, \dots, x_n\}$ from the data with a set of weights $\{w_1, w_2, \dots, w_n\}$ that either reinforce or dampen these inputs, thereby assigning significance to the inputs with respect to the task that the algorithm is attempting to learn (Horvat, 2021). These input weighting products are then summed, and the sum is passed through what is called a node's activation function to determine whether and to what extent the signal should be passed further through the network to affect the final result, i.e., a classification process. When the signal passes through, the neuron has been "activated" (Kalla, 2021). The operation of an artificial neuron is demonstrated in the following section using a perceptron.

Perceptron

The simplest unit of a neural network is the perceptron, which is the basic building block of neural network layers (Horvat, 2021). The perceptron was historically developed on the model of a biological neuron (Rao & McMahan, 2019).

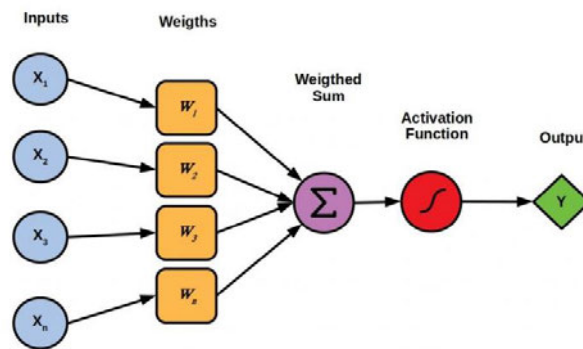


Figure 27: Perceptron unit (Rainergewalt, 2022)

As shown in Figure 27, each perceptron unit has an input (x), an output (y), a set of weights (w), and an activation function (f). In some cases, a bias (b) is also present, which is used to shift the activation function by adding a constant to the input. The weights and bias are learned from the data, and the activation function is chosen based on the network designer's intuition for the network and its target outputs. Mathematically, this can be expressed as follows:

$$y = f(wx + b) \quad (2.4.1)$$

Equation 2.4.1 shows the corresponding formula for calculating the output of the artificial neuron shown in Figure 27. It is usually the case that there is more than one input to the perceptron. Therefore, this general case can be represented with the help of vectors. That is, x and w are vectors, and the product of w and x is replaced by a dot product. The activation function, denoted here by f , is usually a non-linear function. In this example, $wx + b$ is a linear function, which is a function where the graph is represented by a straight line. So a perceptron is essentially a composition of linear and non-linear functions (Rao & McMahan, 2019).

Activation Functions

An activation function determines how the weighted sum of the input is converted into an output from a node or nodes in a layer of the neural network (Horvat, 2021). Basically, it decides if a neuron is activated or not by computing the weighted sum and adding a bias. The main idea behind applying an activation function is to introduce non-linearity into the output of a neuron to detect complex relationships in data (Rao & McMahan, 2019).

A neural network consists of neurons that operate according to the weighting, bias, and their respective activation functions. In the network, the weights and biases of the neurons are updated based on the output error. This process is called backpropagation. Activation functions enable backpropagation due to the gradients being provided along with the error to update the weights and biases. A neural network without an activation function is essentially simply a linear regression model. The activation function performs the non-linear transformation of the input and allows more complex tasks to be taught and performed (Kalla, 2021).

Rectified Linear Unit (ReLU)

The Rectified Linear Unit (ReLU) is the state-of-the-art activation function that is arguably the most important activation function deployed today. In fact, one could argue that many of the recent innovations in Deep Learning would have been impossible without the use of ReLU (Rao & McMahan, 2019). In particular, ReLUs lead to faster learning time for training Deep Learning models. Moreover, it is not prone to a vanishing gradient and is therefore widely used in practice. ReLU computes the maximum between 0 and the linear result of a neuron (Blanck, 2019). This can be seen in the following:

$$f(x) = \max(0, x) \tag{2.4.2}$$

So all a ReLU unit does is clip the negative values to zero, as shown in the Figure 28.

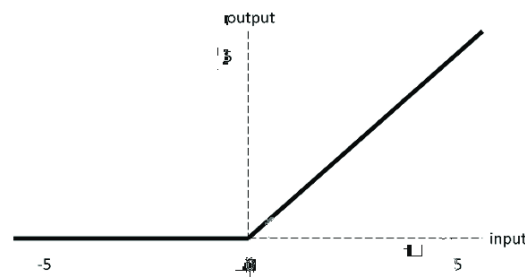


Figure 28: Rectified Linear Unit (ReLU) (H. Sultan, et al., 2019)

Deep Learning

The motivation for Deep Learning stemmed partly from addressing the dimensionality problem of machine learning algorithms that arises when working with high-dimensional data. The characteristic difference of Deep Learning compared to other smaller neural networks lies in the increased number of hidden layers to form a larger network. Individual perceptrons with the same input signals are combined into one layer (Horvat, 2021). Figure 29 illustrates the three main layers of a neural network of Deep Learning or Deep Neural Network (DNN): the input layer (left), the multiple hidden layers (centre), and the output layer (right).

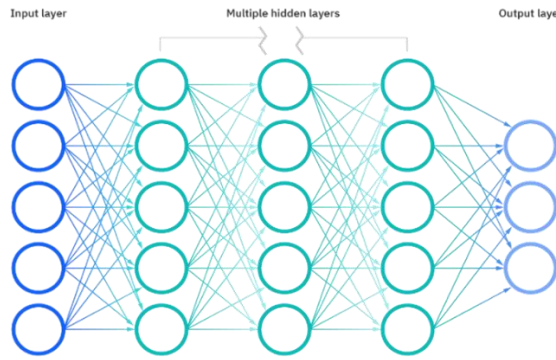


Figure 29: Deep neural network (IBM Cloud Education, 2020)

2.4.3 Training of Neural Networks

Training neural networks is a fundamental part of the Deep Learning process. It introduces experience and learning into the network and optimizes the weights in the network with the intention of achieving better predictions from the model (Blanck, 2019) (Horvat, 2021).

Supervised Training

Supervised learning is the problem of learning how to map observations to specified targets given labelled examples. Supervised learning requires the following: a model, a loss function, input training data, and an optimization algorithm (Rao & McMahan, 2019), as can be seen in Figure 30.

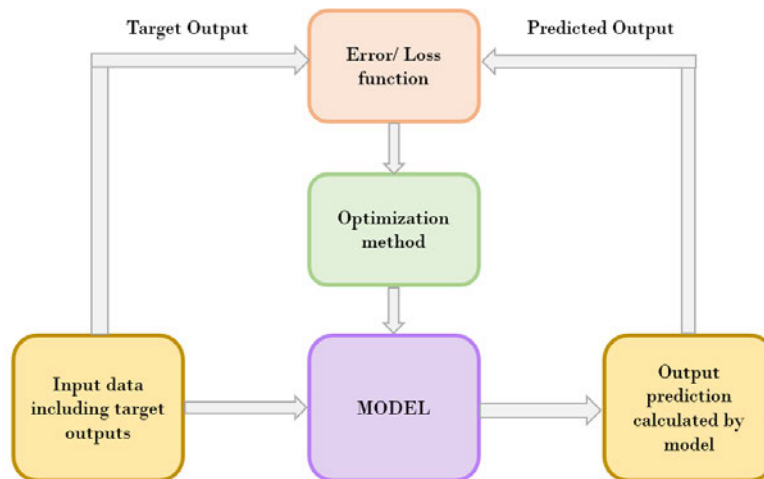


Figure 30: Process of supervised training (Sevarac, 2022)

The process of supervised training goes through the following steps (Horvat, 2021):

1. **Forward propagation:** computing the output corresponding to a prediction for a sample of input data.
2. **Error calculation:** computing the error in the discrepancy between the predicted output and the target output using loss function.
3. **Backpropagation:** the error calculated in the previous step is propagated backwards through the output layer to the input layer. In this process, the gradients are calculated considering the input value and weights.
4. **Parameter update:** the weights of the neuron connections are optimized depending on their influence on the error. The goal is to reduce the error for the corresponding input.

Essentially, the main goal of the supervised training is to adjust the parameters of the model using a gradient-based optimization algorithm so that losses are as small as possible (Rao & McMahan, 2019). The following section begins with an explanation of loss functions, followed by a description of the optimization method.

Loss Functions

All training algorithms in Deep Learning are trained with the so-called loss function. The terms loss function, cost function, and error function are often used interchangeably. This function attempts to match the algorithm's performance to the expected output. The loss function takes a true value (y), also called an "error", and a prediction (\hat{y}) as input and produces a real value. The higher this value, the worse the prediction of the model. The loss function is used in gradient descent and backpropagation to automatically update the weights based on the measurement performance (Rao & McMahan, 2019). The most common loss functions are the Mean-Square-Error (MSE) and the Cross-Entropy Loss.

Cross-Entropy Loss

Cross-Entropy Loss measures the performance of a classification model whose output is a probability value between 0 and 1. It is typically used in multiclass classification, where the results are interpreted as predictions of class matching probabilities. The target (y) is a vector with n elements representing the true multinomial distribution over all classes. The output of the network (\hat{y}) is also a vector with n elements, but represents the network's prediction for the multinomial distribution. The categorical cross entropy compares these two vectors (y , \hat{y}) to measure the loss, as shown in Equation 2.4.3.

$$L_{cross_entropy(y,\hat{y})} = - \sum_i y_i \log(\hat{y}_i) \quad (2.4.3)$$

The probability of the correct class should be strived for close to 1, while the other classes should have a probability close to 0 (Rao & McMahan, 2019).

Optimizer

Optimization is an important concept in Deep Learning and the optimizer selection process is a part of hyper-parameter tuning. During the training of the Deep Learning model, the weights of each epoch or iteration must be altered and the loss function must be minimized. An optimizer is a function that changes the neural network attributes or hyper-parameters, such as weights and learning rate. In this way, it helps to reduce the overall loss and improve the accuracy (Rao & McMahan, 2019). Optimization is usually performed with gradient-based optimization algorithms such as gradient descent or Adam optimizer.

Adam Optimizer

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first- and second-order moments. It is based on two methods, namely the Adaptive Gradient Algorithm (AdaGrad) and the Root Mean Squared Propagation (RMSProp), and is derived from **Adaptive moment estimation** method, hence the name **Adam** (Keras, 2022). It uses an approximation of the first- and second-order moments as moving averages of the gradient and of the squared gradient exponentially decaying. In practice, Adam optimizer works best and also converges quickly (Mirtaheri & Shahbazian, 2022). According to (Kingma & Ba, 2017), the method is "*computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data and parameters*". For these reasons, it is currently applied to most datasets.

Auxiliary Training Concepts

The core idea of supervised learning is quite straightforward: define a model, compute the results, use a loss function to compute gradients, and apply an optimization algorithm to update the model parameters based on the gradient. However, there are several auxiliary concepts that are important to the training process (Rao & McMahan, 2019).

Correctly Measuring Model Performance: Evaluation Metrics

The most important component outside the supervised training loop is an objective measurement of performance on data that the model has never been trained on or seen. In NLP, models are evaluated using several evaluation metrics, with "*accuracy*" being the most common one. Accuracy is simply the proportion of correct predictions on a dataset unseen during training (Rao & McMahan, 2019).

Splitting the Dataset

To achieve better model performance, it is common to split a data set into three randomly selected partitions called the training, validation, and test data sets. Splitting into three partitions is considered the easiest method to obtain good generalization because it requires only a single calculation. It has proven useful to aggregate the data set by class labels and then randomly split each set separated by class labels into the training, validation, and test data sets. A common split percentage is to allocate 70% for training, 15% for validation, and 15% for testing (Rao & McMahan, 2019).

Finding the Right Hyper-parameters

As described in the previous sections, the values of the weights are learned from the data, where the weights take real values that are adjusted by an optimizer with respect to a fixed subset of training data. In contrast, a hyper-parameter is any parameter whose value is used to control the learning process. The setting influences the number of parameters in the model and the values assumed by the parameters. The selection of the optimal hyper-parameter includes the optimal choice of a loss function, the optimizer, the learning rate(s) for the optimizer, the number of epochs i.e. number of iterations, the size of the batch and hidden layers, the patience for early termination, and several regularization decisions. It is important to note that these decisions can have a large impact on the convergence and performance of a model, and therefore the optimal choice of the various decisions should be explored systematically (Rao & McMahan, 2019).

A model is said to generalize and perform better than another model if it reduces the error not only in the samples from the training data, but also in the samples from the unseen distribution. A finite sample, which is called the training data, is a distribution of data that is an approximation or incomplete picture of the true distribution. As the model works to reduce its loss on the training data, it may "*overfit*" and adjust for characteristics that are not actually part of the true data distribution. Overfitting is a phenomenon where the model fits the input training data exactly, causing the algorithm to produce inaccurate results on unseen data, defeating its purpose (Rao & McMahan, 2019).

Regularization

Regularization, one of the most important concepts in Deep Learning, is a set of methods for restricting a model's expressiveness in order to prevent it from overfitting. One of the most common regularization methods is the weight decay method. The weight decay limits the growth of the network parameters and improves the generalization of the network. The growth of the parameters is limited by the value of the weight decay. The weights are chosen so that the network continues to classify the instance correctly, but does not memorize it. Therefore, at each step of updating the weights, all parameters are penalized by a term in the loss function (Blanck, 2019).

Early stopping

A key aspect of properly measuring model performance is using the metric to determine when to stop training. Early stopping is an important technique that uses a heuristic to prevent the overfitting of the neural networks. In this method, the training phase of the network is stopped after a certain number of epochs, which terminates further application to the training data (Blanck, 2019). This method tracks the performance of the validation dataset by evaluating and monitoring the training and validation error from epoch to epoch and determining when performance stops improving. If performance continues to not improve, the training is stopped. The number of epochs to wait before stopping the training is referred to as "*patience*". In general, the point at which a model in a dataset stops improving is called the convergence of the model (Rao & McMahan, 2019).

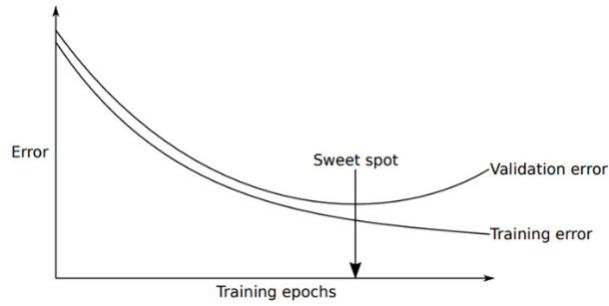


Figure 31: Early stopping method (Nielsen, 2017)

As can be seen in Figure 31, the "Sweet spot" indicates the convergence point, i.e., the point at which the neural network begins to overfit to the training data and at which, training should be stopped (Nielsen, 2017). This both prevents overfitting and reduces the training time, however at the cost of removing a small portion of the data set.

2.4.4 Technologies and Framework

In the following section, the PyTorch and TensorFlow Deep Learning frameworks are first introduced and the comparison between these two frameworks are subsequently discussed.

PyTorch

PyTorch is an open source, community-driven Deep Learning framework that supports the implementation of Deep Learning models and provides libraries to easily build ANN models. At its core, PyTorch has an optimized tensor manipulation library that provides a set of packages for Deep Learning. It is based on so-called tensors, a mathematical object that contains multi-dimensional data which represents multi-dimensional matrices. All calculations are performed by tensor operations (Rao & McMahan, 2019). As shown in Figure 32, a tensor of order zero is just a number, or a scalar component. A tensor of rank one (1st dimensional tensor) is a series of numbers, or a vector. Similarly, a 2nd dimensional tensor is an array of vectors, or a matrix. Therefore, a tensor can be generalized as an n -dimensional array of scalars.

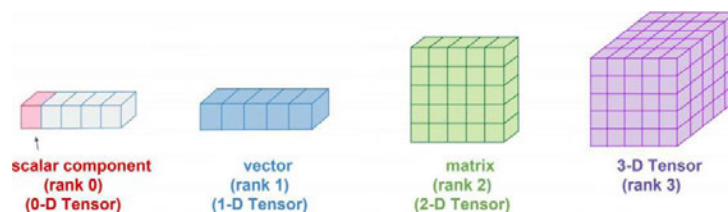


Figure 32: Tensors as a generalization of multidimensional arrays (Carlos, 2019)

PyTorch has also implemented support for Graphics Processing Units (GPUs), which makes it easier to train and analyse neural network models on GPUs. Because of the many tensor operations performed during training, it is reasonable to use them since they are designed for matrix computations (Rao & McMahan, 2019).

TensorFlow

TensorFlow is an open-source machine learning library developed by the Google Brain team. The first version was released in 2015 and is written in Python, C++, and Compute Unified Device Architecture (CUDA). Similar to PyTorch, TensorFlow focuses heavily on DNN and allows users to create and combine different types of Deep Learning models and build graphs of model performance during training (TensorFlow, 2022).

PyTorch vs. TensorFlow

Both PyTorch and TensorFlow are very popular frameworks for neural network applications. In fact, they are often considered by data scientists as the libraries of choice when it comes to developing innovative Deep learning applications or for research purposes. The PyTorch and TensorFlow frameworks have continued to improve over the years. While TensorFlow is considered a more mature library, PyTorch has also proven to be incredibly powerful. PyTorch is becoming increasingly popular in the research field, while TensorFlow is more associated with the development of AI products (O'Connor, 2022).

Dynamic versus static computational graphs

TensorFlow is a static framework that requires the computational graph to be first declared, compiled, and then executed. Although this leads to extremely efficient implementations that are useful in production and mobile environments, it can become quite burdensome in research and development. Modern frameworks such as PyTorch, on the other hand, implement dynamic computation graphs to allow for a more flexible imperative development style without having to compile the models before each execution. Dynamic computation graphs are particularly useful in modelling NLP tasks, where any input can potentially result in a different graph structure. In addition, unlike TensorFlow, PyTorch implements a tape-based automatic differentiation method that allows computational graphs to be defined and executed dynamically. This is extremely useful for debugging and also for constructing sophisticated models with minimal effort (Rao & McMahan, 2019).

Additional Libraries

In addition to PyTorch or TensorFlow, other libraries are also important for the development of a Deep Learning model. One of them is the NumPy library in Python programming language, which is briefly described in the following section.

NumPy

The most commonly used feature of NumPy is the creation of N -dimensional arrays, since Python does not support this basic data structure. It also provides implementations of linear algebra and random number generation. In addition, it is used by many other Python packages such as PyTorch to implement the tensor data structure (NumPy, 2022).

2.5 Application Programming Interface (API)

Definition

Application Programming Interfaces (APIs) are programming instructions that allow different software applications to exchange data and functions easily and securely. APIs help with the integration between different applications and devices, and enable a program to share and exchange data and functionalities with external third-party developers or entities (Bui, 2018). Terms such as libraries, frameworks, development kits, toolkits, and APIs are often used interchangeably, and they can all be considered variations of API, as the differences between them are rather small (Ala-Ilomäki, 2019).

According to (Garber, 2013), APIs are on the rise due to the fact that API technology is one of the building blocks for cloud computing and mobile technologies. In particular, he also highlights the fact that APIs are the easiest way to connect mobile applications to backend applications. For example, sharing photos between platforms is made possible through APIs (Ala-Ilomäki, 2019). Google also provides APIs for its search engine so that third-party developers can integrate them into their websites and applications.

API protocols and architectures

To enable APIs to exchange data and functionalities, clear protocols and architectures are required to represent the rules, structures, and constraints that govern the operation of an API. Currently, there are three common categories of API protocols or architectures: Representational State Transfer (REST), Remote Procedure Call (RPC), and Simple Object Access Protocol (SOAP).

Representational State Transfer (REST)

The REST architecture is perhaps the most widely used approach to building APIs. REST is not an actual software architecture, but "*a coordinated set of architectural constraints that attempts to minimize latency and network communication while maximizing the independence and scalability of component implementations*" (Fielding & Taylor, 2002). REST is based on a client/server approach that separates the front and backends of the API and provides significant flexibility in development and implementation. REST is "*stateless*", meaning that the API does not store data or state between requests. REST APIs, commonly referred to as "RESTful APIs," can also communicate directly or work through intermediate systems such as API gateways and load balancers.

Interaction process with APIs

The interaction process with APIs is often the same for most applications. Developers create applications that make calls to APIs using HyperText Transfer Protocol (HTTP) in the application. The result of the call depends on the HTTP methods used, which includes HTTP requests and HTTP responses. The process of interaction between a client application and a web server through an API is shown in Figure 33 (IBM, 2020).

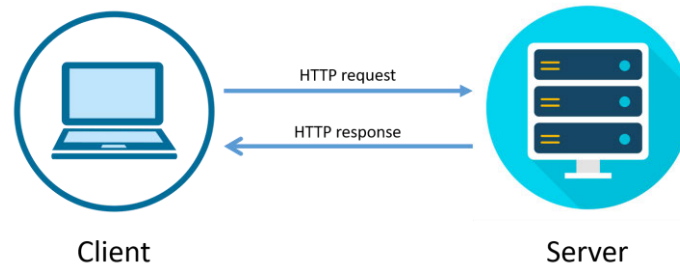


Figure 33: Interaction process between client and server via APIs (Pelayo, 2022)

1. A client application initiates an API call to retrieve information - also known as a **request**. This request is passed from an application to the web server and contains a request verb, headers, and sometimes request text.
2. After receiving a valid request, the API calls the external program or the web server.
3. The server sends a **response** to the API with the requested information.
4. The API transfers the data to the original requesting application.

Libraries

Some libraries that provide the functionalities of APIs are described in the following section.

Py4J

Py4J library allows Python programs running in a Python interpreter to dynamically access Java objects in a Java Virtual Machine (JVM). In other words, it enables the connection and exchange of data and functions between the Python and Java programming languages. Methods are called as if the Java objects were in the Python interpreter, and Java collections can be accessed using standard Python collection methods. Py4J also allows Java programs to call back Python objects (Py4J, 2009).

Flask

Flask is a lightweight framework for web applications, which is designed to make getting started with web application development quick and easy, with the ability to scale complex applications. It has become one of the most popular Python frameworks for web applications in recent years and has numerous extensions provided by the community that make it easy to add new features. This includes the extension that adds support for quickly building REST APIs. It defines the resource class, which contains methods for each HTTP method. (Pallets, 2010).

Flask-CORS

Flask-CORS is a Flask extension for handling Cross-Origin Resource Sharing (CORS) that makes cross-origin possible. To receive data from other applications, CORS must be enabled, otherwise the browser blocks the server's response for security reasons. CORS is a mechanism that uses additional HTTP headers to inform browsers to allow a web application running at one origin to access selected resources from another origin (Flask-CORS, 2022).

3. Methodology

This chapter begins with a description of the software and hardware used in the development of the program. Then, the technologies and framework applied are presented. It then outlines the steps of program development, which involves initially understanding the business requirements, subsequently examining the available data, followed by specifying the properties that fulfil the business requirements, then modelling the algorithms to perform the required functions, and finally integrating different programming languages via APIs.

3.1 Software and Hardware

The software and hardware used in this work are described in the following section.

3.1.1 Software

An overview of all software, including the libraries deployed for the development of the digital assistant program, is shown in Figure 34.

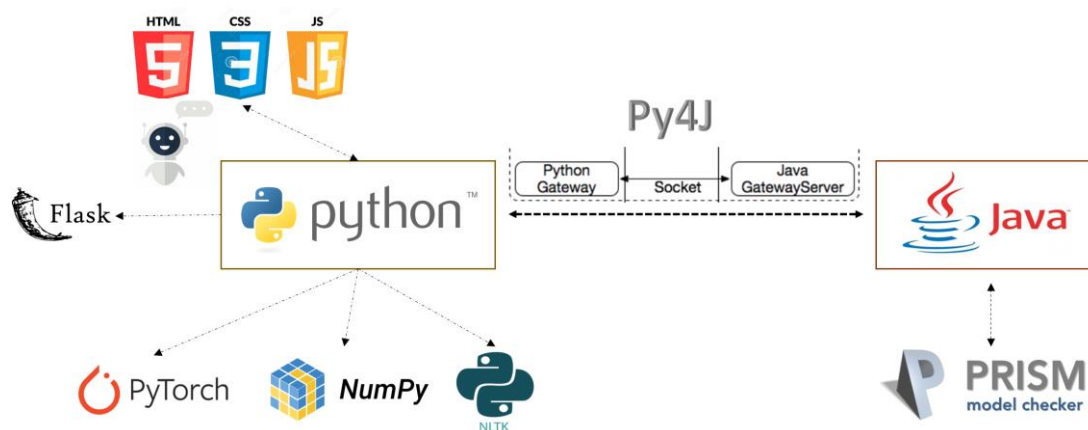


Figure 34: Overview of software of the digital assistant program

The Python programming language is used for the development of the digital assistant, which includes the NLP pre-processing methods, supported by the NumPy library, which helps to create multi-dimensional arrays of the data, and by the NLTK library, which provides access to different corpus and different pre-processing methods. PyTorch is used as a framework for Deep Learning neural network modelling, which is also based on the Python language. The PyTorch framework is chosen due to the following reasons. PyTorch framework implements dynamic computation graphs instead of static graphs to allow for a more flexible imperative development style without having to compile the models before each execution. Dynamic computation graphs are particularly useful in modelling NLP tasks, where each input can potentially result in a different graph structure. In addition, PyTorch also implements a tape-based automatic differentiation method that allows computation graphs to be defined and executed dynamically.

This is extremely useful for debugging and also for constructing complex models with minimal effort. PyTorch has also implemented support for Graphics Processing Units (GPUs), which makes it easier to train and analyse neural network models on GPUs. This was, however, not utilized in this work.

The PRISM API, provided in the Java programming language, enables the connection between the PRISM tool and Java, and the execution of PRISM functions on Java programs. The Py4J library is used to enable the connection and exchange of data and functionality between the Python and Java programs. A combination of JavaScript, Cascading Style Sheets (CSS) and Hypertext Markup Language (HTML) is used to develop the front-end visualization design. Flask library is used for web application development and acts as a framework to integrate Python and JavaScript programs via REST API. Flask-CORS is used as a Flask extension to handle Cross-Origin Resource Sharing (CORS), which makes cross-origin possible. In addition, the plots for the training results were generated using the Matplotlib library in Python. The complete software and libraries used can be summaries as follows:

- **Python** programming language
- **JAVA** programming language
- **JavaScript**, **CSS**, and **HTML** languages
- Probabilistic Model Checking tool, **PRISM**

Library

- Deep Learning **PyTorch** framework
- **Flask** Web Development
- **Flask-CORS** extension
- **Py4J**
- **NumPy**
- **NLTK**
- **Matplotlib**

3.1.2 Hardware

The experiments were run on a machine model “Microsoft Surface Laptop 4” with specification listed in Table 2.

Table 2: Hardware specifications and values

Specification	Value
Processor	AMD Ryzen 7 Microsoft Surface (R) Edition
Clock speed	2.00 GHz
Cores	8
Installed RAM (Memory Size)	16 GB
System type	64-bit operating system, x64-based processor
GPU	AMD Radeon™ Graphics

3.2 Technology and Framework

An overview of the technology and framework used in the development of the digital assistant program is displayed in Figure 35.

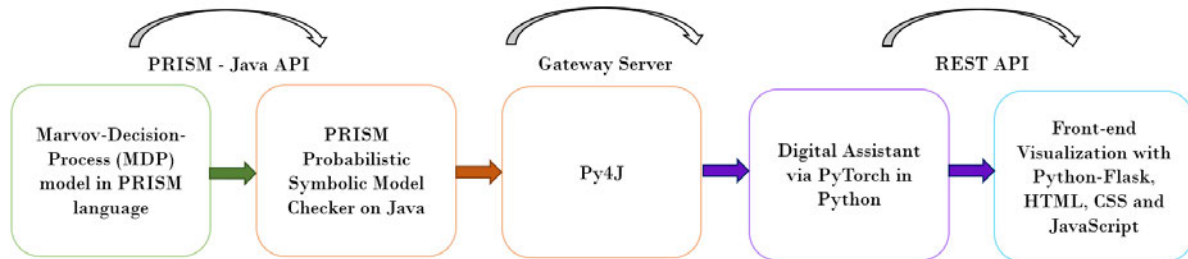


Figure 35: Technology and Framework used in the program development

1. Markov Decision Process (MDP) model is provided as input into the PRISM tool.
2. PRISM tool is integrated to Java program via PRISM API.
3. PRISM tool is connected to the Java's Py4J library via *Gateway Server*.
4. PyTorch in Python program has access to PRISM tool on Java using Python's Py4J library.
5. Python program with the help of Flask library is connected to the front-end visualization written with JavaScript, HTML and CSS via REST API.

The process begins by providing the Markov Decision Process (MDP) model, described in the PRISM language (*.prism*), to the Probabilistic Symbolic Model Checker, PRISM. Using the PRISM API written in Java language, the PRISM tool is able to connect to the Java program. The next step is to connect the Java program to the Python program, since the digital assistant is developed using the PyTorch framework, which is based on the Python programming language. The connection between Java and Python programs can be established using the Py4J library through the *Gateway Server*. Once the connection is established, the digital assistant can access, send and receive data from the PRISM tool through the API. Lastly, the front-end visualization design of the digital assistant is developed using a combination of HTML, CSS, and JavaScript languages, and the connection between them are established via REST API with the help of the Python's Flask framework.

3.3 Program Development

This section describes the process of developing the digital assistant from start to finish. The process follows the steps shown in Figure 36.



Figure 36: Process of digital assistant program development

Initially, the business requirements relevant to the use case of this work are studied and understood. Second, the available data in this work are examined and analysed. Third, the specification of the properties that meet the business requirements is defined and constructed based on the available data. Fourth, the process of modelling the algorithms to develop the program is carried out. Finally, the various programming languages used in the construction of the program are linked and integrated through the use of APIs or appropriate libraries.

3.3.1 Business Understanding

As described in Section 2.1.3, Product Engineering Processes (PEP) of the automotive industry are usually associated with the development of complex products, such as the manufacturing of highly engineered products or the operation of embedded software systems with a high degree of complexity. The need for safety and reliability in automotive business processes, especially in processes with a high degree of complexity, has led to the introduction of the A-SPICE standard.

In this work, the correctness and integrity of PEP must be verified to comply with A-SPICE. The standard defines six levels of process capability, with each level determining the quality of the processes. Currently, the highest achievable process capability level for the use case in this work is level 2. Level 2 (Managed) means that everyone in the organization understands the goal and the dos and don'ts. Specifically, there are document templates and checklists for the work products to be created in the project. In addition, responsibilities are clarified and team members receive appropriate training on their role in the project. Furthermore, all actions and work steps are planned and their execution is reviewed. From level 2, a large number of General Practices (GP) must be fulfilled.

A-SPICE defines the GP as an evaluation benchmark for an organization's compliance with the standard. The assessment team documents and evaluates the processes according to the GP. The task of this work is to verify the correctness and integrity of the processes of PEP, which were modelled in the form of MDP models, using the probabilistic model checking technique via the PRISM tool. This can lead to the satisfaction of the properties specified in the GP of A-SPICE level 2. There are in particular 4 mandatory properties ($\varphi_1, \varphi_2, \varphi_3, \varphi_4$) that have been specified in this work, which can be verified using the PRISM tool. The description of these properties and the associated GP are outlined in the following section.

Properties φ_1 , φ_2 , φ_3 , φ_4

1. **Property 1, φ_1 :** The property 1 verifies GP 2.1.7 of A-SPICE, which ensures consistency and no deadlocks in the processes and that, the final state of the process will always be achieved with the probability of 100%. The property 1 also verifies GP 2.1.2, which ensures the ability of performing the process to fulfil the identified objectives. A better description of the GPs are as follows:
 - **GP 2.1.7 (Manage the interfaces between involved parties)** specifies that the individuals and groups involved in the process performance are determined. Besides that, responsibilities of the involved parties are assigned and the interfaces are managed. Communication is assured between the involved parties and the communication is assured to be effective.
 - **GP 2.1.2 (Plan the performance of the process to fulfil the identified objectives)** specifies that plan(s) for the performance of the process are developed and the process performance cycle, key milestones for the performance of the process and process activities and tasks are defined. Schedule is also defined and aligned with the approach to performing the process.
2. **Property 2, φ_2 :** Property 2 delivers the minimum days (d) for performing the whole process.
3. **Property 3, φ_3 :** Property 3 delivers an estimation of expected days for performing the whole process obtained in accumulated working days (wd).
4. **Property 4, φ_4 :** Property 4 verifies the GP 2.1.3, which ensures that the process does not deviate from its original setting according to A-SPICE. It proves the execution of the processes based on A-SPICE guidelines. A more detailed description of the GP is as follows:
 - **GP 2.1.3 (Monitor the performance of the process against the plans)** specifies that the process is performed according to the plan(s). Process performance is also monitored to ensure planned results are achieved and to identify possible deviations.

All the properties φ_1 , φ_2 , φ_3 , and φ_4 can be summarised in Figure 37.

- › φ_1 proves the consistency of the processes by no deadlocks
- › φ_2 delivers the minimum days for performing the whole process
- › φ_3 delivers an estimation of the expected cost of the processes
- › φ_4 proves the execution of the processes based on A-SPICE guidelines

Figure 37: Description of properties φ_1 , φ_2 , φ_3 , and φ_4

3.3.2 Model Examination

Now that the business requirements, which include the guidelines and properties that must be met and verified, have been properly understood, the next step is to examine and understand the process models of the PEP. The process models to be examined in this work were revised and constructed in the previous work preceding this paper.

To recap, as described in Section 1.2, the PEP in their current state were first modelled using the BPMN standard. Then, the process models were revised and modified by removing redundant processes and enabling interchange between different levels of process abstraction to ensure consistency and traceability of the processes in the PEP, as specified in the A-SPICE. The revised BPMN models were then converted using the SMC4PEP tool into a stochastic control process model referred to as the MDP model, described by PRISM language, to capture the stochastic and probabilistic properties of the processes. Finally, the correctness and integrity of the MDP models can be verified using the PRISM tool.

The PEP covered in this thesis are a general workflow for testing an autonomous parking "pilot". The processes first describe the requirements for the test area and subsequently the main steps of executing the testing phase. Table 3 lists all the constructed MDP models provided in this work with the corresponding reward availability, number of states, and number of transitions. The models are arranged in descending order starting from the highest number of state and transition to the lowest.

Table 3: MDP models reward availability, number of total states, number of building time and model construction time

MDP Model	Name	Reward Availability	No. of states	No. of transitions
1	UC2_5levels_event	No	1977602936544	24110938911905
2	UC2_2levels_pool	No	525760512	4135546401
3	UC2_2levels_event	No	20086680	138989561
4	UC2_5levels_event_nfm	No	693938	3533483
5	UC1_long_pool	No	86616	491856
6	UC1_long_event	Yes	73102	352892
7	UC1_reduced_pool	No	10188	46872
8	UC1_reduced_events	Yes	2544	8374
9	UC2_2levels_event_nfm	Yes	1069	2586

Each state represents a possible configuration of the modelled system. The process starts at the so-called "initial state" and ends at the so-called "end state". The transition represents a possible progression of the system from one configuration to another across time.

MDP model description

In the following section, the MDP model file used in this case study is described, starting with a fragment of the top part of the model, followed by the module section, and finally by the reward structure. Figure 38 shows a fragment of the top part of the MDP model.

```
mdp

label "end_state" = (s0a = 0) & (s0b = 0) & (s0c = 0) & (s0d = 0) & (s0e = 0)
                    &(s0f = 0) & (s0g = 0) & (s0h = 0) & (s0i = 0) & (s0j = 3)
                    &(s1a = 0) & (s1b = 0) & (s1c = 0) & (s1d = 0) & (s1e = 12)
                    &(s2a = 0) & (s2b = 0) & (s2c = 0) & (s2d = 0) & (s2e = 11)
                    &(s3 = 13) & (s4 = 14) & (s5a = 0) & (s5b = 0) & (s5c = 0) & (s5d = 5);
```

List of all variables of the MDP model and the corresponding final state

Figure 38: Fragment of the top part of the MDP model

At the very top, the model file specifies what type of probabilistic model this file concerns. In this example, “*mdp*” stands for the Markov Decision Process (MDP) model. The following section displays the “*label "end_state" =*”, which describes the complete list of variables in this model reaching their final state. The label specifies the property requirement that the whole process must meet to reach its final state, i.e., the whole process reaches the final state only if all variables in the model reach their respective final states.

Module section

The MDP model consists of a set of modules representing different processes of the system being modelled. The state of each module is represented by a set of variables with finite range. The fragment of the model in Figure 39 shows the module section that represents the processes of the model. The module section begins with the command “*module*” accompanied by the name of the module, which is “*Mie*”, and ends with the command “*endmodule*”. In between, the states and transitions of all the processes involved in the module are depicted.

<pre> module M1e s1e: [0..12] init 0; [j1d1] s1e = 0 -> 1: (s1e' = 1); [] s1e = 1 -> 1:(s1e' = 2); [] s1e = 2 -> 1:(s1e' = 3); [] s1e = 3 -> 0.5:(s1e' = 4) + 0.5:(s1e' = 5); [] s1e = 4 -> 1:(s1e' = 5); [] s1e = 5 -> 1:(s1e' = 6); [] s1e = 6 -> 1:(s1e' = 7); [] s1e = 7 -> 1:(s1e' = 8); [] s1e = 8 -> 1:(s1e' = 9); [] s1e = 9 -> 1:(s1e' = 10); [] s1e = 10 -> 0.5:(s1e' = 7) + 0.5: (s1e' = 11); [] s1e = 11 -> 1: (s1e' = 12); endmodule </pre>	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> Model, states and transitions </div>
---	--

Figure 39: Module section representing the processes of the model

Consider the second line from the module section in Figure 39:

```
s1e: [0..12] init 0;
```

As explained in Section 2.2.3, this line indicates that the state of the variable “s1e” is an integer value between 0 and 12. “init” represents the initial value of the state, which is 0 in this case. The behaviour of a module, i.e., the possible changes in its state, is specified by a set of protected commands. Consider the following example taken from the module section:

```
[] s1e = 1 → 1 : (s1e' = 2);
```

The command, for example, describes the changes that occur when a process moves from one particular state to the next. In the example, the variable “s1e” moves from state 1 to state 2. The rate 1 represents the probability of 100% that the transition occurs.

```
[] s1e = 3 -> 0.5:(s1e' = 4) + 0.5:(s1e' = 5);
```

On the other hand, if the rate is less than 1, e.g. 0.5, as can be seen in the example above taken from the module section, the variable “s1e” has a deterministic choice between transitioning from state 3 to either state 4 or state 5, with each choice having a 0.5 (50%) probability that the transition occurs.

Reward structure

This section discusses the addition of rewards to the model, which are used to specify additional quantitative measures of interest. Rewards are simply labels of states and transitions with real values, referred to in PRISM as state rewards and transition rewards. PRISM also allows multiple named reward structures, each defining a label of state and/or transition rewards.

```
rewards
s0a = 2: 17/6;
s0b = 1: 17/6/3;
s0c = 1: 17/6/3;
s0d = 1: 17/6/3;
endrewards
```

List of rewards

Figure 40: Reward structure of the model

The fragment of reward structure of the model in Figure 40 illustrates the specification of rewards in the PRISM modelling language. Each reward structure starts with the command “rewards” and ends with the command “endrewards”. State rewards typically represent the rate at which rewards are accumulated.

soa = 2: 17/6;

For example, the reward specification above associates a state reward of 17/6 with the variable “soa” reaching state 2. From all the accumulated rewards, the total process time or the time the system is in operation can be derived.

Model Visualization

Another fragment of the model, shown in Figure 41, depicts the module section that represents the processes of the module “Moe”. This section describes and visualizes all of the process states and transitions involved in the module.

```

module M0e
s0e: [0..6] init 0;

[j1d0] s0e = 0 -> 1: (s0e' = 1);
[] s0e = 1 -> 1:(s0e' = 2);
[] s0e = 2 -> 1:(s0e' = 3);
[] s0e = 3 -> 1:(s0e' = 4);
[] s0e = 4 -> 0.5:(s0e' = 5) + 0.5:(s0e' = 6);
[d1d0] s0e = 6 -> 1:(s0e' = 0);
[f2d0] s0e = 5 -> 1:(s0e' = 0);
endmodule

```

Figure 41: Module section Moe representing the processes

The process states and transitions of the module can be visualized in form of a figure, as can be seen in Figure 42. The figure shows an example of module “Moe” representing processes with seven states ($s_0, s_1, s_2, s_3, s_4, s_5, s_6$), where both initial and final states are at s_0 .

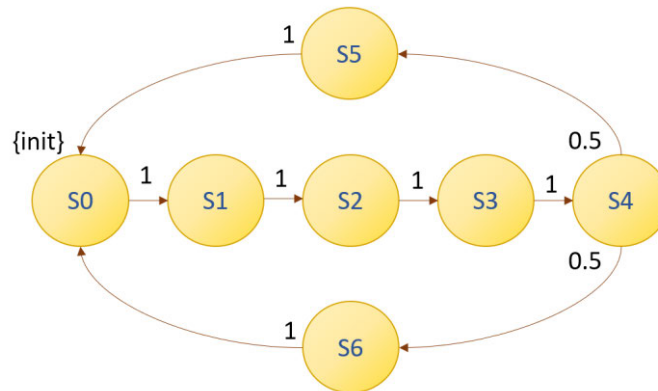


Figure 42: Visual representation of module “Moe”

The process initially starts at state s_0 , which is indicated by “init”. From s_0 to s_4 , the process is given only one option. i.e. with a 100 % probability of moving directly to the next state. After step s_4 , a non-deterministic choice is offered between two actions: 1) first action gives s_4 a probabilistic choice of 0.5 (50%) to move to s_5 and finally return back to s_0 , which is also the final state, and 2) second action gives s_4 also a probabilistic choice of 0.5 or (50%) to move to s_6 and eventually return to the final state, s_0 .

The next section describes the property specification process to ensure the compliance of the processes of the MDP models with A-SPICE.

3.3.3 Property Specification

In order to analyse the MDP models described in the previous section with the PRISM tool, the probabilistic temporal logic specification must be carried out to specify the relevant properties. This step specifies the properties that meet the GP requirements of the A-SPICE standard, as described in Section 3.1, namely the 4 properties ($\varphi_1, \varphi_2, \varphi_3, \varphi_4$).

Firstly, in order to verify φ_1 and φ_4 , it is necessary to ensure the consistency and absence of deadlocks in the processes (GP 2.1.7), the ability of the process performance to meet the identified objectives (GP 2.1.2), and that the process does not deviate from its initial setting based on the A-SPICE guidelines (GP 2.1.3). All of these requirements can be verified by proving that the final state of the process is always reached with a probability of 100%. If the probability is lower than 100%, it implies that there is a bottleneck, deadlocks or delays, which violates the GP of A-SPICE.

Probability-based properties

The probability that the system is in a particular state of interest, can be expressed using the P operator. To determine the probability that the whole process reaches the final state, the following property temporal logic is specified in the PRISM tool, as explained in Section 2.2.3:

- $P_{\min=?} [F \text{"end_state"}]$ – *“the minimum probability that the whole process reaches the final state”*

Furthermore, there are additional functions that allow users to specify which module or variable reaches the final state or which module or variable reaches a specific state. This is helpful to get an overall view of the state of the processes, and in case of a delay or deadlock, the exact variable or module where the deadlock occurs can be identified. The specifications of the temporal logic are as follows:

- $P_{\min=?} [F x = \text{„end_state“}]$ – *“the minimum probability that the variable x reaches the final state”*
- $P_{\min=?} [F x = y]$ – *“the minimum probability that the variable x reaches the state y ”*

Next, in order to verify property 3, φ_3 and obtain information about the process duration, particularly the expected days of the whole process reaching the final state in accumulated working days (wd), the temporal logic of the reward-based properties can be specified in the PRISM tool.

Reward-based properties

The R operator is used to denote the expected value of a random variable defined in terms of a reward structure. Reward can also be referred to as days or cost in this work. The following property specification describes the reward of the process:

- $R_{min=?} [F \text{"end_state"}]$ – “the expected minimum rewards (days) that the whole process reaches the final state”

Furthermore, similar to the probability-based property, there are additional features that allow the specification of the expected minimum rewards (days) for a given module or variable to reach the final state, or the expected minimum (days) for a given module or variable to reach a given state. The specifications of the PCTL are as follows:

- $R_{min=?} [F x = \text{"end_state"}]$ – “the expected minimum rewards (days) that the variable x reaches the final state”
- $R_{min=?} [F x = y]$ – “the expected minimum rewards (days) that the variable x reaches the state y ”

To verify property 2, φ_2 , which provides information about the minimum days (d) to complete the whole process, a simulation of the minimum reward can be performed using the PRISM tool. Alternatively, the minimum days can be determined based on the minimum steps required by the variables of a given module of an MDP model to reach the final state starting from an initial state. For this purpose, all possible scenarios of process transitions in the MDP model are analysed for each module, and only the variables and states that require the fewest steps to reach the final state are considered. Ultimately, the corresponding reward of the variables and states in consideration are accumulated to obtain the minimum days.

3.3.4 Modelling

Once the MDP models have been examined and the properties to be verified with the PRISM tool have been specified, the development of the digital assistant program with Python can begin. The steps taken to model the complete algorithms are described in this section.

As can be seen in Figure 43, the process of modelling the algorithms for the development of the digital assistant goes through three sequential steps. The process starts with the creation of the intent data, followed by the implementation of the training pipeline to generate training data, and finally launching the program. In the first part, the intent data is generated from the manual input of the conversational intents, in which each intent consist of a tag, patterns, and responses. In the second part, the generated intent data is used as input to create the training data by first being pre-processed using the NLP methods and then trained using the PyTorch Deep Learning neural network models. Finally, the generated training file is used as input for the program to make predictions and generate results.

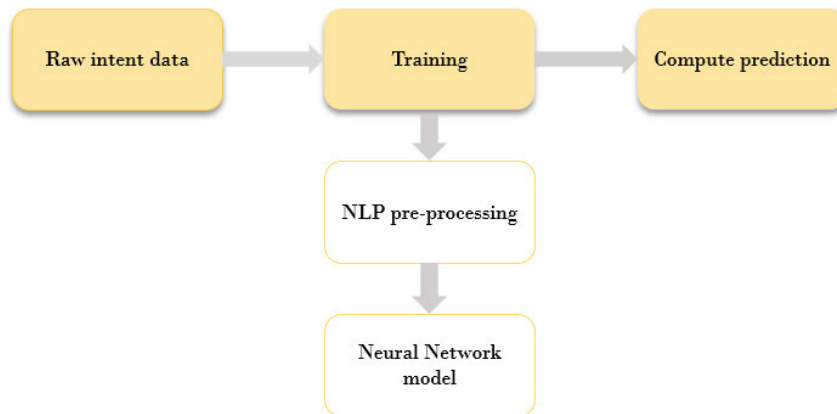


Figure 43: Steps to model the algorithms for the development of the digital assistant

The input and output of each steps are summarised as follows:

1. **Input:** Conversational intents (tag, patterns, and responses), **Output:** Intent file
2. **Input:** Intent file, **Output:** Training file
3. **Input:** Training file, **Output:** Prediction results

i. Model intent data

Input data must first be provided to the model, in order for the Deep Learning model to learn and perform pattern recognition on the data and thus, compute prediction of an outcome. The more data provided, the better the performance of the neural network, as it can be trained for many different types of input data, including different linguistic styles of texts, as well as slangs of words that may vary from user to user. The intent data is constructed in the form of a structure, in which tag, patterns and responses of a conversational intent are manually defined. In order to create the intent data, each conversational intent should contain:

- tag
- patterns
- response

The digital assistant examines and studies the characteristics of the input patterns specified in the intent data and then attempts to predict the correct result to a new input message from a user by performing pattern recognition, i.e. attempting to identify the similarities between the user's message and the specified patterns. If the similarities match to a certain probability, one of the specified responses of the matched patterns is returned to the user as the reply. Each patterns and response pair is specified with a tag, which is simply a unique name for each pair.

Intent parameters

In order to specify the tag, pattern, and responses in the intent data, it is first necessary to define all the functions and parameters that the digital assistant will base its predictions on. Besides performing functions that satisfy the A-SPICE properties φ_1 , φ_2 , φ_3 , and φ_4 , the digital assistant should also be able to compute predictions that return the outcome of other additional model checking parameters, and provide other relevant information concerning the processes. All the defined parameters are listed in Table 4.

Table 4: All parameters for the digital assistant program

1	List of A-SPICE Generic Practices (GP) Level 2
2	Probability of the whole process reaching final state
3	Probability of a user-defined module reaching final state
4	Probability of a user-defined module reaching a user-defined state
5	Minimum days for performing whole process
6	Estimation of expected minimum days for performing whole process
7	Estimation of expected minimum days a user-defined module reaching final state
8	Estimation of expected minimum days a user-defined module reaching a user-defined state
9	Total number of transitions
10	Total number of states
11	Value of initial state
12	Building model time
13	Model checking time
14	List of modules names
15	List of variables names

Now that all the parameters are defined, the intent data, which consists of tags, patterns and responses based on the parameters can be constructed. All parameters with the corresponding tag, patterns and responses are presented in the form of tables in the following section.

- **A-SPICE Generic Practices (GP)**

Table 5 lists the tag, patterns and responses specified for the parameter A-SPICE Generic Practices (GP).

Table 5: Tag, patterns and responses for parameter A-SPICE Generic Practices (GP)

Tag	operation
Patterns	"What is the highest reachable A-SPICE level?", "What is the highest A-SPICE level that we can currently achieve?", "Highest A-SPICE level?"
Responses	"As of right now, Level 2 is the highest reachable A-SPICE Level. Should I list down all the Generic Practice of A-SPICE level 2?"

- **Probability of the whole process reaching final state**

Table 6 shows the tag, patterns and responses specified for the parameter probability of the whole process reaching the final state.

Table 6: Tag, patterns and responses for parameter probability of the whole process reaching the final state

Tag	one_one
Patterns	"What is the probability of the whole process?", "Please verify if GP 2.1.7 is met", "Please verify if GP 2.1.3 is met"
Responses	The probability of the whole process reaching the final state is: { <i>results</i> }

- **Probability of a specific module reaching final state**

Table 7 displayed the tag, patterns and responses specified for the parameter probability of a specific module reaching the final state.

Table 7: Tag, patterns and responses for parameter probability of a specific module reaching the final state

Tag	one_two
Patterns	"Please verify the probability of a specific module reaching final state", "What is the probability of a specific module reaching final state?"
Responses	The probability of { <i>module</i> } reaching the final state of the process is: { <i>results</i> }

- **Probability of a specific module reaching a specific state**

Table 8 lists the tag, patterns and responses specified for the parameter probability of a specific module reaching a specific state.

Table 8: Tag, patterns and responses for parameter probability of a specific module reaching a specific state

Tag	one_three
Patterns	"Please verify the probability of a specific module reaching a specific state", "What is the probability of a specific module reaching a specific state?"
Responses	The probability of { <i>module</i> } reaching the { <i>state</i> } of the process is: { <i>results</i> }

- **Minimum days for performing whole process**

Table 9 displays the tag, patterns and responses specified for the parameter minimum days for performing the whole process.

Table 9: Tag, patterns and responses for parameter minimum days for performing the whole process

Tag	two_one
Patterns	"What is the minimum days for performing whole process?", "Please tell me how long is the duration of the whole process?"
Responses	The minimum days for performing whole process is { <i>results</i> } days.

- **Estimation of expected minimum days for performing whole process**

Table 10 shows the tag, patterns and responses specified for the parameter estimation of expected minimum days for performing the whole process.

Table 10: Tag, patterns and responses for parameter estimation of expected minimum days for performing the whole process

Tag	three_one
Patterns	"What is the expected minimum days of my whole process?", "How long is the expected minimum days of whole process?"
Responses	The estimation of expected minimum days for performing whole process is { <i>results</i> } days.

- **Estimation of expected minimum days a specific module reaching final state**

Table 11 lists the tag, patterns and responses specified for the parameter estimation of expected minimum days a specific module reaching the final state.

Table 11: Tag, patterns and responses for parameter estimation of expected minimum days a specific module reaching the final state

Tag	three_two
Patterns	"What is the estimation of expected minimum days of a specific module reaching final state?", "Please tell me the estimation of expected minimum days of a specific module reaching final state"
Responses	The estimation of expected minimum days for { <i>module</i> } to reach final state is { <i>results</i> } days.

- **Estimation of expected minimum days a specific module reaching a specific state**

Table 12 shows the tag, patterns and responses specified for the parameter estimation of expected minimum days a specific module reaching a specific state.

Table 12: Tag, patterns and responses for parameter expected minimum days a specific module reaching a specific state

Tag	three_three
Patterns	"What is the estimation of expected minimum days of a specific module reaching specific state?", "Please tell me the estimation of expected minimum days of a specific module reaching specific state"
Responses	The estimation of expected minimum days for { <i>module</i> } to reach { <i>state</i> } is { <i>results</i> } days.

- **Total number of transitions**

Table 13 illustrates the tag, patterns and responses specified for the parameter total number of transitions.

Table 13: Tag, patterns and responses for parameter total number of transitions

Tag	transitions
Patterns	"What is the total number of transitions in my process?", "Please tell me the total number of transitions of the processes?"
Responses	The total number of transitions of the processes is { <i>results</i> }.

- **Total number of states**

Table 14 lists the tag, patterns and responses specified for the parameter total number of states.

Table 14: Tag, patterns and responses for parameter total number of states

Tag	states
Patterns	"What is the total number of states in my process?", "Please tell me the total number of states of the processes?"
Responses	The total number of states of the processes is { <i>results</i> }.

- **Value of initial state**

Table 15 shows the tag, patterns and responses specified for the parameter value of initial state.

Table 15: Tag, patterns and responses for parameter value of initial state

Tag	four_three
Patterns	"What is the value of initial state of my process?", "Please tell me the value of initial state of my processes?"
Responses	The value of initial state of the processes is { <i>results</i> }.

- **Building model time**

Table 16 displays the tag, patterns and responses specified for the parameter building or construction model time.

Table 16: Tag, patterns and responses for parameter building model time

Tag	four_four
Patterns	"What is the construction model time of the process?", "Please tell me the construction time of the model?"
Responses	Time for model construction is { <i>results</i> } seconds.

- **Model checking time**

Table 17 lists the tag, patterns and responses specified for the parameter model checking time.

Table 17: Tag, patterns and responses for parameter model checking time

Tag	five_one
Patterns	"What is the checking model time of the process?", "Please tell me the checking model time."
Responses	Time for model checking is { <i>results</i> } seconds.

- **List of modules names**

Table 18 demonstrates the tag, patterns and responses specified for the parameter list of modules names.

Table 18: Tag, patterns and responses for parameter list of modules names

Tag	five_two
Patterns	"Please list down all modules name of the model", "What is the modules name of the model?"
Responses	The names of the modules are [<i>results</i>].

- **List of variables names**

Table 19 lists the tag, patterns and responses specified for the parameter list of variables names.

Table 19: Tag, patterns and responses for parameter list of variables names

Tag	five_three
Patterns	"Please list down all variables name of the model", "What is the variables name of the model?"
Responses	The names of the variables are [<i>results</i>].

The pseudo-code in Figure 44 illustrates the programming steps for developing the intent data from the manually entered tags, patterns, and responses.

Algorithm 1: Create intent data

Input: Tags, Patterns and Responses data

Output: Intents (Intent data)

1. **SET** Intents **as** Intent data
2. **FOR** each element of Intents **do**
3. **READ** tags
4. **READ** patterns
5. **READ** responses
6. **ENDFOR**
7. **RETURN** Intents

Figure 44: Pseudo-code of creating an intent data

ii. Model Training

Once the intent data have been generated, it goes through pre-processing steps using various NLP methods. Then, the data is trained with Deep Learning models which generates the training data. This section describes the steps of pre-processing with NLP methods and the training process with the neural network.

- **Pre-processing with NLP methods**

NLP methods, which include tokenizing, lower casing and stemming, removing of punctuation characters and Bag-of-Words (BoW), as explained in Section 2.3.2, are applied to perform pre-processing of the intent data. The entire pre-processing pipeline is shown in Figure 45.

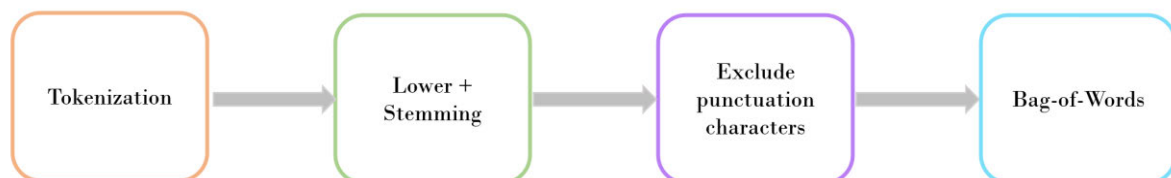


Figure 45: NLP pre-processing pipeline

The intent data is used as input in this method. In the first step, all strings contained in the data are separated according to the sentence boundaries and stored in a list. Then, the individual words of each sentence are extracted and tokenized and then stored sentence by sentence in a separate list. Subsequently, the individual tokenized words are lowercased and stemmed to obtain the root form of the words. Punctuation marks are removed from the word list in the following step. Finally, each sentence is converted into a BoW that counts the total occurrences of the words in the list. This converts the pattern strings into numerical data that the Deep Learning model can understand.

In this work, the NLTK and NumPy libraries, as described in sections 2.3.2 and 2.4.4, respectively, are applied to assist the implementation of NLP methods in Python programs.

Natural Language Toolkit

Natural Language Toolkit (NLTK) offers various pre-processing methods applied in this work, such as stemming and tokenization. For the implementation of this work, the “*punkt*” library is used from NLTK.

NumPy

Additionally, the NumPy library is also used to create multi-dimensional arrays of the data since Python does not support this basic data structure. Besides, it provides implementations of linear algebra and random number generation and also used together with PyTorch to implement the tensor data structure.

The code to import the NLTK module, specifically the “*punkt*” and “*PorterStemmer*” libraries and NumPy module, is shown in Figure 46.

```
import numpy as np
import nltk
nltk.download('punkt')
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()
```

Figure 46: Importing the libraries of NLTK and NumPy in Python

Tokenization

Tokenization is the process of breaking down a sentence or paragraph into smaller units called tokens. In order to perform word tokenization, *word_tokenize()* class from NLTK module is applied in Python, as can be seen in the code in Figure 47.

```
def tokenize(sentence):
    return nltk.word_tokenize(sentence)
```

Figure 47: Code sample of tokenization in Python

Stemming

Stemming is a normalization technique where list of tokenized words are converted into shorten root words to remove redundancy and can be performed by applying the `stemmer.stem()` from “*PorterStemmer*” class of NLTK module in Python, as shown in Figure 48.

```
def stem(word):  
    return stemmer.stem(word.lower())
```

Figure 48: Code sample of stemming in Python

Bag-of-Words

To feed the text input into the Deep Learning model, the text must first be converted into number vectors using BoW. This is done by calculating the total number of occurrence of the words in the input list following the previous NLP methods. The NumPy class function `numpy.zeros()` is used to create an array filled with zeros. The complete code for applying the BoW method is shown in Figure 49.

```
def bag_of_words(tokenized_sentence, words):  
    # stem each word  
    sentence_words = [stem(word) for word in tokenized_sentence]  
    # initialize bag with 0 for each word  
    bag = np.zeros(len(words), dtype=np.float32)  
    for idx, w in enumerate(words):  
        if w in sentence_words:  
            bag[idx] = 1  
    return bag
```

Figure 49: Code sample of Bag-of-Words (BoW) in Python

The pseudo-code in Figure 50 illustrates the programming steps of the complete pre-processing sequence using the NLP methods.

Algorithm 2: Pre-processing with NLP

Input: Intents (Intent data)

Output: Pre-processed data

1. **FOR** each pattern in Intents (Patterns) **do**
2. **CALL** NLTK module **RETURNING** Tokenizing
3. **Tokenize** each word in sentence
4. **ADD** tokenized patterns to all words list
5. **ADD** Intents (Tag) and Intents (Patterns) to pair list
6. **ENDFOR**

7. **FOR** each words in all words **do**
8. **IF** words is not ignore words **THEN**
9. **CALL** NLTK module **RETURNING** Stemming
10. **Stem** and **lower** each word
11. **ENDIF**
12. **ADD** stem words to all words
13. **ENDFOR**

14. **FOR** each patterns sentence and tag in pair list **do**
15. **CALL** NLTK module **RETURNING** Stemming
16. **Stem** each word in tokenized sentence
17. **INIT** BoW with 0 for each word
18. **FOR** each word **do**
19. **IF** word in sentenced words **THEN**
20. **SET** element of BoW to 1
21. **ENDIF**
22. **ENDFOR**
23. **RETURN** BoW

Figure 50: Pseudo-code of pre-processing with NLP

- **Implementation of Neural Network**

After the pre-processing steps have been performed, the data can now be trained with a Deep Learning model. The model is implemented using a Feed Forward Neural Network (FNN) model with 2 hidden layers. An overview of the steps performed in this phase can be seen in Figure 51. First, the pre-processed data is fed into the input layer, and then passes through the 2 hidden layers of the model to be trained. Finally, the output layer describes the last layer where the predictions are computed.

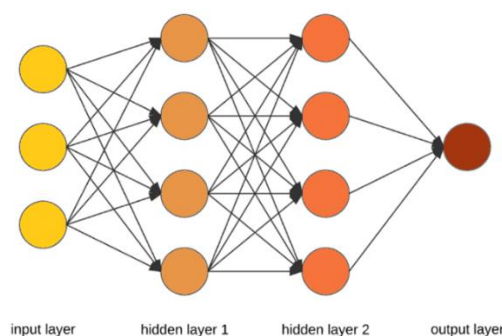


Figure 51: Overview of training process with Deep Learning model

The goal of supervised learning is to build a model that makes predictions on the outcomes of new data based on a given sample of data. The following procedures are used for supervised training of the Deep Learning model, as explained in section 2.4.3. First, a model is defined, followed by the activation function, the loss function, and the optimizer. Finally, all the hyperparameter values are specified.

Choosing a model

It is first important to define the model in order to start the supervised training process. As mentioned, the model chosen for this work is a FNN with 2 hidden layers. The input size is determined by the number of word elements in the pre-processed data.

Choosing activation function

Activation function: ReLU

The Rectified Linear Unit (ReLU) is arguably the most important activation function in use today, as it offers faster learning time and lower probability of gradient vanishing due to sparsity compared to other activation functions, as described in Section 2.4.3. For these reasons, the ReLU is chosen as the activation function in this work.

The code in Figure 52 presents a Deep Learning model implementation in PyTorch that takes the input and hidden layer size, does the linear transformation, applies ReLU as an activation function, and produces a single output.

```
import torch
import torch.nn as nn
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.l2 = nn.Linear(hidden_size, hidden_size)
        self.l3 = nn.Linear(hidden_size, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)
        return out
```

Figure 52: Code for implementation of Deep Learning model on PyTorch

PyTorch conveniently offers a `Linear()` class in the `torch.nn` module that applies a linear transformation to the incoming data, and does the bookkeeping needed for the weights and biases. PyTorch also offers a `ReLU()` class that applies the ReLU function element-wise to the data (Rao & McMahan, 2019). The class `super()` is also applied as it is useful for accessing inherited methods that have been overridden in a class.

After the model architecture has been constructed with the help of an activation function, there are two other vital components to consider in supervised training: a loss function and an optimizer.

Choosing a loss function

Loss function attempts to match the algorithm's performance to the expected output. The loss function takes a true value (y), also called the “error”, and a prediction (\hat{y}) as input and produces a real value that indicates the performance of the prediction.

Loss function: Cross-Entropy Loss

Cross-entropy loss is chosen in this work as the loss function because it is mainly used for classification problems and for situations, in which the model's output is a probability, as it minimizes the distance between two probability distributions of the predicted and actual, as described in Section 2.4.3. The `nn.CrossEntropyLoss()` class from PyTorch is applied for the Cross-Entropy loss function. The implementation of the class in the code is displayed in Figure 53.

```
import torch
import torch.nn as nn

# Loss function
criterion = nn.CrossEntropyLoss()
```

Figure 53: Code for implementation of Cross-Entropy Loss on PyTorch

Choosing an optimizer

The final selection point of supervised training is the optimizer. The optimizer is used to reduce the error rates when training the neural networks by updating the weights of the model based on the error signal. The optimizer is associated with the learning rate hyper-parameter, which controls the extent to which the error signal affects the updating of the weights.

Optimizer: Adam optimizer

The Adam optimizer is chosen as the optimizer in this work because this type of optimizer currently have better performance, faster computation time, and require fewer parameters to tune compared to the other optimization algorithms, as mentioned in Section 2.4.3. PyTorch offers the class `torch.optim.Adam()` for implementing the Adam optimizer algorithms. The code is displayed in Figure 54.

```
import torch
import torch.nn as nn
from model import NeuralNet

# Optimzer
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Figure 54: Code for implementation of Adam Optimizer on PyTorch

Early stopping

Early stopping is another technique applied in this work to determine when to stop training in order to prevent the overfitting of the neural network. The training phase of the network is stopped after a certain number of epochs, which is when the performance continues to not improve. This point can also be referred to as convergence. To visualize the convergence point, the loss curve is used to analyse the loss of a neural network during training. The loss refers to the loss value over the training data after each epoch and gives a snapshot of the training process and the direction in which the network is learning. This is the value that the optimization process aims to minimize with training, so the lower the better (Sharma, 2019).

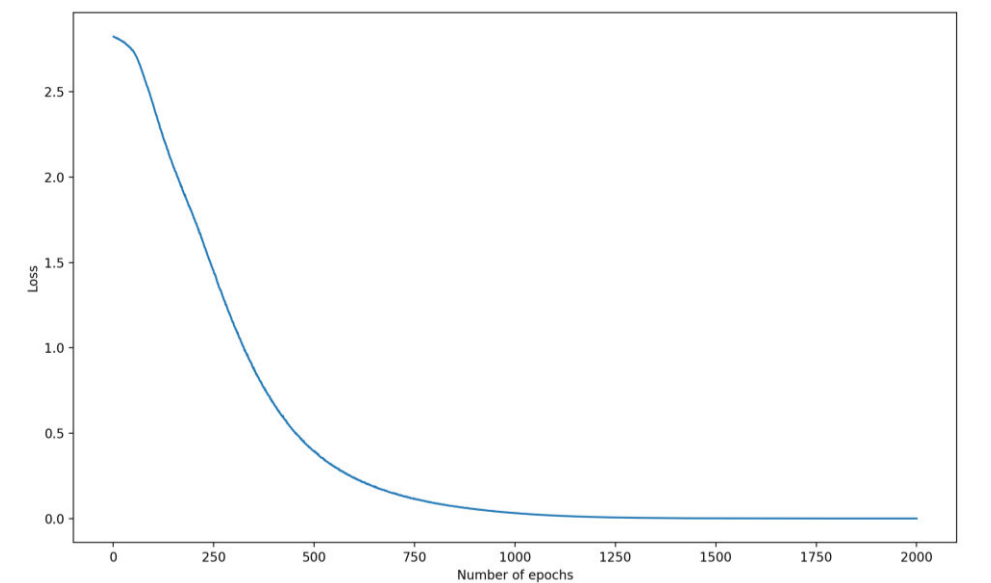


Figure 55: Loss curve

Figure 55 shows a plot generated with the help of the Matplotlib library in Python, demonstrating the influence of an increasing number of epochs on the loss value of the training model. It can be seen that the performance does not improve, i.e. the loss value stops decreasing after approximately 1000 epochs, indicating that the training phase of the network should be stopped at this point, as suggested by the early stopping method to avoid overfitting of the network. The number of epochs set at this point can also be referred to as “convergence”. The value of 1000 epochs is therefore used as the hyper-parameter value for the optimal number of epoch in the next section.

Hyper-parameters

A hyper-parameter is any parameter whose value is used to control the learning process, which can have a large impact on the convergence and performance of a model. The selection of the hyper-parameter includes the optimal choice of the learning rate(s) for the optimizer, the number of epochs, the size of the batch and the hidden layers, and the output size. The values of the hyper-parameters specified in this work are listed in Table 20.

Table 20: Hyper-parameter values

Hyper-parameters	Values
Number of epochs	1000
Batch size	8
Learning rate	0.001
Hidden size	2

The optimal number of epochs were specified as 1000, based on the results of the early stopping method. The batch sizes were specified as 8, based on trial-and-error. For the selection of learning rates, the following considerations are taken into account. Large learning rates result in larger changes to the parameters and can affect convergence. Learning rates that are too small, on the other hand, can lead to very little progress in training (Rao & McMahan, 2019). For Adam optimizer, the default learning rate is 0.001, which is also the recommended value. Therefore, this value is used as the learning rate in this work. The input size corresponds to the number of elements in the pre-processed intent file. The hidden size is the number of hidden layers of the model, which is 2. Lastly, the output size is the number of tags present in the intent data.

Once all steps are performed, the training phase of the model is started. The pseudo-code in Figure 56 illustrates the overview of the programming steps necessary to implement the training process of the Deep Learning neural network.

Algorithm 3: Implement the Training Pipeline

Input: Pre-processed data

Output: Training data

1. **SET** Neural Network (Inputs: Input size, hidden size, ReLU as activation function)
2. **SET** Hyper-parameters (Inputs: num_epochs, batch_size, learning_rate, input_size, hidden_size, output_size)
- 3.
4. **FOR** each epoch in total number of epochs **do**
5. **FOR** every words and labels in Train Loader **do**
6. **CALL** Forward Pass **RETURNING** model words
7. **CALL** Cross-Entropy Loss (Inputs: model words and labels)
8. **CALL** Backward Pass
9. **CALL** Adam Optimizer (Inputs: model parameters and learning rate)
10. **ENDFOR**
11. **ENDFOR**
12. **RETURN** training data to "data.pth"

Figure 56: Pseudo-code of implementing the training pipeline

Once the training is complete, the training data is stored in the “*data.pth*” file. This file is subsequently used by the model to calculate predictions and return outputs in the step described in the following section.

iii. Launching

Once the training has been completed and training data (*data.pth*) has been created, the algorithms to launch the program is now developed, in order for the program to make predictions and return results. The training data can be imported in the program by the code shown in Figure 57.

```
FILE = "data.pth"  
data = torch.load(FILE)
```

Figure 57: Importing training data

- **MDP File Upload**

For the program to be operational, it must first be ensured that an MDP model with the correct file extension is uploaded to the program. Either the “.prism” or “.txt” file extension is permitted for the model. If no MDP model with the correct file extension is uploaded, an error message is displayed until the model with the correct format is uploaded. The pseudo-code in Figure 58 illustrates the programming steps to implement this functionality.

1. **WHILE** MDP model file does not exist:
2. **DISPLAY** error message
3. **IF** MDP model file exists **THEN**
4. **ENDWHILE**
5. **ENDIF**

Figure 58: Pseudo-code of uploading MDP file

- **Computing predictions**

The next step is to develop the algorithms to compute predictions based on the training data generated in the training phase, as described in the previous section. In this phase, the digital assistant program's algorithm is constructed to first receive the user's message, predict its intent, and then send a query based on the predicted intent to the PRISM tool on the backend. Here, the results are computed, then received back by the program, and finally delivered to the user as a response. The graphical representation of the process of computing predictions and data exchange between the users, the digital assistant program and the PRISM tool is shown in Figure 59.



Figure 59: Graphical representation of the process computing predictions and data exchange

Data exchange on the conversational front-end

The process of computing predictions and data exchange on the conversational front-end between the user and the program involves the following steps. The program is provided with an input, which can be either a string of characters or numbers representing the message sent by the user. The Deep Learning model of the digital assistant then tries to predict the correct desired response to this input message from the user based on the generated training data by performing pattern recognition on the intent, i.e., by trying to identify the similarities between the user's message and the patterns specified in the intent data. If the similarities match with a probability of 75% or higher, the response of the corresponding matched patterns in the intent data is returned to the user as a reply. On the other hand, if the message from user does not match any of the specified patterns in the intent data with a probability of at least 75%, the program will tell the user to rephrase the input message by returning "I don't understand, please rephrase the question".

Data exchange at the backend

The data exchange process between the digital assistant program and the PRISM tool at the backend is only executed if the input message matches the patterns in the intent file with a probability of higher than 75%. If this is the case, the corresponding PRISM class of the matching tag is called via the Java API on the backend. The results are computed by the PRISM tool and then returned by the class to the digital assistant program, and subsequently appended to the response of the matching tag. Table 21 lists all parameters and the corresponding PRISM class functions.

Table 21: All parameters with the associated PRISM class function

Parameter	PRISM class
Probability of the whole process reaching final state	<i>prism.modelCheck</i> (<i>propertiesFile, propertiesFile.</i> <i>getPropertyObject()</i>)
Probability of a user-defined module reaching final state	<i>prism.modelCheck</i> (<i>propertiesFile, propertiesFile.</i> <i>getPropertyObject()</i>)
Probability of a user-defined module reaching a user-defined state	<i>prism.modelCheck</i> (<i>propertiesFile, propertiesFile.</i> <i>getPropertyObject()</i>)
Minimum days for performing whole process	<i>sim.getCumulativeRewardUpToPathStep,</i> <i>((int)sim.getPathSize(),0)</i>
Estimation of expected minimum days for performing whole process	<i>prism.modelCheck</i> (<i>propertiesFile, propertiesFile.</i> <i>getPropertyObject()</i>)
Estimation of expected minimum days a user-defined module reaching final state	<i>prism.modelCheck</i> (<i>propertiesFile, propertiesFile.</i> <i>getPropertyObject()</i>)
Estimation of expected minimum days a user-defined module reaching a user-defined state	<i>prism.modelCheck</i> (<i>propertiesFile, propertiesFile.</i> <i>getPropertyObject()</i>)
Total number of transitions	<i>prism.buildModel</i> <i>(modulesFile).getNumTransitions()</i>
Total number of states	<i>prism.buildModel</i> <i>(modulesFile).getNumStates()</i>
Value of initial state	<i>prism.buildModel</i> <i>(modulesFile).getNumStartStates()</i>
Building model time	<i>prism.buildModel.</i> <i>System.currentTimeMillis()</i>
Model checking time	<i>prism.modelCheck.</i> <i>System.currentTimeMillis()</i>
List of modules names	<i>modulesFile.getModuleName()</i>
List of variables names	<i>modulesFile.getVarName()</i>

Figure 60 depicts the data exchange interaction between the program parameter and the corresponding PRISM class in the backend.

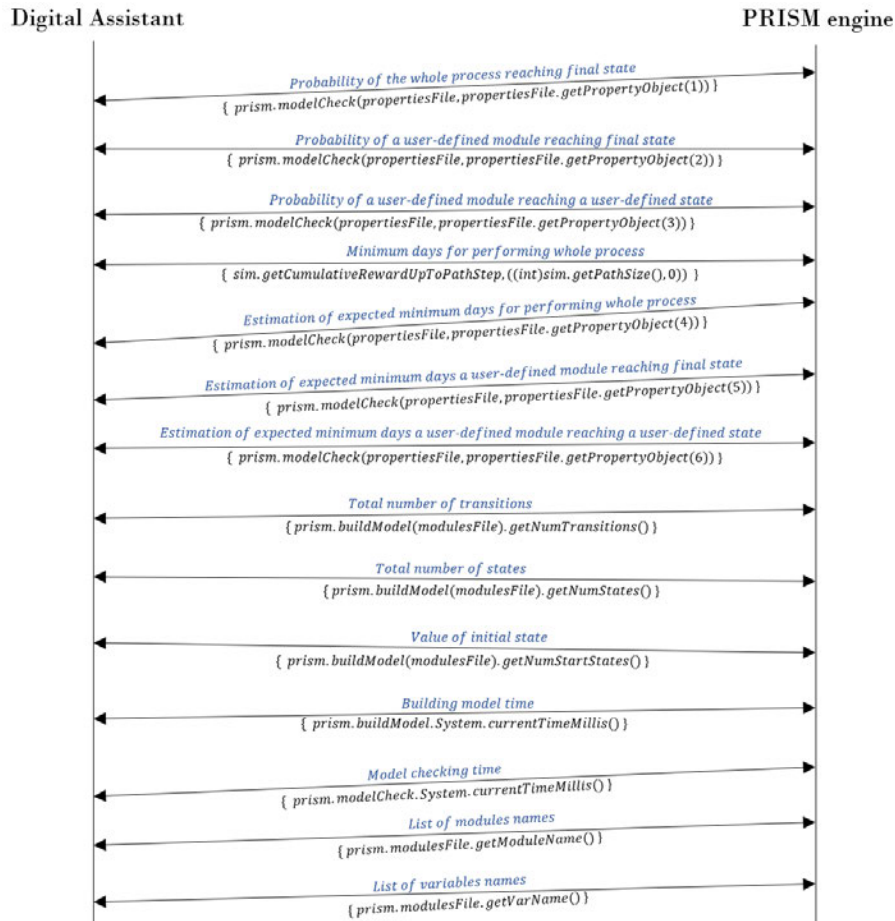


Figure 60: Data exchange between the digital assistant program with the back-end PRISM tool

The pseudo-code in Figure 61, Figure 62 and Figure 63 illustrate the programming steps of the entire implementation of the prediction algorithms.

Algorithm 4: Computing Results

Input: Training data

Output: Prediction results

1. **GET** message from user
2. **COMPUTE** prediction of accuracy via torch models of message matching tags
3. **IF** message has higher probability than 0.75 of matching Intents **THEN**
4. **FOR** each Intent (tag):
5. **GET** MDP model, PRCTL Properties

6. **IF** the tag intent = "one_one" **THEN**
7. **READ** all modules and corresponding final state
8. **CALL** PRISM probability
9. **RETURN** Probability of whole process reaching final state
10. **ENDIF**

Figure 61: The pseudo-code of implementation of the prediction algorithms (1)

```

11.   IF the tag intent = “one_two” THEN
12.     GET module number from user
13.     READ final state of the corresponding module
14.     CALL PRISM probability with user-defined module
15.     RETURN Probability of a specific module reaching the final state
16.   ENDIF

17.   IF the tag intent = “one_three” THEN
18.     GET module number from user
19.     GET state number from user
20.     CALL PRISM probability with user-defined module, user-defined state
21.     RETURN Probability of a specific module reaching the specific state
22.   ENDIF

23.   IF the tag intent = “two_one” THEN
24.     READ all modules and corresponding final state
25.     CALL PRISM minimum days
26.     RETURN Minimum days for performing whole process
27.   ENDIF

28.   IF the tag intent = “three_one” THEN
29.     READ all modules and corresponding final state
30.     CALL PRISM estimation of expected minimum days
31.     RETURN estimation of expected minimum days for performing whole process
32.   ENDIF

33.   IF the tag intent = “three_two” THEN
34.     GET module number from user
35.     READ final state of the corresponding module
36.     CALL PRISM estimation of expected minimum days with user-defined module
37.     RETURN estimation of expected minimum days of specific module reaching the final state
38.   ENDIF

39.   IF the tag intent = “three_three” THEN
40.     GET module number from user
41.     GET state number from user
42.     CALL PRISM estimation of expected minimum days with specific module and specific state
43.     RETURN estimation of expected minimum days of specific module reaching specific state
44.   ENDIF

45.   IF the tag intent = “four_one” THEN
46.     READ all modules and corresponding final state
47.     CALL PRISM total number of transitions
48.     RETURN total number of transitions of model
49.   ENDIF

50.   IF the tag intent = “four_two” THEN
51.     READ all modules and corresponding final state
52.     CALL PRISM total number of states
53.     RETURN total number of states of model
54.   ENDIF

```

Figure 62: The pseudo-code of implementation of the prediction algorithms (2)

```

55.   IF the tag intent = "four_three" THEN
56.     READ all modules and corresponding final state
57.     CALL PRISM model initial state
58.     RETURN initial state of model
59.   ENDF

60.   IF the tag intent = "four_four" THEN
61.     READ all modules and corresponding final state
62.     CALL PRISM model building time
63.     RETURN model building time
64.   ENDF

65.   IF the tag intent = "five_one" THEN
66.     READ all modules and corresponding final state
67.     CALL PRISM model checking time
68.     RETURN model checking time
69.   ENDF

70.   IF the tag intent = "five_two" THEN
71.     READ all modules and corresponding final state
72.     CALL PRISM all modules names
73.     RETURN all modules names of model
74.   ENDF

75.   IF the tag intent = "five_three" THEN
76.     READ all modules and corresponding final state
77.     CALL PRISM all variables names
78.     RETURN all variables names of model
79.   ENDF

80.   IF the tag intent = "operation" THEN
81.     CALL Generic Practices
82.     RETURN Generic Practices A-SPICE
83.   ENDF
84.
85. ENDFOR
86. ENDF

```

Figure 63: The pseudo-code of implementation of the prediction algorithms (3)

iv. Model front-end visualization

Now that the program algorithms have been constructed, the development of the front-end visualization design of the digital assistant will begin. The design includes a display panel to visualize the exchange of messages between the user and the digital assistant, a text input placeholder for the user to enter the message, a button for the user to send the message, and a file upload function that allows the user to browse the MDP model and upload it to the program.

- **User-interface**

The front-end visualization functionalities (display panel, placeholder for text input, send and upload file function) are developed using a combination of HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS).

- **HTML** is used to structure and define the contents of the web page.
- **CSS** is a style sheet language used to describe the presentation and define the layout of the web page written in HTML and the JavaScript language.
- **JavaScript** is used to program the behaviour and functionality of the web page.

A template of the original structure of the front-end visualization design is provided on the GitHub page of (Nacario, 2020).

- **Upload file function**

The file upload function consists of a button to browse and select the MDP model, permitting only the *.txt* or *.prism* file extension, and another button to upload the model. A message notifying the user that the model has been successfully uploaded to the program is also provided. To develop the file upload function, the codes are written in both Python and JavaScript. The implementation of the code for both languages are described in the following section.

Python

Once the user has done selecting and submitting an MDP file, the route `"/upload_static_file"` receives a POST request and the uploaded file from JavaScript. The underlying function `f.save` then performs a save operation. The code in Figure 64 creates a Python server for the function.

```

1 @app.post("/upload_static_file")
2 def upload_static_file():
3     print("Received request in static files")
4     f = request.files['static_file']
5     f.save(f.filename)
6     resp = {"success": True, "response": "file saved!"}
7     return jsonify(resp)

```

Figure 64: Code of implementing upload file function in Python

JavaScript

On a button click from user, the function `uploadFile` is called from the JavaScript file. The complete form is passed in the function parameters, as can be seen in Figure 65. At the beginning of the function, a new object of `FormData` is created and subsequently, the `form` received from the HTML file is passed. After that, a new `XMLHttpRequest` object i.e. `oReq` is created in order to send the form data to the Python server. The statement `oReq.open("POST", "upload_static_file", true)` mentions the request type and the route for the server.

```

1  function uploadFile(form)
2  {
3    const formData = new FormData(form);
4    var oOutput = document.getElementById("static_file_respons
e")
5    var oReq = new XMLHttpRequest();
6      oReq.open("POST", "upload_static_file", true);
7    oReq.onload = function(oEvent) {
8      oOutput.innerHTML = "File uploaded succesfully!";
9      console.log(oReq.response)
10   oOutput.innerHTML = "Sending file!";
11   console.log("Sending file!")
12   oReq.send(formData);
13 }

```

Figure 65: Code of implementing upload file function on JavaScript

3.3.5 Integration of Programming Languages

Finally, this section describes the process of connecting all different programming languages used in the development of the program. APIs and other relevant libraries are applied to support integration between different applications by allowing a program to share and exchange data and functionalities with other external third parties.

In this work, 4 different languages are used to develop the program. The PRISM language is used by the PRISM tool to perform model checking functions, Java language is used to write the PRISM API, which allows the execution of model checking tool functions on Java program, Python language is used to develop the digital assistant program and to construct Deep Learning models using the PyTorch framework, and lastly, JavaScript, including HTML and CSS are used to develop the front-end visualization design. PRISM and Java are linked together via an API to provide a programmatic connection to the PRISM functions on Java. Java and Python are integrated to allow exchange of the requests and results of model checking functions. PRISM and Python cannot be integrated directly because the API to programmatically connect to the PRISM tool is only available in Java language. Therefore, Java is used as an intermediate program to link PRISM and Python. Python and JavaScript are connected together to allow the front-end visualization of the program. The whole interchange between the different programming languages can be seen in Figure 66.

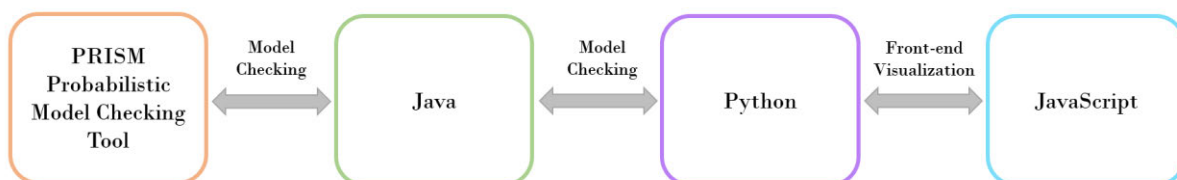


Figure 66: Interaction between different programming languages

I. Connecting PRISM and JAVA via JAVA API

PRISM and Java are integrated to allow the execution of model checking tool functions on Java program. The programmatic connection between the PRISM tool and Java is established via an API, as illustrated in Figure 67.



Figure 67: Interaction between PRISM and Java program via API

The API is written in Java language and is available on the PRISM's GitHub page of (Parker, 2022). To establish the connection, a copy of the PRISM source code is first downloaded and constructed by cloning the “prism” folder and running the “make” command in the interpreter's terminal. Then, the “prism-api” folder is cloned up one level from the directory where the PRISM source distribution is located, and finally, the “make” command is executed again. More detailed instructions can be found on the GitHub page.

II. Connecting Java and Python via Py4J library

Java and Python are integrated to allow the exchange of requests and results of model checking functions. Py4J is used as a library to enable the connection between Java and Python programming languages. Specifically, it allows the Python program to dynamically access Java objects in a Java Virtual Machine (JVM), which includes the PRISM functions written as Java objects. The methods are called as if the Java objects were in the Python program. Py4J also allows Java programs to call back Python objects. Before running Py4J in Python, it is necessary to ensure that the Py4J library is installed in the Python libraries by running the “pip install py4j” command on the interpreter's terminal. The architecture of the data exchange between Python and Java via the Py4J library can be seen in Figure 68.

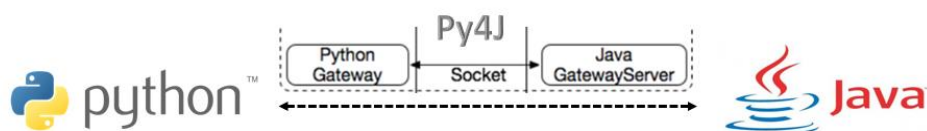


Figure 68: Interaction between Python and Java program via Py4J

The following section describes the code implementation of the Py4J library in both Python and Java (Py4J, 2009).

Python

When a *GatewayServer* instance is started, Python programs can connect to the JVM by calling, `gateway = JavaGateway()`. The `entryPoint` passed to a *GatewayServer* can be accessed with the `entry_point` member: `gateway.entry_point`. The implementation code for Py4J module in Python is shown in Figure 69.

```
1 from py4j.java_gateway import JavaGateway
2 gateway = JavaGateway()
3 prism_java = gateway.entry_point
```

Figure 69: Implementation code of Py4J Python

JAVA

It is to note that the Java program must be started before executing the Python code in Figure 69. In other words, the Py4J does not start a JVM. A Java *GatewayServer* is the main interaction point between a Python Virtual Machine (VM) and a JVM. Python *GatewayServer* class enables Python programs to access a Java program. The implementation code for Py4J module in Java is shown in Figure 70.

```
1 import py4j.GatewayServer;
2 public static void main(String[] args)
3 {
4     // app is now the gateway.entry_point
5     GatewayServer server = new GatewayServer(app);
6     server.start();
7 }
```

Figure 70: Implementation code of Py4J Java

III. Connecting Python and JavaScript, (CSS, HTML) via REST API

The connection between Python and JavaScript, CSS and HTML programs is established via REST API architecture, as shown in Figure 71.



Figure 71: Interaction between Python Flask and JavaScript, CSS, HTML

Flask, a Python web framework, is used to retrieve and exchange data between Python and JavaScript via REST API. In order to use Flask, the library is first installed in Python by executing the command “*pip install Flask*” on the interpreter’s terminal. The implementation of the code for both Python and JavaScript are described in the following section.

Python

The POST function within the REST API can be used to enable the exchange of data. POST refers to sending information and data to a specific location. The POST call returns a response from Python, and the page is redirected to the result page with the specified unique ID. A URL for POST requests is created in Python, and the code implementation is shown in Figure 72.

```
1 from flask import Flask, jsonify
2 from flask_cors import CORS
3 app = Flask(__name__)
4
5 @app.post("/predict")
6 def predict():
7     return 'This is my prediction.'
8
9 if __name__ == '__main__':
10     app.run()
```

Figure 72: Code of implementing Flask in Python

JavaScript

After the server side in Python is set up, the fetch command on JavaScript can be used to retrieve the data from Python Flask. The code implementation in JavaScript is shown Figure 73.

```
1 let msg = { name: "User", message: text }
2 this.messages.push(msg);
3
4 fetch('http://127.0.0.1:5000/predict', {
5     method: 'POST',
6     body: JSON.stringify({ message: text }),
7     mode: 'cors',
8     headers: {
9         'Content-Type': 'application/json'
10    },
11 })
```

Figure 73: Code of fetching data from Python in JavaScript

After the implementation of the codes, the process of developing the whole digital assistant program is now complete. All steps are performed and the user interface of the digital assistant can be viewed on the front-end of a web browser. This can be done by launching the program and navigating to “*https://127.0.0.1:5000/*” in the browser to view and start using it.

4. Results and Discussion

In this chapter, the outcomes of the work are presented and discussed. The outcomes are divided into two sections: a) the final product of the front-end user-interface design and, b) the prediction results from the digital assistant program.

4.1 User Interface

The final product of the front-end development design of the digital assistant program can be seen in Figure 74.

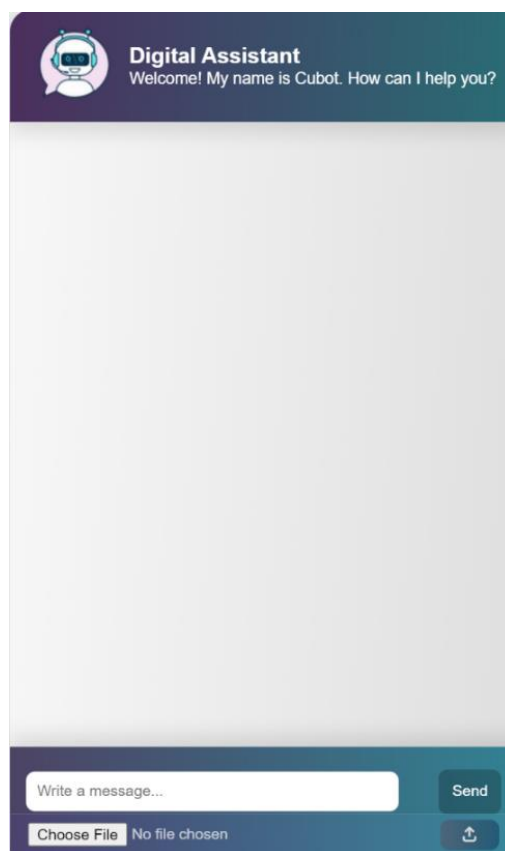


Figure 74: Front-end visualization of the digital assistant program

In the upper part of the program, the icon of the digital assistant is displayed, followed by a welcome message. “Cubot” is given as the name of the digital assistant. The lower part of the program first displays the placeholder for message entry and then the “Send” button next to it to send the message entered in the placeholder. Below that is the file upload function, which consists of the “Choose File” button to browse and select the model file to be analysed in the program and the “Upload File” icon to upload the selected model file. The exchange of messages between the user and the digital assistant is displayed in between the upper and lower parts of the program.

4.2 Questions and Answers

The following section displays the prediction results of the program based on the message input. The messages or questions were not given exactly as the specified patterns in the intent data, although similar. This is to ensure that the model can perform predictions of the desired response on unseen data or message structure.

The following section displays all the answers to the messages given to the digital assistant.

- **A-SPICE Generic Practices (GP)**

Table 22 lists the message input and response prediction for getting information on the A-SPICE Generic Practices (GP).

Table 22: Message input and response prediction of the A-SPICE Generic Practices (GP)

Message:	Hi, I would like to know the highest reachable A-SPICE level.
Answer:	As of right now, Level 2 is the highest reachable A-SPICE Level. Should I list down all the Generic Practice of A-SPICE level 2? (Yes / No)

Figure 75 shows the message exchange between the **user** and **Cubot** as displayed in the digital assistant program.

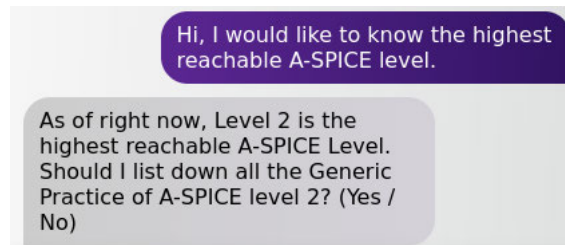


Figure 75: Message exchange between user and Cubot of A-SPICE Generic Practices (GP)

The user is then presented with two choices, “Yes” or “No”. All the GP of A-SPICE level 2 is listed if “Yes” is chosen, as shown in Table 23.

Table 23: Message input and response prediction of the A-SPICE Generic Practices (GP) if “Yes” is selected

Reply:	Yes
Answer:	<p>GP 2.1.1 Identify the objectives for the performance of the process</p> <p>GP 2.1.2 Plan the performance of the process to fulfil the identified objectives</p> <p>GP 2.1.3 Monitor the performance of the process against the plans</p> <p>GP 2.1.4 Adjust the performance of the process</p> <p>GP 2.1.5 Define responsibilities and authorities for performing the process</p> <p>GP 2.1.6 Identify, prepare and make available resources to perform the process according to plan</p> <p>GP 2.1.7 Manage the interfaces between involved parties</p> <p>GP 2.2.1 Define the requirements for the work products</p> <p>GP 2.2.2 Define the requirements for documentation and control of the work products</p> <p>GP 2.2.3 Identify, document and control the work products</p> <p>GP 2.2.4 Review and adjust work products to meet the defined requirement</p>

Figure 76 shows the message exchange between the **user** and **Cubot** on the program if “Yes” is chosen.

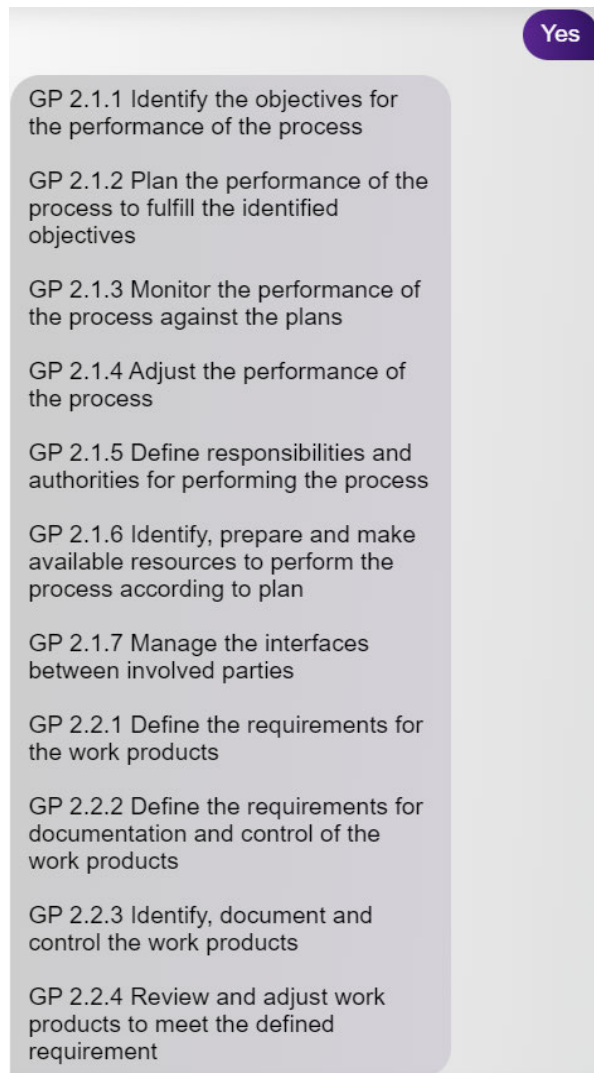


Figure 76: Message exchange between **user** and **Cubot** of A-SPICE Generic Practices (GP) if “Yes” is chosen

Table 24 shows the response if “No” is chosen.

Table 24: Message input and response prediction of the A-SPICE Generic Practices (GP) if “No” is selected

Reply	No
Answer:	Alright, please let me know if you need my assistance.

Figure 77 shows the message exchange between **user** and **Cubot** if “No” is chosen.

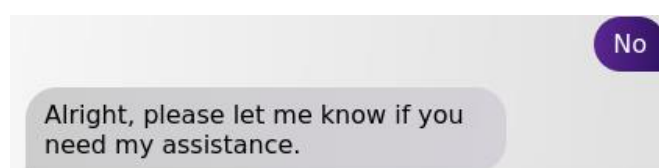


Figure 77: Message exchange between the **user** and **Cubot** of A-SPICE Generic Practices (GP) if “No” is chosen

- **Probability of the whole process reaching final state**

The message input and response prediction for getting information on the probability of the whole process reaching the final state is demonstrated in this section. Three different variations of message formulation are possible to achieve this. Table 25 demonstrates the message exchange using the first variation.

Table 25: Message input and response prediction of first message formulation of probability of the whole process

Message:	Could you please verify the GP 2.1.7?
Answer:	GP 2.1.3 and GP 2.1.7 are satisfied because we have no deadlocks in the process. The probability of reaching the final state is 100.00%. All interfaces are managed well.

Figure 78 shows the message exchange between the **user** and **Cubot** using the first variation as displayed in the digital assistant program.

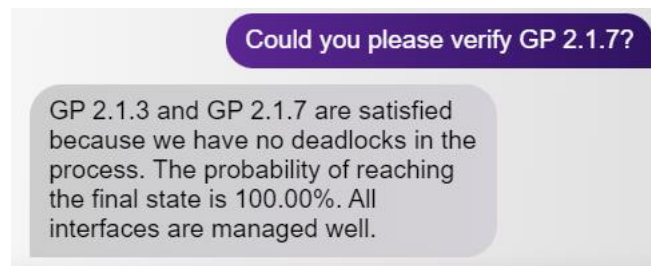


Figure 78: Message exchange between **user** and **Cubot** of probability of the whole process using first formulation variation

Table 26 shows the exchange using the second formulation variation.

Table 26: Message input and response prediction of second formulation variation of probability of the whole process

Message:	Could you please verify the GP 2.1.3?
Answer:	GP 2.1.3 and GP 2.1.7 are satisfied because we have no deadlocks in the process. The probability of reaching the final state is 100.00%. All interfaces are managed well.

Figure 79 shows the message exchange between the **user** and **Cubot** using the second formulation variation.

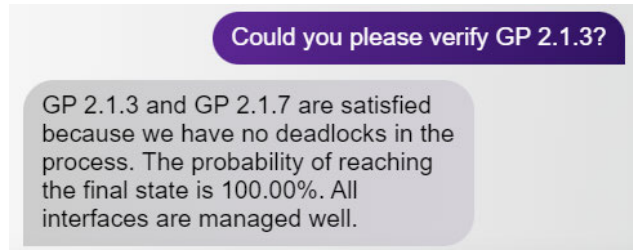


Figure 79: Message exchange between **user** and **Cubot** of probability of the whole process using the second variation

Table 27 lists the third message formulation variation.

Table 27: Message input and response prediction of third message formulation of probability of the whole process

Message:	I would like to know the probability of the whole process reaching final state
Answer:	GP 2.1.3 and GP 2.1.7 are satisfied because we have no deadlocks in the process. The probability of reaching the final state is 100.00%. All interfaces are managed well.

Figure 80 shows the message exchange between the **user** and **Cubot** using the third formulation variation.

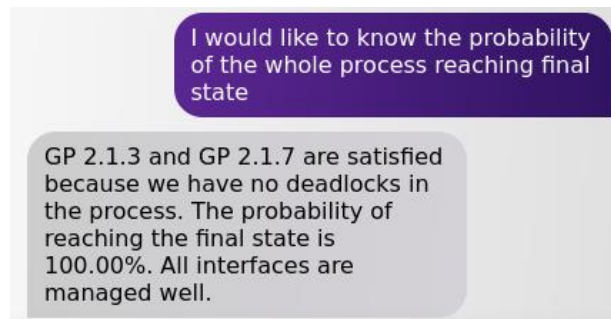


Figure 80: Message exchange between **user** and **Cubot** of probability of the whole process using the third formulation variation

All three formulation variations returned the same response as specified in the patterns of the intent data. This is due to the fact that all three formulations intend to obtain information on the same desired parameter, namely the probability of the whole process reaching the final state, and a probability of 100% leads to the compliance of GP 2.1.3 and GP 2.1.7 of A-SPICE.

Furthermore, the following section shows the prediction results of additional functions that allow users to determine the probability of a particular module or variable reaching the final state or the probability of a particular module or variable reaching a particular state.

- **Probability of a specific module reaching final state**

Table 28 demonstrates the message input and response prediction for getting information on the probability of a particular module or variable reaching the final state.

Table 28: Message input and response prediction of the probability of a variable s2b reaching final state

Message:	Could you please tell me the probability of a specific module reaching final state?
Answer:	Please enter variable number:
Reply:	s2b
Answer:	The probability of P=? [F s2b = o] reaching the final state of the process is: 100.00%. There are no deadlocks and therefore, the final state can always be reached and proves the compliance with A-SPICE Level 2 of GP 2.1.3 and GP 2.1.7

The command P=? [F s2b = o] displayed in the answer shows the temporal logic specification, which indicates the probability of user-defined variable s2b reaching the final state of o. The probability of 100% is obtained in this example proves the compliance with the A-SPICE level 2 of GP 2.1.3 and GP 2.1.7. Figure 81 shows the message exchange between the **user** and **Cubot** as displayed in the digital assistant program.

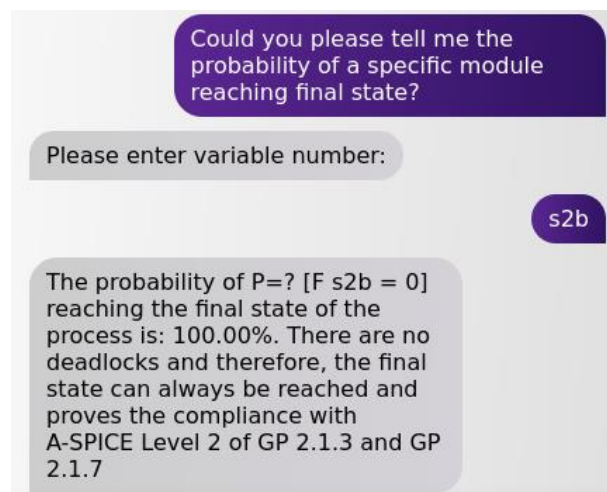


Figure 81: Message exchange between user and Cubot of probability of a specific module reaching the final state

- **Probability of a specific module reaching a specific state**

Table 29 displays the message input and response prediction for getting information on the probability of a particular module or variable reaching a particular state.

Table 29: Message input and response prediction of the probability of a variable sod reaching state o

Message:	Could you please say the probability of a specific module reaching a specific state?
Answer:	Please enter variable number:
Reply:	sod
Answer:	Please enter the state number:
Reply:	o
Answer:	The probability of variable sod to reach state o of the process is: 100.00%. There are no deadlocks in the processes and thus proves the compliance with A-SPICE Level 2 of GP 2.1.3 and GP 2.1.7

The probability of user-defined variable sod reaching user-defined state of o is 100% and this proves the compliance with the A-SPICE Level 2 of GP 2.1.3 and GP 2.1.7. Figure 82 shows the message exchange between the **user** and **Cubot** for determining the probability of a variable sod reaching state o.

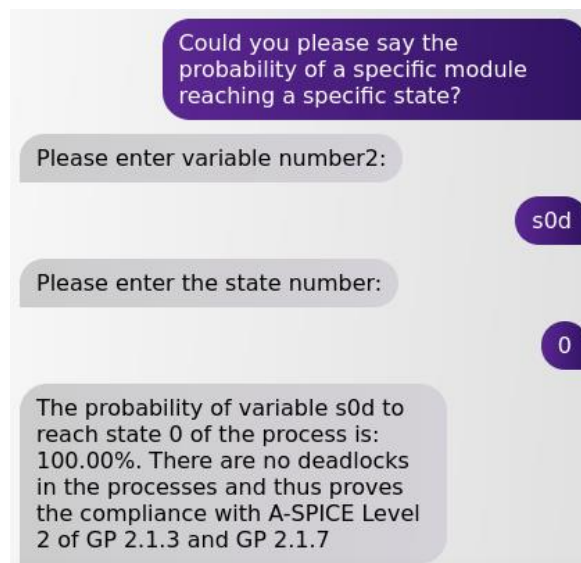


Figure 82: Message exchange between user and Cubot of probability of a variable sod reaching state o

Table 30 illustrates the response of another user-defined variable and state.

Table 30: Message input and response prediction of the probability of a variable soh reaching state 3

Message:	Tell me the probability of a specific module reaching a specific state.
Answer:	Please enter variable number:
Reply:	soh
Answer:	Please enter the state number:
Reply:	3
Answer:	The probability of variable soh to reach state 3 of the process is: 0.00%. Unfortunately, there are deadlocks in the processes. The A-SPICE Level 2 of GP 2.1.3 and GP 2.1.7 are not complied.

The probability of the user-defined variable soh reaching the user-defined final state of 3 is 0%. This indicates that there are deadlocks in the processes and the A-SPICE Level 2 of GP 2.1.3 and GP 2.1.7 are therefore not met.

Figure 83 shows the message exchange between the user and Cubot to get information on the probability of a variable soh reaching state 3.

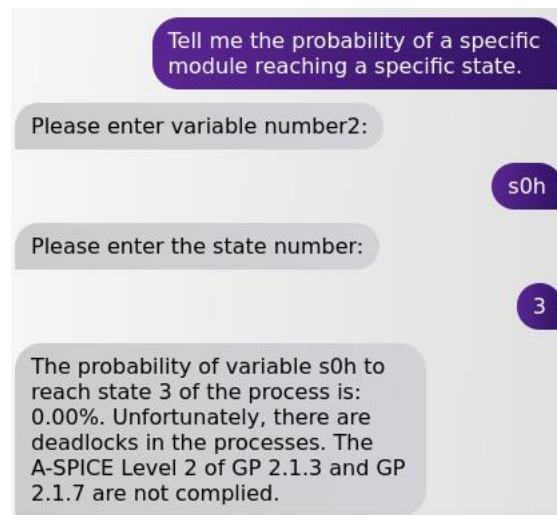


Figure 83: Message exchange between user and Cubot of probability of a variable soh reaching state 3

Next, two options are possible to obtain information about the minimum days (d) for the completion of the whole process. First, through a simulation of the minimum reward performed with the PRISM tool. Alternatively, the minimum days can be determined based on the calculation of accumulated reward of the minimum steps taken by the variables of an MDP model to reach the final state starting from an initial state.

- **Minimum days for performing whole process**

Table 31 depicts the message input and response prediction for getting information on the minimum days for performing the whole process. The user is then offered two choices: “one”, which returns the result from the PRISM simulation or “two”, which returns the result from the accumulated rewards of the MDP model. If “one” is selected, the user is also offered an additional possibility to open the file containing the simulation data for reviewing purposes. Table 31 lists the response if “one” and subsequently “Yes” are selected.

Table 31: Message input and response prediction of the minimum days for performing whole process if “one” and “Yes” are selected

Message:	Hi, I would like to know the minimum days for performing whole process.
Answer:	Please choose: (one/two) one) from PRISM Simulation two) from Rewards of MDP Model
Reply:	one
Answer:	The minimum days for performing whole process is 213.0 days. Would you like me to open the simulation data file?
Reply:	Yes (Simulation file opens in a new window)
Answer:	File opened...

Figure 84 shows the message exchange between the **user** and **Cubot** as displayed in the digital assistant program if “one” and subsequently “Yes” are selected.

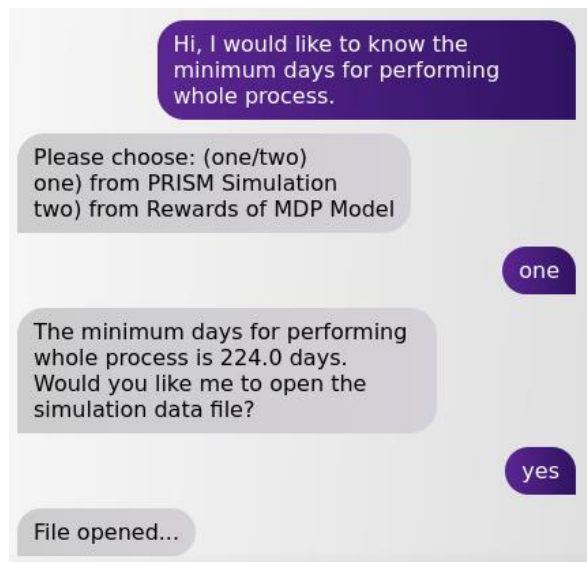


Figure 84: Message exchange between user and Cubot of minimum days for performing whole process if “one” and “Yes” are selected

Table 32 shows the response if “two” is chosen.

Table 32: Message input and response prediction of the minimum days for performing whole process if “two” is selected

Message:	Tell me the minimum days for performing whole process.
Answer:	Please choose: (one/two) one) from PRISM Simulation two) from Rewards of MDP Model
Reply:	two
Answer:	The minimum days for performing whole process from Rewards of MDP Model is 145.67 days.

Figure 85 shows the message exchange between the **user** and **Cubot** as displayed in the digital assistant program if “two” is selected.

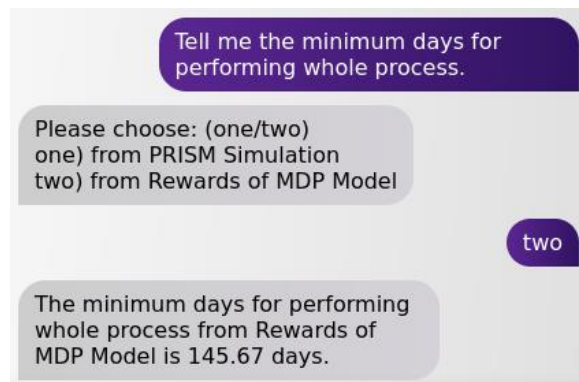


Figure 85: Message exchange between user and Cubot of minimum days for performing whole process if “two” is selected

It is to note that the results of “one” are not consistent, but produces a different outcome every iteration. This is due to the fact that the results are computed with the simulator of the PRISM tool, which produces a different result for each simulation, although with only a slight difference. Option “two”, on the other hand, always gives a consistent result, since it only depends on the calculation of a fixed set of rewards of a given MDP model.

- **Estimation of expected minimum days for performing whole process**

The message input and response prediction for getting information on the estimation of the expected minimum days for performing the whole process is demonstrated in this section. This parameter verifies the property 3, φ_3 of A-SPICE. Table 33 lists the message exchange.

Table 33: Message input and response prediction of the estimation of expected minimum days for performing whole process

Message:	Could you please tell me the estimation of expected minimum days for performing whole process?
Answer:	The estimation of expected minimum days for performing whole process is 254.85 days.

Figure 86 shows the message exchange between the **user** and **Cubot** as displayed in the digital assistant program.

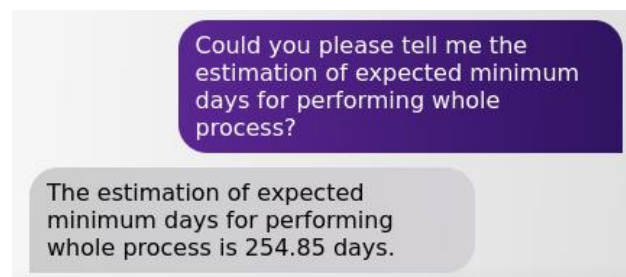


Figure 86: Message exchange between user and Cubot of estimation of expected minimum days for performing the whole process

Furthermore, similar to the probability-based property, there are additional features that allow the user to specify the expected minimum rewards (days) for a particular module or variable to reach the final state, or the expected minimum (days) for a particular module or variable to reach a given state.

- **Estimation of expected minimum days a specific module reaching final state**

Table 34 shows the message input and response prediction for getting information on the expected minimum days of a particular module or variable reaching the final state.

Table 34: Message input and response prediction of the estimation of expected minimum days variable s1a reaching the final state

Message:	I would like to know the estimation of expected minimum days of a specific module reaching final state.
Answer:	Please enter variable number:
Reply:	s1a
Answer:	The estimation of expected minimum days for variable s1a to reach final state is 18.17 days.

The expected minimum days for the given variable s1a to reach the final state is returned as the answer. Figure 87 shows the message exchange between the **user** and **Cubot** in the program.

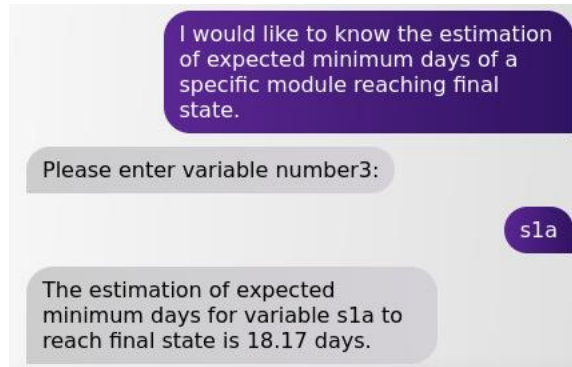


Figure 87: Message exchange between user and Cubot of estimation of expected minimum days for variable s1a reaching the final state

- **Estimation of expected minimum days a specific module reaching a specific state**

Table 35 presents the message input and response prediction for getting information on the expected minimum days of a particular module or variable reaching a particular state.

Table 35: Message input and response prediction of the estimation of expected minimum days variable s1e reaching the state 3

Message:	Please share with me the expected minimum days of a specific module reaching a specific state?
Answer:	Please enter variable number:
Reply:	s1e
Answer:	Please enter the state number:
Reply:	3
Answer:	The estimation of expected minimum days for variable s1e to reach state 3 is 24.17 days.

The expected minimum days for the given variable s1e to reach the given state of 3 is returned as the answer. Figure 88 shows the message exchange between the **user** and **Cubot** in the digital assistant program.

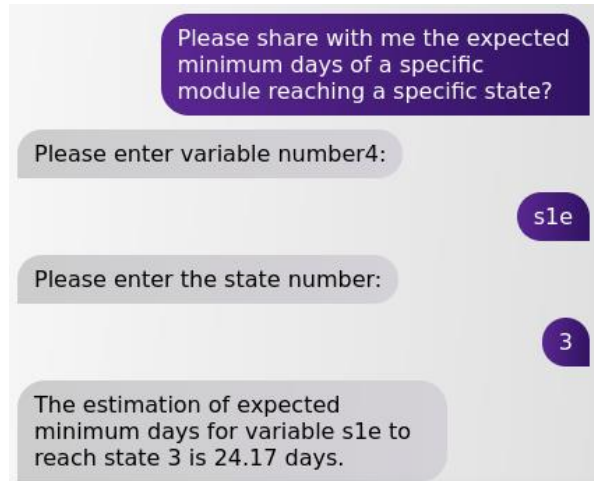


Figure 88: Message exchange between user and Cubot of estimation of expected minimum days for variable *s1e* reaching the state 3

- **Total number of transitions**

Table 36 displays the message input and response prediction for getting information on the total number of process transitions in the MDP model.

Table 36: Message input and response prediction of the total number of transitions

Message:	Could you tell me the total number of transitions?
Answer:	The total number of transitions of the processes is 352892.

Figure 89 shows the message exchange between the **user** and **Cubot** as displayed in the digital assistant program.

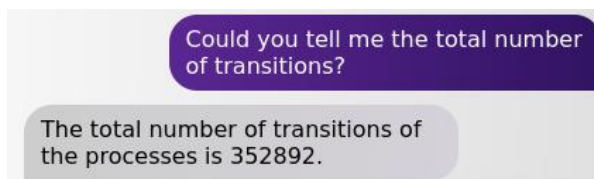


Figure 89: Message exchange between user and Cubot of total number of transitions

- **Total number of states**

Table 37 shows the message input and response prediction for getting information on the total number of process states in the MDP model.

Table 37: Message input and response prediction of the total number of states

Message:	Share with me the total number of states.
Answer:	The total number of states of the processes is 73102.

Figure 90 shows the message exchange between the **user** and **Cubot**.

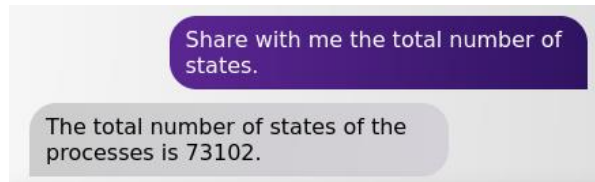


Figure 90: Message exchange between user and Cubot of total number of states

- **Value of initial state**

Table 38 displays the message input and response prediction for getting the value of the initial state of the processes in the MDP model.

Table 38: Message input and response prediction of the value of initial state

Message:	I would like to know the value of initial state.
Answer:	The value of initial state of the processes is 1.

Figure 91 visualizes the message exchange between the **user** and **Cubot** in the program.

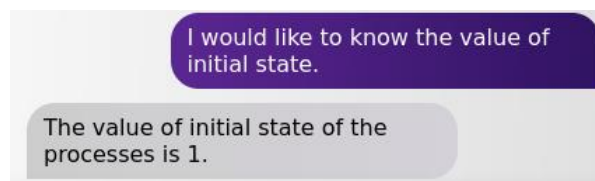


Figure 91: Message exchange between user and Cubot of value of the initial state

- **Model construction_time**

Table 39 shows the message input and response prediction for getting information on the MDP model construction time.

Table 39: Message input and response prediction of the model construction time

Message:	What's the construction model time?
Answer:	Time for model construction is 0.824 seconds.

Figure 92 shows the message exchange between the **user** and **Cubot** as displayed in the digital assistant program.

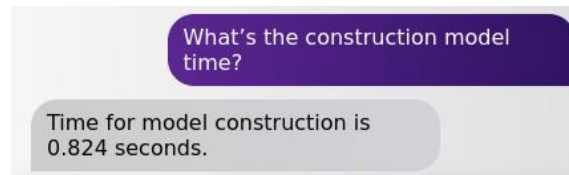


Figure 92: Message exchange between user and Cubot of the model construction time

- **Model checking time**

Table 40 lists the message input and response prediction for getting information on the MDP model checking time.

Table 40: Message input and response prediction of the model checking time

Message:	I would like to know the model checking time.
Answer:	Time for model checking is 1.124 seconds.

Figure 93 shows the message exchange between the **user** and **Cubot**.

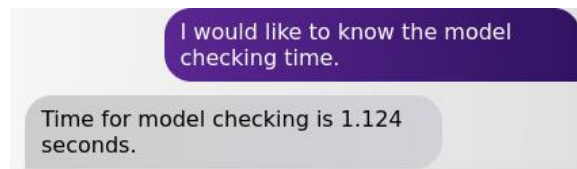


Figure 93: Message exchange between user and Cubot of the model checking time

- **List of modules names**

Table 41 displays the message input and response prediction for getting the complete list of modules names of the MDP model.

Table 41: Message input and response prediction of the list of modules names

Message:	Could you please list down all modules name.
Answer:	The names of the modules are [Moa, Mob, Moc, Mod, Moe, Mof, Mog, Moh, Moi, Moj, M1a, M1b, M1c, M1d, M1e, M2a, M2b, M2c, M2d, M2e, M3, M4, M5a, M5b, M5c, M5d].

Figure 94 illustrates the message exchange between the **user** and **Cubot** for requesting the list of modules names.

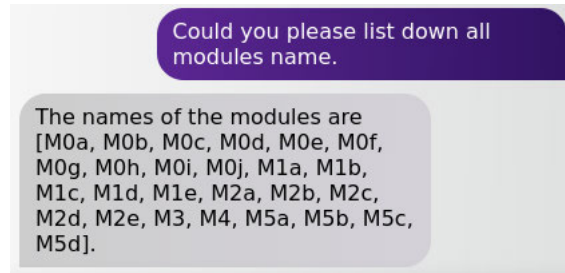


Figure 94: Message exchange between user and Cubot of the list of modules names

- **List of variables names**

Table 42 displays the message input and response prediction for getting the complete list of variables names of the MDP model.

Table 42: Message input and response prediction of the list of variables names

Message:	List down all variables name please.
Answer:	The names of the variables are [soa, sob, soc, sod, soe, sof, sog, soh, soi, soj, s1a, s1b, s1c, s1d, s1e, s2a, s2b, s2c, s2d, s2e, s3, s4, s5a, s5b, s5c, s5d].

Figure 95 shows the message exchange between the **user** and **Cubot** for requesting the list of variables names.

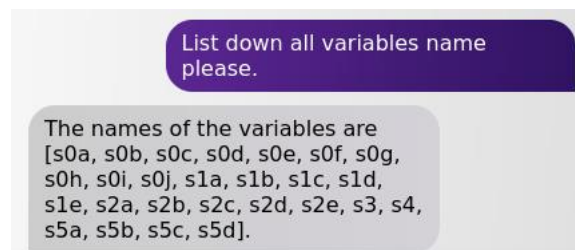


Figure 95: Message exchange between user and Cubot of the list of variables names

All the predictions of responses were accurate even when the messages given were not formulated exactly as specified in the patterns of the intent data. This shows that the Deep Learning model has been successfully developed to make correct predictions of the desired results from unseen message structures.

Through multiple trial and error, the keywords of all patterns of each tag were determined. It was found that, as long as the keyword is present in the user's message, the program always returns an accurate desired response to that message. The keywords represent the most frequently occurring words in the patterns of a tag in the intent data, as determined from the BoW method.

5. Conclusion and Outlook

This chapter summarizes the content of all previous sections in order to recap and outline what has been accomplished. It gives an overall view of the completed work, highlights some of the limitations of the work and provides pointers for future developments.

To recap, model checking is a technique for verifying the correctness of a software system in order to ensure safety, avoid errors and defects in the system, and thus comply with the A-SPICE standard of the automotive industry. Unlike the conventional testing method for verification, model checking considers all possible behaviours of a system. Unfortunately, although model checking is a powerful tool for verifying the correctness of a system, its complex language and unfamiliar specification methods make it difficult for organizations to fully implement it and for auditors to evaluate an organization's compliance with A-SPICE. Therefore, this work dealt with the development of a digital assistant that facilitates model checking for everyone. In its development, the digital assistant applies state-of-the-art NLP methods, which is part of AI that makes natural texts in human language understandable to computers. The focus of this work is on the PEP of the automotive industry, but the program can also be applied in any other industry that implements embedded software systems, robotics or biological systems. For instance, model checking is increasingly being applied in industries involving (safety) critical systems such as process plant control or railway systems (Per, et al., 2001).

The process of developing the digital assistant program goes through the following steps. First, the business requirements were studied and understood. This includes defining the requirements of the A-SPICE standard, which can be verified with the PRISM tool. Second, the available data in this work were examined and analysed. The data available in this work are multiple MDP models of a general workflow process of "*testing an autonomous parking pilot*" of PEP of an automotive supplier. The process first describes the main steps of starting with the requirements for the test area, and subsequently the testing phase. Third, the formal specification of quantitative properties of the system being analysed (MDP models) were constructed and prepared to perform probabilistic model checking. Fourth, the process of modelling the algorithms to develop the program were carried out. The modelling process can be divided into four parts: a) creation of the intent data, b) implementation of the training pipeline, c) launching of the program, and d) development of the front-end visualization design. Finally, the various programming languages used in the development of the program were integrated together through the use of APIs or appropriate libraries. This involves the connection of PRISM and Java languages via an API, connection of Java and Python languages via Py4J library and connection of Python and JavaScript, CSS, and HTML via REST API.

Several considerations were taken into account during the development of the program to ensure the accuracy of the predictive models. One of them is the prevention of overfitting. Overfitting is a phenomenon where the model is fitted exactly to the training data, which causes the algorithm to produce inaccurate results on unseen data and thus defeats its purpose. Therefore, it is important to avoid overfitting the data. To achieve this, the early stopping method was used in this work. Early stopping is a technique to determine when to stop training, thus prevents the overfitting of the neural network. The training phase of the network is stopped at the point of convergence, i.e. after a certain number of epochs, when the performance no longer improves. The corresponding number of epochs was then used as the optimal hyper-parameter value for the training process.

Based on the results, this work can be considered a success, as the main objectives have been achieved. The digital assistant program has been properly developed in both the front-end and the backend, and the data have been successfully exchanged between the digital assistant and the PRISM tool. Besides that, all the questions from the user have been answered correctly by the digital assistant, provided that the questions did not fall far outside the patterns defined in the intent data. Furthermore, the objective of reducing the difficulty and effort in performing model checking can be considered to be achieved, as the user only requires to ask a simple model checking question in natural human language and the digital assistant will provide the accurate desired answer to this question with the help of the PRISM tool. In order to make the implementation of model checking even simpler, a list of parameters that the digital assistant can perform and a template of sample questions were created and provided on the program's homepage to assist the user in obtaining a certain result and to ensure that the user receives the most accurate result to their question. Certainly, the user is free to formulate the question in any way he/she prefers, as long as the keyword is present and the question falls within the scope of the specified patterns, but in order to always get the most accurate answer to the question, it is recommended that the user follows the question examples provided in the template.

In order to even increase the performance of the digital assistant, especially for prediction of questions that fall outside of the question patterns in the intent data, much more data needs to be provided and collected by the program because Deep Learning works better with increasing amounts of data, since it can perform better pattern recognition and thus, provide a more accurate prediction. A feedback feature needs to be developed and deployed during the testing period where users can give feedback in a form of a thumbs up click if the digital assistant returns the correct answer to their question, or otherwise, a thumbs down click if the digital assistant returns an incorrect answer. If a thumbs up is given, the question asked by the user is added to the list of question patterns of the respective tag in the intent data. If a thumbs down is given, a similar question is removed from the list of question patterns of the respective tag. This can help optimize the program and reduce the amount of time and effort required to constantly update and provide new input data to the program manually.

Another possibility to further increase the performance of the program is to apply alternative or additional NLP methods. In addition to the NLP pre-processing applied in this work, other additional methods such as Parts-of-Speech (POS) tagging and Stop-Words Removal can be used in the future work, which could potentially improve the quality of the pre-processed data. The methods used in this work are rather simplified, since the amount of data processed is not too large. For work with much larger data sets, the additional NLP pre-processing methods could be very helpful. Firstly, POS tagging is a method for describing the grammatical function of a word. It involves augmenting the input text with additional information about the grammatical structure of the sentence. This helps in forming named units (nouns) and extracting relationships between words. On the other hand, Stop-Words-Removal is simply the process of removing words that occur frequently in all documents in the corpus, including articles and pronouns. In this way, redundant low-level information is removed from the text to bring the important information to the focus (Sreemany, 2021).

In addition to the PyTorch framework, future work could instead apply the TensorFlow framework to develop the Deep Learning models for the program. Some of the advantages of TensorFlow are that it provides a flexible, extensive ecosystem of community resources and libraries that facilitate the development and deployment of machine learning applications. It is also an open-source platform, available to all users, and can be developed on any system. TensorFlow is also compatible with the Keras library, which allows users to code some high-level functional sections. Keras provides TensorFlow with system-specific functionality, such as pipelining, estimators, and eager execution. For these reasons, a comparison of predictive performance between TensorFlow and PyTorch framework would be of interest.

However, it must be noted that the digital assistant has certain limitations that could be improved. First of all, although the digital assistant returns one of the responses specified in the intent data, it is not possible to return a response that does not already exist in the dataset, i.e. the response which will be returned for a given question is only limited only to the list of responses specified in the intent data. This limitation may not be a problem if the goal is only to verify the process and obtain the accurate model checking results, but if one intends on making the digital assistant more intelligent, interactive and human-like, this limitation should be overcome.

Another limitation of the digital assistant is that, however good its performance may be from this work, the digital assistant still in general has a high likelihood of misunderstanding the user's intent. This is especially the case when it has to deal with messages that have a completely different language style or structure than the one it has normally been trained with. Furthermore, occasionally the message provided by the user is not very clear, too specific, or given in a foreign language, making it difficult for the program to understand the intent. In this case, the user experience with the program will not be very pleasant and can have a negative impact on the verification process. Overall, the answer prediction of the digital assistant should have flawless accuracy to ensure that the user receives the most accurate model checking results at all times, especially when dealing with safety-critical software systems. Therefore, it is recommended to always have a redundant model checking tool or assistant available to validate the answers given by the digital assistant. Alternatively, a program could be developed in future work that automatically validates the correctness of the answers given by the digital assistant, which can be considered as a form of a verification system for the digital assistant.

References

1. 19th International Conference, V. 2., 2018. *Verification, Model Checking, and Abstract Interpretation*. Los Angeles: Springer International Publishing.
2. Ala-Ilomäki, K., 2019. *Application Programming Interface Management For Cloud Entities Of An Enterprise Resource Planning Software*, Lappeenranta: LUT University, School of Engineering Science.
3. Alasmari, N., n.d. *Probabilistic Model Checking Techniques for the Verification and Synthesis of Software Systems*, Heslington: Computer Science, University of York.
4. Allweyer, T., 2016. *BPMN 2.0: Introduction to the Standard for Business Process Modeling*. Norderstedt: Books on Demand.
5. Anon., 2022. *ASPICE 101: What You Need to Know About Automotive SPICE*. [Online] Available at: <https://content.intland.com/blog/aspice-101-what-you-need-to-know-about-automotive-spice> [Accessed 10 07 2022].
6. Automotive SIG, V. Q. W. G. 1., 2015. *Automotive SPICE® Process Reference Model Process Assessment Model Version 3.0*, s.l.: VDA QMC - Quality Management in the Automotive Industry.
7. Blanck, M., 2019. *Predicting Price Residuals in Online Car Marketplaces with Natural Language Processing*, Karlsruhe: Institute of Applied Informatics and Formal Description Methods (AIFB).
8. Bui, D. H., 2018. *Design and Evaluation of a Collaborative Approach for API Lifecycle Management*, Munich: Department of Informatics, Technical University of Munich.
9. Carlos, J., 2019. *Tensors for busy people*. [Online] Available at: <https://dev.to/juancarlospaco/tensors-for-busy-people-315k> [Accessed 12 August 2022].
10. Chavan, J., 2020. *NLP: Tokenization, Stemming, Lemmatization, Bag of Words, TF-IDF, POS*. [Online] Available at: <https://medium.com/@jeevanchavan143/nlp-tokenization-stemming-lemmatization-bag-of-words-tf-idf-pos-765of83c6obe> [Accessed 12 August 2022].
11. Choi, J., 2013. *Model Checking for Decision Making Behaviour of Heterogeneous Multi-Agent Autonomous System*, Cranfield: Cranfield University, Department of Informatics and Systems Engineering.
12. Chtourou, A., 2016. *Graduation Master Thesis Towards model checking TCTLΔh for Timed Concurrent Systems*, Paris: Institut Mines-Télécom.
13. CiGi Technology Limited, 2022. *Industries*. [Online] Available at: <https://www.cigitechnology.com/industries/> [Accessed 12 August 2022].
14. Clarke, E. M., 2018. *Handbook of Model Checking*. Pennsylvania: Springer.

15. Comindware, 2019. *6 Business Process Examples and Automation Ideas*. [Online] Available at: <https://www.comindware.com/blog-6-business-process-examples-automation-ideas/> [Accessed 12 August 2022].
16. Deng, L. & Liu, Y., 2018. *Deep Learning in Natural Language Processing*. Singapore: Springer .
17. Diamandis, P. H. & Kotler, S., 2020. *The Future Is Faster Than You Think: How Converging Technologies Are Transforming Business, Industries, and Our Lives*. New York: Simon & Schuster; 1st edition .
18. Drechsler, R., 2017. *Formal System Verification, State-of The-Art and Future Trends*. Bremen: Springer International Publishing.
19. Feldhusen, J. & Grote, K.-H., 2013. Der Produktentstehungsprozess (PEP). In: J. Feldhusen & K. Grote, eds. *Pahl/Beitz Konstruktionslehre*. Berlin, Heidelberg: Springer Vieweg, p. 11-24.
20. Feldhusen, J. & Grote, K.-H., 2013. *Der Produktentstehungsprozess (PEP)*. Berlin Heidelberg: Springer.
21. Fielding, R. T. & Taylor, R. N., 2002. *Principled Design of the Modern Web Architecture*, California: Information and Computer Science, University of California, Irvine.
22. Flask-CORS, 2022. *Flask-CORS: Documentation*. [Online] Available at: <https://flask-cors.readthedocs.io/en/latest/> [Accessed 12 August 2022].
23. FLI, F. o. L. I., 2022. *Benefits & risks of artificial intelligence*. [Online] Available at: <https://futureoflife.org/background/benefits-risks-of-artificial-intelligence/> [Accessed 13 August 2022].
24. Garber, L., 2013. The Lowly API Is Ready to Step Front and Center. *Computer*, 46(8), pp. 14-17.
25. Godefroid, P. & Sen, K., 2018. Combining Model Checking and Testing. *Handbook of Model Checking*, pp. 613-649.
26. Grinde, A. B. & Johansen, B. M., 2021. *Using Natural Language Processing with Deep Learning to Explore Clinical Notes*, Bergen: Department of Informatics, University of Bergen.
27. Grochowski, E. M. et al., 2015. *Produktentstehungsprozess in den frühen Phasen unter Berücksichtigung der Herausforderungen im Forschungscampus ARENA2036*. Reutlingen: Hochschule Reutlingen.
28. H. Sultan, H., Salem, N. & Al-Atabany, W., 2019. Multi-Classification of Brain Tumor Images Using Deep Neural Network. *IEEE Access*, Volume PP, pp. 1-1.
29. Hage, H., Hashemi, V. & Mantwill, F., 2020. *Towards a Systems Engineering based Automotive Product Engineering Process*, s.l.: arXiv.

30. Hage, H., Seferis, E., Hashemi, V. & Mantwill, F., 2022. SMC4PEP: Stochastic Model Checking of Product. *ArXiv*.
31. Han, S., 2019. *Business process automation through ChatBots implementation: A case study of an IT service process at philips*, Delft: Delft University of Technology,.
32. Heath, J. et al., 2008. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science* 391, pp. 239-257.
33. Hensel, C. et al., 2022. The probabilistic model checker STORM. *International Journal on Software Tools for Technology Transfer*, p. 589–610.
34. Herbert, L. T., Sharp, R. & Hansen, M. R., 2014. *Specification, Verification and Optimisation of Business Processes, A Unified Framework*, Kongens Lyngby: DTU COMPUTE - Department of Applied Mathematics and Computer Science, Technical University of Denmark.
35. Horvat, A., 2021. *Analyse der Named Entity Recognition des KI-Modells BERT in einer spezifischen Sprachdomäne*, Ingolstadt: Informatik, Technische Hochschule Ingolstadt.
36. IBM Cloud Education, 2020. *Neural Networks*. [Online] Available at: <https://www.ibm.com/cloud/learn/neural-networks> [Accessed 12 August 2022].
37. IBM, C. E., 2020. *Application Programming Interface (API)*. [Online] Available at: <https://www.ibm.com/cloud/learn/api> [Accessed 13 August 2022].
38. IBM, C. E., 2022. *The Basics of Business Process Modeling and Notation (BPMN)*. [Online] Available at: <https://www.ibm.com/cloud/blog/bpmn> [Accessed 10 8 2022].
39. Jain, M., 2021. *Tokenization in Natural Language Processing*. [Online] Available at: <https://byteiota.com/tokenization/> [Accessed 12 August 2022].
40. Jia, R., 2020. *Building Robust Natural Language Processing Systems*, Stanford: Department of Computer Science, Stanford University.
41. Kalla, S., 2021. *Neural Networks*. [Online] Available at: <https://medium.com/nerd-for-tech/neural-networks-68531432fb5> [Accessed 12 August 2022].
42. Katoen, J.-P., 2016. *The Probabilistic Model Checking Landscape*, Twente: RWTH Aachen University, Germany and University of Twente.
43. Kaur, J., 2022. *Natural Language Processing Applications and Techniques*. [Online] Available at: <https://www.xenonstack.com/blog/natural-language-processing>
44. Keras, 2022. *Adam*. [Online] Available at: <https://keras.io/api/optimizers/adam/> [Accessed 12 August 2022].

45. Khanna, A., 2022. *Tokenization Techniques in Natural Language Processing in Python*. [Online]
Available at: https://medium.com/@ajay_khanna/tokenization-techniques-in-natural-language-processing-67bb22088c75
46. Kingma, D. P. & Ba, J., 2017. *Adam: A Method for Stochastic Optimization*, s.l.: arXiv.
47. Konecki, P., 2019. *Chatbots in education – providing instant assistance to students*, Innsbruck: Information Systems, Production and Logistics Management, University of Innsbruck.
48. Kuipers, M. & Prasad, R., 2022. Journey of Artificial Intelligence. *Wireless Personal Communications*, 123(1).
49. Kwiatkowska, M., Norman, G. & Parker, D., 2009. PRISM: probabilistic model checking for performance and reliability analysis. *ACM SIGMETRICS Performance Evaluation Review*, 36(4), pp. 40-45.
50. Kwiatkowska, M., Norman, G. & Parker, D., 2017. Probabilistic Model Checking: Advances and Applications. In: *Formal System Verification*. Bremen: Springer, pp. 73-121.
51. Leveson, N., 1993. *Medical Devices: The Therac-25*, Washington: University of Washington.
52. Loeber, P., 2020. *Chat Bot With PyTorch - NLP And Deep Learning*. [Online]
Available at: <https://www.python-engineer.com/posts/chatbot-pytorch/>
[Accessed 14 June 2022].
53. Mazhandu, F., 2021. *Quantum Computing Basics*. [Online]
Available at: <https://tealfeed.com/quantum-computing-basics-c851f>
[Accessed 26 August 2022].
54. Mirtaheri, S. L. & Shahbazian, R., 2022. *Machine Learning Theory to Applications*. 1st Edition ed. Florida: CRC Press.
55. Nacario, K., 2020. *Chat.js Front-end Starter*. [Online]
Available at: <https://github.com/hitchcliff/front-end-chatjs>
[Accessed 10 August 2022].
56. Nadir, S. A., 2017. *Software Development and Correction Estimation in the Automotive Domain*, Thuringia: Fakultät Informatik und Automatisierung der Technischen Universität Ilmenau.
57. Narang, S., Elsen, E., Diamos, G. & Sengupta, S., 2017. *Exploring Sparsity in Recurrent Neural Networks*, s.l.: arXiv.
58. Narayan, A., 2020. *Lemmatization And Stemming In Natural Language Processing*. [Online]
Available at: https://github.com/Learn-Write-Repeat/Open-contributions/blob/master/B2-NLP/Amey_Nlp_Lemmatization_stemming.md
[Accessed 12 August 2022].

59. Nemeskey, D. M., 2020. *Natural Language Processing Methods for Language Modeling*, Budapest: Doctoral School of Informatics Faculty of Informatics, Eötvös Loránd University.
60. Nielsen, L. R., 2017. *Language modelling using deep learning - Generating answers to medical questions using recurrent neural networks*, Kongens Lyngby: Technical University of Denmark, Department of Applied Mathematics and Computer Science.
61. NLTK, 2022. *Natural Language Toolkit*. [Online] Available at: <https://www.nltk.org/> [Accessed 12 August 2022].
62. Nocks, L., 2008. *The Robot: The Life Story of a Technology*. s.l.:Johns Hopkins University Press.
63. NumPy, 2022. *NumPy*. [Online] Available at: <https://numpy.org/> [Accessed 12 August 2022].
64. O'Connor, R., 2022. *PyTorch vs TensorFlow in 2022*. [Online] Available at: <https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/> [Accessed 12 August 2022].
65. Oldenkamp, H., 2007. *Probabilistic model checking: A comparison of tools*, Twente: Faculty EEMCS, Computer Science Department, University of Twente.
66. Oracle, 2022. *What Is a Digital Assistant?*. [Online] Available at: <https://www.oracle.com/chatbots/what-is-a-digital-assistant/> [Accessed 10 August 2022].
67. Pallets, 2010. *Flask's documentation: User's Guide*. [Online] Available at: <https://flask.palletsprojects.com/en/2.2.x/> [Accessed 12 August 2022].
68. Parker, D., 2022. *prism-api*. [Online] Available at: <https://github.com/prismmodelchecker/prism-api> [Accessed 12 August 2022].
69. Parker, D. D., 2011. *Probabilistic Model Checking, Lecture 1, Introduction*, Oxford: Department of Computer Science, University of Oxford.
70. Parker, D. D., 2011. *Probabilistic Model Checking, Lecture 12: Markov Decision Processes*, Oxford : Department of Computer Science, University of Oxford .
71. Pelayo, R., 2022. *Using RESTful APIs with ESP32*. [Online] Available at: <https://www.teachmicro.com/esp32-restful-api/> [Accessed 12 August 2022].
72. Per, C., Ricerca, L., Tecnologica, S. & Cimatti, A., 2001. *Industrial Applications Of Model Checking*. o6.
73. Peters & Florian, 2017-2018. *Design and implementation of a chatbot in the context of customer support*, Liège: University of Liège - Faculty of Applied Sciences.

74. Pogoyants, A., Segala, R. & Lynch, N., 2000. *Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study*, Cambridge: Laboratory for Computer Science, Massachusetts Institute of Technology.
75. Py4J, 2009. *Py4J - A Bridge between Python and Java*. [Online] Available at: <https://www.py4j.org/> [Accessed 12 August 2022].
76. Rainergewalt, 2022. *Perceptrons – These Artificial Neurons are the Fundamentals of Neural Networks*. [Online] Available at: <https://starship-knowledge.com/neural-networks-perceptrons> [Accessed 12 August 2022].
77. Raj, S., 2018. *Building Chatbots with Python: Using Natural Language Processing and Machine Learning*. Kamataka: Apress.
78. Ranjan, A., 2020. *Natural Language Processing*. [Online] Available at: <https://medium.com/analytics-vidhya/natural-language-processing-co1b661ocfa4> [Accessed 12 August 2022].
79. Rao, D. & McMahan, B., 2019. *Natural Language Processing With PyTorch: Build Intelligent Language Applications Using Deep Learning*. California: O'Reilly Media, Inc..
80. Riel, A. et al., 2014. Facilitating Integration to face modern Quality Challenges in Automotive. *Procedia Engineering*.
81. Rizk, Y. et al., 2020. A Conversational Digital Assistant for Intelligent Process Automation. *Business Process Management: Blockchain and Robotic Process Automation Forum, BPM 2020 Blockchain and RPA Forum*, pp. 85-100.
82. Roller, D. & Engesser, E., 2014. BPMN Process Design for Complex Product Development and Production. In: E. Plödereder, L. A. S. E. Grunske & D. Ull, eds. *Informatik*. Stuttgart: Institute of Computer-aided Product Development Systems, Universität Stuttgart, pp. 1979-1984.
83. Rosemann, M. & vom Brocke, J., 2010. *Handbook on Business Process Management 1: Introduction, Methods, and Information Systems*. Berlin Heidelberg: Springer.
84. Schär, A., 2018. *Chatbots/conversational interfaces in the context of the stereotype content model (SCM)*, Zurich: Winterthur : ZHAW Zürcher Hochschule für Angewandte Wissenschaften.
85. Schwoon, S., 2002. *Model-Checking Pushdown Systems*, Munich: Lehrstuhl für Informatik VII - Technische Universität München.
86. Sevarac, Z., 2022. *From Basic Machine Learning to Deep Learning in 5 Minutes*. [Online] Available at: <https://www.deepnetts.com/blog/from-basic-machine-learning-to-deep-learning-in-5-minutes.html> [Accessed 12 August 2022].
87. Sharma, V., 2019. *Data Visualization for Deep Learning Model Using Matplotlib*. [Online] Available at: <https://www.pluralsight.com/guides/data-visualization-deep-learning->

model-using-matplotlib
[Accessed 13 August 2022].

88. Spork, G., 2011. *Efficient Requirements Management Considering Automotive Standards: Best Practice Sharing of Mechatronic Engineering within MAGNA Groups*, Graz: Production Science and Management of Frank Stronach Institute.
89. Sreemany, T., 2021. *Essential Text Pre-processing Techniques for NLP!*. [Online] Available at: https://www.analyticsvidhya.com/blog/author/tithi_sreemany/ [Accessed 12 August 2022].
90. Sreemany, T., 2021. *Essential Text Pre-processing Techniques for NLP!*. [Online] Available at: <https://www.analyticsvidhya.com/blog/2021/09/essential-text-pre-processing-techniques-for-nlp/> [Accessed 12 August 2022].
91. Stahl, A., 2021. *How AI Will Impact The Future Of Work And Life*. [Online] Available at: <https://www.forbes.com/sites/ashleystahl/2021/03/10/how-ai-will-impact-the-future-of-work-and-life/?sh=371ac7f79a30>
92. Stange, A. A., 2019. *Model Checking for SCCharts*, Kiel: Kiel University, Department of Computer Science.
93. Steven Bird, E. K. E. L., 2009. *Natural Language Processing with Python*. Sebastopol: O'Reilly Media.
94. Stevens, E., Antiga, L. & Viehmann, T., 2020. *Deep Learning with PyTorch: Build, train, and tune neural networks using Python tools*. New York: Manning.
95. Suchenia, A., Wiśniewski, P. & Ligeza, A., 2017. *Overview of Verification Tools for Business Process Models*. Prague, 2017 Federated Conference on Computer Science and Information Systems, pp. 295-302.
96. TensorFlow, 2022. *TensorFlow*. [Online] Available at: <https://www.tensorflow.org/> [Accessed 13 August 2022].
97. Thanaki, J., 2017. *Python Natural Language Processing: Advanced machine learning and deep learning techniques for natural language processing*. Birmingham: Packt Publishing.
98. Turing, 2022. *Stemming vs Lemmatization in Python*. [Online] Available at: <https://www.turing.com/kb/stemming-vs-lemmatization-in-python> [Accessed 13 August 2022].
99. Weske, M., 2012. *Business Process Management: Concepts, Languages, Architectures*. Postdam: Springer.
100. Wilfinger, M., 2017. *Development Process Optimization of Mechatronic Systems in Automotive Applications*, Graz: Faculty of Mechanical Engineering and Economic Sciences, Graz University of TechnologyGraz University of Technology.
101. Zhao, Y., 2016. *Online Model Checking Mechanism and Its Applications*, Paderborn: Faculty of Electrical Engineering, Computer Science, and Mathematics, University of Paderborn.

Table of Figure

Figure 1: The growth in processing power of quantum computers between 1998 and 2020 (Mazhandu, 2021)	2
Figure 2: Process model of Product Development and Production Control (Roller & Engesser, 2014)	5
Figure 3: An overview of the previous work	6
Figure 4: Business Process Lifecycle (Weske, 2012)	12
Figure 5: BPMN flow object symbols and connecting objects (Suchenia, et al., 2017)	14
Figure 6: Process of placing an order of a buyer (Weske, 2012)	15
Figure 7: Interacting multiple business processes (Weske, 2012)	16
Figure 8: Phases of Product Engineering Processes (PEP) (Grochowski, et al., 2015)	16
Figure 9: Process Capability Level according to A-SPICE (CiGi Technology Limited, 2022).....	18
Figure 10: Overview of the model checking approach (Hensel, et al., 2022)	23
Figure 11: An overview of probabilistic model checking process (Parker, 2011)	24
Figure 12: Overview of Markov models (Katoen, 2016).....	26
Figure 13: MDP model example of a coin flip (Parker, 2011)	27
Figure 14: MDP model of a process of sending a message (Parker, 2011)	27
Figure 15: Probabilistic model in PRISM modelling language (Kwiatkowska, et al., 2009)	28
Figure 16: Rewards specification in PRISM modelling language (Kwiatkowska, et al., 2009) ..	29
Figure 17: PRISM tool display (1) (Kwiatkowska, et al., 2009)	32
Figure 18: PRISM tool display (2) (Kwiatkowska, et al., 2017)	32
Figure 19: Turing Test (Kuipers & Prasad, 2022)	34
Figure 20: Venn diagram of intersection of NLP with various fields (Ranjan, 2020)	36
Figure 21: Overview of the NLP and Deep Learning process (Kaur, 2022)	36
Figure 22: Overview of the NLP pipeline (Loeber, 2020).....	37
Figure 23: Tokenization of the sentence "Natural Language Processing" (Jain, 2021).....	38
Figure 24: Comparison between stemming and lemmatization (Turing, 2022).....	39
Figure 25: Bag-of-Words (BoW) method (Loeber, 2020)	40
Figure 26: Biological Neuron in comparison to Artificial Neural Network (Kalla, 2021)	42
Figure 27: Perceptron unit (Rainergewalt, 2022)	43
Figure 28: Rectified Linear Unit (ReLU) (H. Sultan, et al., 2019)	44
Figure 29: Deep neural network (IBM Cloud Education, 2020)	45
Figure 30: Process of supervised training (Sevarac, 2022)	45
Figure 31: Early stopping method (Nielsen, 2017)	49
Figure 32: Tensors as a generalization of multidimensional arrays (Carlos, 2019)	49
Figure 33: Interaction process between client and server via APIs (Pelayo, 2022)	52
Figure 34: Overview of software of the digital assistant program	53
Figure 35: Technology and Framework used in the program development	55
Figure 36: Process of digital assistant program development	56
Figure 37: Description of properties φ_1 , φ_2 , φ_3 , and φ_4	57
Figure 38: Fragment of the top part of the MDP model	59
Figure 39: Module section representing the processes of the model	60
Figure 40: Reward structure of the model	61
Figure 41: Module section Moe representing the processes	62
Figure 42: Visual representation of module "Moe"	62
Figure 43: Steps to model the algorithms for the development of the digital assistant.....	65

Figure 44: Pseudo-code of creating an intent data	71
Figure 45: NLP pre-processing pipeline	71
Figure 46: Importing the libraries of NLTK and NumPy in Python.....	72
Figure 47: Code sample of tokenization in Python.....	72
Figure 48: Code sample of stemming in Python.....	73
Figure 49: Code sample of Bag-of-Words (BoW) in Python	73
Figure 50: Pseudo-code of pre-processing with NLP	74
Figure 51: Overview of training process with Deep Learning model.....	74
Figure 52: Code for implementation of Deep Learning model on PyTorch.....	75
Figure 53: Code for implementation of Cross-Entropy Loss on PyTorch	76
Figure 54: Code for implementation of Adam Optimizer on PyTorch	77
Figure 55: Loss curve	77
Figure 56: Pseudo-code of implementing the training pipeline.....	78
Figure 57: Importing training data	79
Figure 58: Pseudo-code of uploading MDP file.....	79
Figure 59: Graphical representation of the process computing predictions and data exchange	80
Figure 60: Data exchange between the digital assistant program with the back-end PRISM tool	82
Figure 61: The pseudo-code of implementation of the prediction algorithms (1)	82
Figure 62: The pseudo-code of implementation of the prediction algorithms (2).....	83
Figure 63: The pseudo-code of implementation of the prediction algorithms (3)	84
Figure 64: Code of implementing upload file function in Python	85
Figure 65: Code of implementing upload file function on JavaScript	86
Figure 66: Interaction between different programming languages	86
Figure 67: Interaction between PRISM and Java program via API.....	87
Figure 68: Interaction between Python and Java program via Py4J.....	87
Figure 69: Implementation code of Py4J Python	88
Figure 70: Implementation code of Py4J Java	88
Figure 71: Interaction between Python Flask and JavaScript, CSS, HTML	88
Figure 72: Code of implementing Flask in Python.....	89
Figure 73: Code of fetching data from Python in JavaScript.....	89
Figure 74: Front-end visualization of the digital assistant program	90
Figure 75: Message exchange between user and Cubot of A-SPICE Generic Practices (GP)	91
Figure 76: Message exchange between user and Cubot of A-SPICE Generic Practices (GP) if “Yes” is chosen.....	93
Figure 77: Message exchange between the user and Cubot of A-SPICE Generic Practices (GP) if “No” is chosen.....	93
Figure 78: Message exchange between user and Cubot of probability of the whole process using first formulation variation	94
Figure 79: Message exchange between user and Cubot of probability of the whole process using the second variation.....	95
Figure 80: Message exchange between user and Cubot of probability of the whole process using the third formulation variation	95
Figure 81: Message exchange between user and Cubot of probability of a specific module reaching the final state.....	96
Figure 82: Message exchange between user and Cubot of probability of a variable sod reaching state 0.....	97

Figure 83: Message exchange between user and Cubot of probability of a variable soh reaching state 3.....	98
Figure 84: Message exchange between user and Cubot of minimum days for performing whole process if “one” and “Yes” are selected	99
Figure 85: Message exchange between user and Cubot of minimum days for performing whole process if “two” is selected	100
Figure 86: Message exchange between user and Cubot of estimation of expected minimum days for performing the whole process.....	101
Figure 87: Message exchange between user and Cubot of estimation of expected minimum days for variable sia reaching the final state	102
Figure 88: Message exchange between user and Cubot of estimation of expected minimum days for variable sie reaching the state 3.....	103
Figure 89: Message exchange between user and Cubot of total number of transitions	103
Figure 90: Message exchange between user and Cubot of total number of states.....	104
Figure 91: Message exchange between user and Cubot of value of the initial state	104
Figure 92: Message exchange between user and Cubot of the model construction time	105
Figure 93: Message exchange between user and Cubot of the model checking time	105
Figure 94: Message exchange between user and Cubot of the list of modules names.....	106
Figure 95: Message exchange between user and Cubot of the list of variables names	106

List of Tables

Table 1: Business Process Examples (Comindware, 2019).....	11
Table 2: Hardware specifications and values	54
Table 3: MDP models reward availability, number of total states, number of building time and model construction time.....	58
Table 4: All parameters for the digital assistant program	66
Table 5: Tag, patterns and responses for parameter A-SPICE Generic Practices (GP)	66
Table 6: Tag, patterns and responses for parameter probability of the whole process reaching the final state	67
Table 7: Tag, patterns and responses for parameter probability of a specific module reaching the final state	67
Table 8: Tag, patterns and responses for parameter probability of a specific module reaching a specific state	67
Table 9: Tag, patterns and responses for parameter minimum days for performing the whole process	68
Table 10: Tag, patterns and responses for parameter estimation of expected minimum days for performing the whole process	68
Table 11: Tag, patterns and responses for parameter estimation of expected minimum days a specific module reaching the final state.....	68
Table 12: Tag, patterns and responses for parameter expected minimum days a specific module reaching a specific state	69
Table 13: Tag, patterns and responses for parameter total number of transitions	69
Table 14: Tag, patterns and responses for parameter total number of states	69
Table 15: Tag, patterns and responses for parameter value of initial state	69
Table 16: Tag, patterns and responses for parameter building model time.....	70
Table 17: Tag, patterns and responses for parameter model checking time.....	70
Table 18: Tag, patterns and responses for parameter list of modules names	70
Table 19: Tag, patterns and responses for parameter list of variables names.....	70
Table 20: Hyper-parameter values.....	78
Table 21: All parameters with the associated PRISM class function	81
Table 22: Message input and response prediction of the A-SPICE Generic Practices (GP)	91
Table 23: Message input and response prediction of the A-SPICE Generic Practices (GP) if “Yes” is selected.....	92
Table 24: Message input and response prediction of the A-SPICE Generic Practices (GP) if “No” is selected.....	93
Table 25: Message input and response prediction of first message formulation of probability of the whole process.....	94
Table 26: Message input and response prediction of second formulation variation of probability of the whole process.....	94
Table 27: Message input and response prediction of third message formulation of probability of the whole process.....	95
Table 28: Message input and response prediction of the probability of a variable s2b reaching final state	96
Table 29: Message input and response prediction of the probability of a variable sod reaching state 0.....	97
Table 30: Message input and response prediction of the probability of a variable soh reaching state 3.....	98

Table 31: Message input and response prediction of the minimum days for performing whole process if “one” and “Yes” are selected	99
Table 32: Message input and response prediction of the minimum days for performing whole process if “two” is selected	100
Table 33: Message input and response prediction of the estimation of expected minimum days for performing whole process	101
Table 34: Message input and response prediction of the estimation of expected minimum days variable s1a reaching the final state	101
Table 35: Message input and response prediction of the estimation of expected minimum days variable s1e reaching the state 3.....	102
Table 36: Message input and response prediction of the total number of transitions.....	103
Table 37: Message input and response prediction of the total number of states	104
Table 38: Message input and response prediction of the value of initial state.....	104
Table 39: Message input and response prediction of the model construction time	104
Table 40: Message input and response prediction of the model checking time	105
Table 41: Message input and response prediction of the list of modules names.....	105
Table 42: Message input and response prediction of the list of variables names.....	106