DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

MIGUEL APPLETON FERNANDES
BSc in Computer Science

A SIMULATION FRAMEWORK FOR UML EDUCATION

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

A SIMULATION FRAMEWORK FOR UML EDUCATION

MIGUEL APPLETON FERNANDES

BSc in Computer Science

Adviser: Vasco Amaral
Associate Professor, NOVA University of Lisbon

Examination Committee:

Chair: Armanda Rodrigues
Associate Professor, NOVA University of Lisbon

Rapporteur: Ademar Aguiar
Associate Professor, University of Porto

Adviser: Vasco Amaral
Associate Professor, NOVA University of Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

A Simulation Framework for UML education

Para o Cosme

First of all, I would like to thank my adviser, professor Vasco Amaral, for all the help and guidance that he's provided me with in the course of this thesis, always taking his time to offer me valuable insights into the area (which he knows so well) and into life in general. I also thank him, NOVA School of Science and Technology and the Department of Computer Science for the opportunity to develop this thesis which, not only has taught me a lot, but also will hopefully be able to help others learn more in the future.

I also want to thank my colleagues in the Modelshake project. I want to thank João Carvalho for kindly taking his time to guide me through the beginning of the development of my framework (which oftentimes is the scariest part of a big project like this) and for sharing with me the experience that he acquired through a process of trial and error, much of which he allowed me to skip. I would also like to thank Maria Silva, who helped me in integrating my service into the UML Educational Platform.

I would like to thank Abel Gómez (researcher and professor at the Universitat Oberta de Catalunya, as well as the developer of the CPN Tools Toolkit) for his fast help regarding his plugins and availability to go through my code and provide me with useful suggestions, as well as the effort to make the plugins available to be used as Java libraries (which was previously not possible).

I would also like to thank Dimitrios Kolovos (researcher and professor at the University of York, as well as the lead developer of Epsilon) for his fast and useful help via the Epsilon forum, where he always promptly helps anyone who uses the software and is in need of guidance or assistance.

I would like to send the world's biggest and warmest hug to Bárbara, for all the help that she's given me: technical, emotional, motivational, and much more. Thanks for giving me so much of your love and so much of your time, you genuinely really helped to make this possible.

I want to thank my family for all the love and support. Thanks to my mom and dad and to my sister for the daily patience and for all the laughs, for the talks that never seem to end, for the advice and company and for the hugs. It makes a very big difference. Thanks to my parents and grandparents for making this whole journey possible and for always taking care of me. Thanks to my uncles, aunts and cousins. I really do enjoy being with you all.

I also want to thank my dear friends for the laughs, the long talks, the drinks, the vacations together, the football games, skate hangouts, gaming evenings, music sessions, bike rides, games of chess, dinners, lunches, etc. I really needed all of those.

Although they cannot read, I thank my four-legged friends for always being the happiest every time I opened the door: my black Labrador twin, my crazy chocolate Labrador, my obsessed Boxer and my little Yorkshire.

Abstract

Nowadays, Software Engineering is usually connected to Modelling, both for system prescription and description, and, for most Computer Science students, that means learning UML and using the standard. Learning about models is a great asset for every student of the field, as it allows software development with greater planning and understanding.

UML students, as in other areas, need to receive feedback on their work. Feedback can always be received directly from professors, but it is much more effective if students can also receive it in an automated and ever available manner.

One of the problems in learning UML is that, very often when designing static models, students do not have an in-depth understanding of the implications that may come from the transposition of that model into a dynamic system. This can lead to students creating models with inconsistencies or, for example, concurrency problems such as deadlocks.

In contrast, one of the advantages of UML is that it offers separation of concerns (thanks to its sublanguages working in a relatively independent way), allowing students to understand how to model a software system without needing to understand all of the 14 UML diagrams. As a standard, UML has widespread use, which equates to an abundance of tools that can work with it.

In this thesis, a simulation environment of UML is implemented to support UML courses, integrating existing technology.

There is then benefit in making available for students a framework for automated model simulation that allows them to submit their models and receive feedback on how they work in practice, using state-of-the-art technology to simulate different UML sublanguages.

Keywords: Modelling, UML, Model-Driven Engineering, Model simulation, Model execution, Model transformation, Epsilon, Systematic Literature Review

Resumo

Atualmente, a Engenharia de Software está geralmente em contacto com a Modelação, com o intuito de prescrição e descrição de sistemas. Para a maioria dos estudantes de Engenharia Informática, isso significa aprender UML e utilizar o standard. Aprender a modelar é uma grande mais-valia para os estudantes da área, permitindo-lhes que desenvolvam software de modo mais esclarecido e planeado.

Os estudantes de UML, como noutras áreas, têm a necessidade de receber feedback relativo ao seu trabalho. Embora seja sempre possível recebê-lo diretamente dos professores, é muito mais eficaz que os alunos possam também receber feedback imediato e automatizado.

Um dos problemas dos estudantes de UML é que, muitas vezes, por não possuírem conhecimento aprofundado das implicações da aplicação de um modelo estático num um sistema dinâmico, acabam por criar modelos com inconsistências ou, por exemplo, problemas de concorrência (como deadlocks).

Uma das vantagens da linguagem UML é que possui separação de conceitos (devido ao facto das suas sub-linguagens funcionarem de modo relativamente independente), o que permite aos estudantes ter a perceção de como modelar software sem que seja necessário conhecer todos os 14 diagramas oferecidos pela linguagem. O facto de UML ser um stadard traduz-se também numa abundância de ferramentas que suportam a linguagem.

Nesta dissertação é implementado um ambiente de simulação de UML para suporte a cadeiras de UML, que integra tecnologia existente.

Assim, existe grande benefício em tornar disponível para uso educacional um serviço automatizado de simulação de modelos que permita que os alunos submetam os seus modelos e recebam feedback de como estes funcionam num contexto prático, usando tecnologia de ponta para simular várias das sub-linguagens UML.

Palavras-chave: Modelação, UML, Engenharia Orientada a Modelos, Simulação de modelos, Execução de modelos, Transformação de modelos, Epsilon, Revisão Sistemática da Literatura

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the context and motivation for this work, together with the main research questions and objectives. Then, there is an overview of the contributions from the work that was produced.

1.1 Context

Abstraction is a natural concept for the human mind. Our mind naturally applies cognitive processes to rework the information it receives from reality. We use models since the earliest stages of humanity, to either plan things or to help us reason, to try to better understand something. Models are used across many fields because they are not just a convention, models reflect one of the characteristics which define us most as a species: wanting to know and understand more.

Model-Driven Development (MDD) is a paradigm that uses models as the primary artefact of software development. [11] Model-Driven Architecture (MDA) is an MDD approach to software design and development which provides guidelines for structuring software specifications that are expressed as models. [58] Through the use of platform-independent models, constructed with modelling standards (namely UML [56]), MDA can be applied to a wide array of system types (applications, integrated systems, web services, among many others). These models will therefore capture the functionalities, behaviour and inner-logics of a system independently of the platform-specific code that will finally be used to implement that same system. [58]

UML, or the Unified Modeling Language, is a graphical language for specification (as well as visualisation) of different aspects of software systems, from the behaviour of specific system artefacts, to the system's own abstract concept. [66] UML includes 14 diagrams, aimed at representing different information (which are further detailed in

section 2.4).

1.2 Institutional Context

This thesis will be developed in the context of an ongoing project regarding a UML educational platform called Modelshake, developed by the ASE (Automated Software Engineering) team from NOVA LINCS [52] to support the existing courses in Software Engineering. This platform aggregates modules for evaluation, agile process support, and version control of UML projects.

1.3 Motivation

The importance of teaching UML within the context of Software Engineering (SE) is well known and documented. Engels et al. [31] note that in 2001 the IEEE/ACM curricula for Computing mentioned UML only briefly, whereas in 2004 UML was already placed in the centre of the core course on Software Engineering, something which has not changed as of the last published version of said curricula and its respective Software Engineering volume (SE2014) [6].

The use of tools to aid UML and modelling education has also already been debated [48], with the conclusion that, although informal modelling techniques should also be taken into consideration, the use of formal models (related closer to code) is also very necessary. Furthermore, it is expanded that tools for educational use should have at least some different key characteristics from those to be used in industrial activities. Although usability is cited to be the key aspect most sought after in both fields, in the industry there does not appear to be the need for tools to be quick and efficient for small-scale projects or to provide the user with positive feedback on its progress, something which is referred to as helpful by students.

Some information is available on the importance of integrating simulation into modelling tools targeted at education. For example, Akaiama et al. [3] cite, through their experience with students in Software Engineering courses, that Visual Paradigm's utility would be greatly increased in the context of the courses they teach if it offered any way to simulate models compellingly. Likewise, Westphal (in his 2019 book titled *Teaching Software Modelling in an Undergraduate Introduction to Software Engineering* [93]), states that the use of tools capable of verification and simulation can be of great help in making behavioural models tangible.

Unlike with specifically educational tools, there is a lot of information on UML execution tools generically, whether it is through lists available online that aggregate that information [12, 95], or literature, in the form of many articles on specific tools or even Systematic Literature Reviews on the subject [14].

However, there is still much to be done in terms of documenting and comparing tools that are suited (or recommended) specifically for educational use and whether they are

in use (and with which degree of success). There is the need for analysing what is the available offer in terms of educationally-targeted tools for UML simulation, how these tools work, and how can they be gathered in a multi-purpose service integrated into the context of existing courses that make use of UML and modelling. It is also important to discover what assets can these tools, gathered in a service like the one described, offer not only to the teaching of UML but also to Modelling and Software Engineering education in general. The ultimate purpose of using these tools would be to help the students with useful feedback in a more automated and always available manner.

1.4 Problem Statement

Given the challenges presented in the previous sections, we are faced with the need and ultimate goal to develop a system that integrates state of the art simulation technology and can simulate the behaviour of different aspects and elements of UML models in the various UML sublanguages. Through this, there is the goal to achieve an improvement in the process of learning and understanding the different UML sublanguages.

To understand how to answer these demands, we must first understand how the UML language can be simulated, and what tools and techniques are there for this purpose; if they are open-source; and how to integrate them into an automated process.

1.5 Objectives

To support the need to evaluate, compare (and ultimately integrate into a single service) UML simulation tools that are targeted towards (or at least adapt to) education, we must first gather a selection of these tools.

The goal is to integrate the selected tools into a solution that allows the simulation of UML models in a modular way (i.e. by sublanguage), to facilitate their understanding, as well as to ease the processes of error identification and correction. This solution integrates multiple tools into a service to be used in the context of the existing courses that tackle UML, modelling, and software engineering. This will impose some constraints, especially regarding the technologies to be utilised: preferentially open-source tools, as well as open-source frameworks such as Eclipse.

This service should be able to be easily integrated into other contexts, but it is also important that it works in tandem with the already implemented educational platform (mentioned in 1.2), used in the context of the existing Software Engineering course. The service must transform input models into information that can be provided to the simulation tools, as well as return the output of the chosen tools to the platform responsible for their submission.

1.6 Contributions

An initial contribution is the systematic review that was carried out (according to procedural standards) on UML simulation tools that are used in, proposed for, or recommended for education (section 3.2). This is in itself relevant as it partially tackles the impending need to clarify how the simulation of UML models is currently being used in education.

The main contribution of this work is an education-targeted simulation framework for UML models [33], alongside its design and implementation process. This framework is integrated as a module into an educational platform for UML, with ongoing use in the Software Engineering courses, but can be also integrated in other contexts.

The work developed allows for an easier understanding of the core UML sublanguages by students, especially the behavioural diagrams, making use of simulation to expose problems and errors in the construction of the models, facilitating their correction.

This simulation service gathers software that can simulate certain models independently, and integrates the various tools together. It allows the submission of simulation requests for specific UML models and is able to respond to them with the simulation result. Using exclusively open-source technology, this simulation framework (and the educational platform) can be used in any Modelling or Software Engineering course to aid professors and students with feedback on the inner-logics and construction of UML diagrams.

The subsection below will further expand on these contributions, analysing this work in the light of Design Science [94].

1.6.1 Design Science

Design science is an outcome-focused research methodology. In design science, one iterates over: **1)** designing an artefact targeted at improving something, and **2)** empirically investigating the performance of an artefact in a certain context. Since these artefacts are designed in a context, they should also be studied in that context. "Artefacts" here means methods, techniques, algorithms, etc., related to software and information systems. The context would be the design, development, maintenance, and use of software and information systems. [94]

Hevner et. al [37] give 7 guidelines for research to be coherent with the design science approach. Below is an overview of each of these guidelines and how they relate to the work that was developed and is detailed in this document:

1. **Design as an artefact**: *The end result of design science research should be a purposeful artefact.*

   This guideline ties to what was previously said in this section: that the main contribution of this work is a simulation service for different UML sublanguages that allows the submission of UML models, and then simulates them and returns the corresponding feedback.

2. **Problem relevance**: *The objective of design science is to produce technological solutions to relevant existing problems.*

   This ties closely to what is introduced in sections 1.3 and 1.4 and concluded in 3.2.5: that there is very little information on the use of simulation to aid student understanding of UML and its sublanguages, as well as very little information on the use of simulation tools gathered in a single framework that can be easy and effective for students to use.

3. **Design evaluation**: *There must be reliable evaluation methods to validate the utility, quality and efficacy of an artefact.*

   Table 1.1 shows the Hevner et. al [37] design evaluation methods. The evaluation conducted for the work presented in this document ties closely to the third category presented in the aforementioned table: Experimental. To test the service, a model dataset will be used, representative of the type of models submitted by students. This is the second experimental approach detailed in 1.1. These models are targeted at testing various aspects of the transformations and simulations done by the framework, for each of the simulated diagram types. To showcase this process, multiple case studies were conducted. This is further detailed in chapter 6.

4. **Research contributions**: *Design science must provide clear contributions in the areas of the design artefact, design foundations, and/or design methodologies.*

   This is explored in-depth at the beginning of this section. This work contributes first and foremost with the UML simulation framework that was developed and implemented. Another contribution is the systematic literature review on UML simulation tools in education.

5. **Research rigour**: *Design science depends on rigorous methods in the processes of building and evaluating design artefacts.*

   The processes of developing this work taking design science into account and performing a systematic review to evaluate the state of the art are both targeted at increasing the rigour and objectivity of the work.

6. **Design as a search process**: *Searching for an effective artefact requires the use of available means to satisfy the desired ends while complying with the rules of the problem environment.*

   The systematic review that was performed with the aim at understanding the available means, together with further comparison and analysis of the tools found, allowed the analysis of state of the art technology and understanding of how each tool treats the UML language and its sublanguages. This is further explained in chapter 4. Then, the implementation of the framework (detailed in 5) was able to use the findings from the SLR to achieve a solution that addresses the questions expressed in sections 1.3 and 1.4.

7. **Communication of research**: *Design science research should be presented effectively not only to audiences more related to technology but also to management-oriented audiences.*

   This is something related to this whole dissertation, which includes intensive discussion of the process and end result of the work, showcasing the solution found, as well as describing its implementation process. The aim was to present this work in a way that is compelling and understandable not only to technology-oriented audiences but also to public of the management and business domain.

Table 1.1: Hevner et. al [37] reference table for Design Evaluation Methods.

1. Observational	Case Study: Study artefact in depth in business environment
	Field Study: Monitor use of artefact in multiple projects
2. Analytical	Static Analysis: Examine structure of artefact for static qualities (e.g., complexity)
	Architecture Analysis: Study fit of artefact into technical IS architecture
	Optimisation: Demonstrate inherent optimal properties of artefact or provide optimality bounds on artefact behaviour
	Dynamic Analysis: Study artefact in use for dynamic qualities (e.g., performance)
3. Experimental	Controlled Experiment: Study artefact in controlled environment for qualities (e.g., usability)
	Simulation: Execute artefact with artificial data
4. Testing	Functional (Black Box) Testing: Execute artefact interfaces to discover failures and identify defects
	Structural (White Box) Testing: Perform coverage testing of some metric (e.g., execution paths) in the artefact implementation
5. Descriptive	Informed Argument: Use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artefact's utility
	Scenarios: Construct detailed scenarios around the artefact to demonstrate its utility

1.7 Document Structure

The following chapters are organised as follows:

- Chapter 2 - Background

  This chapter presents further information necessary to give a richer context to the problem in question, also introducing the definition of concepts considered relevant for this work.

- Chapter 3 - Related Work

  In this chapter, an analysis is performed to the existing information on the topic through the elaboration of a Systematic Literature Review on the specific presented subject. Information on a set of tools found through further research is also presented in a synthesised manner.

- Chapter 4 - Solution

This chapter provides a description of the solution developed to tackle the problem stated, as well as a view of the high-level architecture of the service, together with an analysis of the related technologies that were necessary to achieve this solution.

- Chapter 5 - Implementation

  This chapter describes the implementation process of the simulation service, further detailing the structure of the framework, the steps taken and the rationale behind the decisions made.

- Chapter 6 - Design Evaluation

  This chapter details the evaluation process conducted to validate the presented work, which includes a case study for each of the three simulation modules implemented.

- Chapter 7 - Conclusions and Future Work

  This chapter concludes this dissertation, overviewing the work done and its relevant contributions. It also analyses possible complementary future work.

Chapter 2

Background

This chapter takes on the important task to provide information that will allow for a better understanding of not only the problem at stake but also the solution developed to tackle it. The concepts of Model-Driven Engineering, Development, and Architecture will be discussed and explained for context. It is also crucial to offer insight into the structure of the Unified Modeling Language, as well as into various concepts related to modelling and simulation.

2.1 Modelling

A model is an abstraction (a simplified or partial representation) of a system, used to make predictions or inferences. [47] Modelling, in the context of Software Engineering, is creating a representation of the knowledge about a system: its concept, structure, elements, behaviour, etc. Software models are a way of expressing Software design [62], and modelling is an essential part of Software design. Modelling allows the builder to put their theory somewhat in practice with a high risk-return ratio, steering clear of the unnecessary and avoidable problems faced when building without planning. [66]

2.2 MDE and MDD

Model-Driven Engineering (MDE) classifies a set of approaches for the development, based on modelling, of software artefacts. [11] One of these approaches is the Model-Driven Architecture initiative (section 2.3) by the Object Management Group (OMG), supported by other OMG standards that are also a vital part of MDE. Also important to MDE are other concepts such as model transformation (explained in section 2.9). The use of transformation tools allows us to analyse elements and aspects of models to then obtain

artefacts like source code, simulation inputs, XML deployment descriptions, etc. This aids in the consistency between software implementations and the information comprised within the models. [68]

Model-Driven Development (MDD) describes a purely development-related subset of MDE. The term refers to the pattern of using models as the primary artefact of the software development process, which generally means using models for (at least partial) generation of the implementation. [11]

2.3 MDA

Model-Driven Architecture (MDA) is an MDD approach [11] to software design, development and implementation, spearheaded by the OMG [60]. Launched in 2001, it provides guidelines for structuring software specifications that are expressed as models. [58]

MDA values the use of platform-independent models – constructed with modelling standards such as UML (section 2.4) – to capture the behaviour and structure of a software system. These models offer more in terms of independence when compared to the platform-specific code that will implement the system.

MDA is strongly related to other relevant standards, such as the Unified Modeling Language, and can be applied to a wide array of areas and domains.

2.4 UML

The Unified Modeling Language (UML) is a modelling language that was developed in 1994 and 1995 (and additionally in 1996) by Rational Software, with the goal of creating a standard for software design. It was adopted as such in 1997 as it was brought into OMG (the Object Management Group). The Unified Modeling Language User Guide [66] describes UML as "a standard language for writing software blueprints", with the capability to "visualise, specify, construct, and document the artefacts of a software-intensive system."

UML's variety of possibilities allows it to be adequate for modelling a wide array of system types such as information systems used in big companies, but also Web-based applications, and much more. UML can also be applied in a vast array of areas and domains. The different types of diagrams also aid in UML's ability to represent different parts of the system's structure in different degrees of abstraction, as UML can be used to describe the system's concept but can also model specific behaviour related to specific components that are a part of that same system.

The basic components of UML are Diagrams, Relationships and Things. [66] There are four types of things: Structural, Behavioural, Grouping and Annotational. Structural Things are mostly static elements, physical or conceptual: Classes, Interfaces, Use Cases, etc. Behavioural Things represent dynamics and change through space and time: an example are Interactions or Actions. Grouping Things are, for example, Packages, as they

have the task of organisation. Lastly, Annotational Things are explanatory elements, such as a Note.

UML has 14 diagrams, which divide into Structural and Behavioural. The diagrams that represent Structure are the following diagrams: Class, Component, Composite Structure, Deployment, Object, Package, and Profile diagrams. The ones that refer to Behaviour are the Activity, State Machine, Use Case, Communication, Interaction Overview, Sequence, and Timing diagrams, with the last four belonging to a subset designated by Interaction Diagrams. This is further represented in figure 2.1.

It is relevant to do a quick analysis of each diagram to showcase the wide array of applications for the Unified Modeling Language that come from the different characteristics of the various diagrams.

1. Structural Diagrams [83]:

    a) The Class Diagram: It can either represent systems or components, showing their structure as various related elements (Classes and Interfaces) with features, relationship constraints, etc.

    b) The Component Diagram: Represents Components and their dependencies.

    c) The Composite Structure Diagram: It can be used to represent the internal structure of a classifier or the behaviour of a collaboration.

    d) The Deployment Diagram: An architectural view of the system as software artefacts deployed to targets.

    e) The Object Diagram: It works as a Class Diagram for the instance level, representing relationships between Objects.

    f) The Package Diagram: Shows the package structure and relationships between Packages.

    g) The Profile Diagram: Allows adaptation of how UML works, adapting the UML metamodel to new domains or platforms through the definition of custom constraints, etc.

2. Behavioural Diagrams [83]:

    a) The Activity Diagram: Represents sequence and conditions for low-level behaviour.

    b) The State Machine Diagram: Expresses finite-state transitions to describe discrete behaviour.

    c) The Use Case Diagram: Represents actions that a system can perform and their relations with each other and with the external Actors.

    d) The *Interaction Diagrams*:

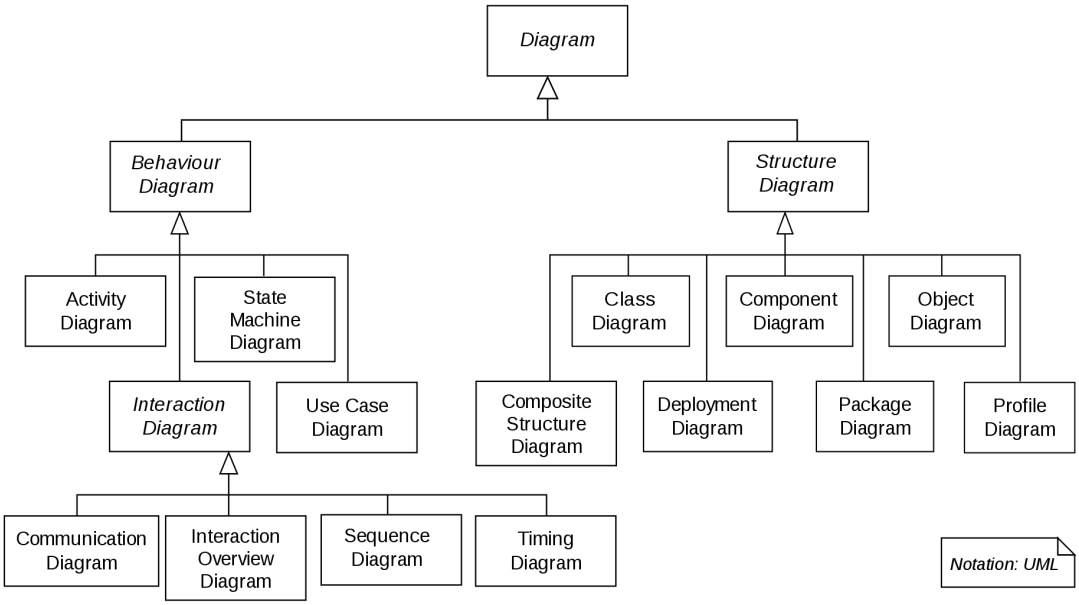


Figure 2.1: The UML diagrams [67].

i. The Communication Diagram: Represents interactions between Lifelines and how the Message being passed relates to the internal structure.

ii. The Interaction Overview Diagram: Works as an Activity Diagram focused on the overview of the flow of control.

iii. The Sequence Diagram: Represents the Message interchange between Lifelines.

iv. The Timing Diagram: Represents Interaction when the focal point of it is Time and its impact on factors like conditions.

2.5 UML in education

As referred in Chapter 1, between 2001 and 2004, there was a big shift in the perceived importance of UML in the Software Engineering community, reflected in the IEEE/ACM joint curricula for Computing (in the Software Engineering volume). There, UML progressed from briefly referenced to centrally important [31]. We still get that same impression when analysing the Software Engineering volume of the last version published of the IEEE/ACM Computing curricula (SE2014) [6].

Tools are used in UML education because they are a necessary complement to traditional informal techniques of modelling as they offer an approach closer and more connected to code. Despite this, it should not come as a surprise that the simple importation into education of tools fit for industry and used in that context can be ineffective. Liebel

et al. [48] refer in their 2017 study the need for some specific characteristics in education-related UML tools. The main characteristics they refer are quickness and efficiency for smaller projects and the need for positive feedback (related to motivation).

2.6 Simulation

Simulation refers to mimicking or imitating the operation or execution of a process, system, component, etc. [8, 70] Simulation can be applied to multiple areas and have multiple purposes, such as quality assessment, performance optimisation, education, and many others.

2.7 Model execution and simulation

Within Computer Science, simulation can be applied to a wide array of subdomains, such as Software Engineering, and more specifically to Modelling. Models are abstractions, and therefore the simulation of models can work as the operation of a less concrete version of what is being modelled. As previously stated, models offer a way of testing something without having to build it. Therefore, simulation can be the perfect complement to it, allowing for the testing of an imitation of what is being ultimately implemented, offering more detailed testing with virtually no additional cost, if this simulation can be generated or performed simply through the use of the already existing models.

There is a lot of information on the execution or simulation of UML models. A very relevant reference for the subject is the 2018 Ciccozi et al. Systematic Literature Review (SLR) on the matter [14], which not only gives great insight into the variety of tools that exist for this purpose but also gives some interesting context to the problem and the area of study itself.

The review classifies model execution [1] as either Interpretative or Translational. Using the interpretative approach, the model would be interpreted directly by a (language-specific) program, which would execute the model on the target platform, whereas in the translational approach, the end goal is to obtain object code from the model and, after that, to execute that code on the target platform.

This review also notes that, although UML was widely accepted as standard from early on, until UML version 2.0 (2005), the language "was still considered ambiguous and partially inconsistent, especially in terms of execution semantics". This is because UML 2.0 brought the standardisation of fUML (Foundational Subset For Executable UML

[1] The Ciccozi et al. SLR on the subject prefers the use of the term **model execution**, as they already use the term **model simulation** for the specific case of the "execution of a model in an environment (e.g., an IDE) that is different from the ultimately intended target environment." As this notion is not necessary in the context of this document, and because both terms were found during research to be used generically regarding "running" a model, both terms will be used interchangeably in that context (however with more focus on the keyword **simulation**).

Models) [55], alongside the Alf language (Action Language for Foundational UML), finally offering precise execution semantics, although only regarding composite structures, classes, activities and state machines.

The Alf Specification [2] states the following: "The UML Superstructure specification defines the standard graphical and textual notations used to express a UML model. In this context, Alf can be used as an alternative textual notation to represent portions of the overall model."

2.8 Model checking

Model checking is the process of automatically verifying finite-state systems. [15] It is used to determine if a system satisfies some specification. Through the use of algorithms, a system model expressed in some precise language can be analysed taking into account an also precisely formulated specification. Model checking can be used to detect structural and logical problems in software models.

2.9 Metamodels, model transformation and modelling tools

A metamodel is a model of a model [79] or, more precisely, it is the model of a modelling language [69]. Therefore, metamodelling encapsulates, in simple terms, the analysis and building of the rules for modelling a set of situations (whose modelling context has something in common).

In OMG's Model-Driven Architecture approach, all modelling languages are ultimately defined in the OMG standard metamodelling language, called the Meta-Object Facility (MOF). [32]

In general, model transformation is the process of converting a model type or instance into another model. Regarding MDA, a model transformation is generally the process of converting a Platform-Independent Model into a Platform-Specific Model, using additional information and transformation rules. [72] A model transformation defines the model type for its input and output through the definition of the metamodels to which the input and output models must conform.

The MOF standard [59] is an indispensable cog in the MDA mechanism, offering common knowledge and a gathering point for tools, easing tool interoperability. [32] Tools are a big part of Model-Driven Architecture. It is unfeasible that a single tool can congregate every desirable functionality related to MDA, so these functionalities are distributed between the many tools available. That means that most likely there will be separate tools for metamodel analysis, model transformation, model simulation, compilation, etc. These tools can be used together in the MDA environment with aid of the XML based serialisation format XMI.

XMI (XML Metadata Interchange) [57] is another OMG standard, created for metadata information exchange via XML (if this metadata's metamodel can be expressed in MOF). One of XMI's most common uses is regarding UML models.

Chapter 3

Related Work

To produce a thorough and precise analysis of the state of the art, a Systematic Literature Review (SLR) was conducted on UML simulation tools that are used in (or proposed for) an educational context. First, it is relevant to clarify the concept of a Systematic Literature Review. After that, the Review itself will be presented. At the end of the chapter, some known tools for UML simulation will be described in synthesis.

3.1 Systematic Literature Reviews

A Systematic Review is a review that follows a concise analytical process that can be reproducible. The idea is for it to fetch data related to a certain field or domain (it can be something specific, but also refer to broader areas of study related to specific problems) and then select and synthesise that information, with guidance from the prior specification of research questions.

Bolderston [10], in a 2008 paper about conducting effective literature reviews, states that "a literature review can be an informative, critical, and useful synthesis of a particular topic. It can identify what is known (and unknown) in the subject area, identify areas of controversy or debate, and help formulate questions that need further research." The author justifies the value in an SLR by stating that "creating a systematic review allows for increased power and precision in estimating effects and risks. In addition, the systematic review is an invaluable practice tool. Large quantities of information can be evaluated and synthesised into a shorter document."

The studies that contribute to a systematic review are called primary studies, while an SLR is a form of secondary study.

The process of writing an SLR follows specific guidelines to make it easier to reproduce it (to test its validity) and to provide coherency in approaches between different

documents. These guidelines can also be adapted to the area to which they apply: in this case, Software Engineering. The review that follows will be guided by the Guidelines for performing Systematic Literature Reviews in Software Engineering [46].

The following quote appears in the introductory chapters of that document, reinforcing the importance of systematic reviews, in general, but also in our specific area:

"Systematic literature reviews in all disciplines allow us to stand on the shoulders of giants and in computing allow us to get off each others' feet."

3.2 A Systematic Literature Review on UML simulation tools in education

3.2.1 Review motivations

UML education is a very important part of teaching modelling and Software Engineering (SE). One way of proving this is through its relevance in the ACM/IEEE joint curricula for Computing, where UML is at the centre of its SE course. [6, 31]

The use of tools in the context of UML-modelling education is an important complement to informal modelling, as it is an approach with a closer relation to the to-be-implemented code, which can be beneficial to students. The characteristics of a tool adequate for education are not entirely similar to those of a tool targeted at the business and industry domain. The students not only do not have some needs present in the industry but also have additional needs, like positive feedback, easy learning, and efficiency in small-scale projects. [48]

There is some information available on the importance of UML-simulation tools in the context of teaching UML and modelling, namely through the experience of teachers that express the value it can add to UML-related tools used by students. [3, 93]

There is also a lot of gathered information on UML simulation or execution tools [12, 14, 95], although it is very hard to understand through that same information whether these tools can be used for educational proposes, if their main purpose is related to the academic world, or if they have been used in that context.

This review aims at understanding which UML simulation tools are there for educational use. This means that it is necessary to search for a few different things. Firstly, there is a need to search for tools that were built for educational use, even if they have not been used yet. Despite this, the tool must be implemented, so the search will not focus on proposed implementations or concepts for this kind of tool. Secondly, there is the need to search for tools that are already in use educationally, even if these tools are traditionally used in industrial activities, as there might be some features of a tool that can make it appropriate to be used in teaching. There is the additional need to search for tools that are recommended for educational use, as there can already be some form of analysis of

an industry-targeted tool concerning its educational potential, even without it having yet been put into practice in that domain.

Lastly, and before initiating the review methodology, it is important to clarify that simulation in this context means any form of animation or execution of the model ("running" the model), although this review will also consider tools that allow the generation of executable code as capable of UML-simulation (even if that code has to be executed outside of the tool), as they provide the desired capability to simulate models.

3.2.2 Research method

This review takes into consideration the Guidelines for performing Systematic Literature Reviews in Software Engineering [46]. As such, this subsection will approach the different steps described in the aforementioned document.

3.2.2.1 Research questions

The Guidelines state that "the aim of a systematic review is to find as many primary studies relating to the research question as possible using an unbiased search strategy. The rigour of the search process is one factor that distinguishes systematic reviews from traditional reviews." This emphasises the importance of defining the appropriate research questions as the first step.

Taking into account the information already stated, the following research question and sub-questions were formulated:

> **RQ1. Are there tools for simulation of UML models that are used, recommended or proposed for education?**
>
> RQ1.1. Which tools?
> *rq1.1.1. What type are these tools? (runtime, visual aid, textual, etc.)*
> *rq1.1.2. For which UML sublanguages are these tools? (if not all)*
> *rq1.1.3. How are these tools being used? (through web services, standalone tools, plugins, etc.)*
> *rq1.1.4. Do these tools have an educational objective or were they re-purposed for that matter?*
>
> RQ1.2. Is there feedback on the advantages of using these tools?
> *rq1.2.1. Does the utilisation of such tools suggest improvements?*
>
> RQ1.3. At what education levels is this approach being used? (basic, introductory, advanced)

These questions will allow us to understand not only how UML simulation tools are directed towards (or at least appropriate for) education, but also how these tools work, and whether there is excess or deficit in the types of approaches that the tools implement. The aim of these questions is also to cover other matters, like providing validation for the UML-simulation approach in modelling education.

The research questions were formulated with guidance from the approach proposed by the aforementioned guidelines, taking into account the following PICOC criteria [46]:

- *Population*: UML simulation tools.
- *Intervention*: Suggested, recommended, or ongoing use in education;
- *Comparison*: UML sublanguages simulated; Tool type (regarding the simulation type and also the type of use or access).
- *Outcome*: Success in educational use (pedagogical factors).
- *Context*: Articles and research papers.

3.2.2.2 Search Process

The first part of this step is the initial, wider search. Initially, the idea was to perform this search in the IEEE Xplore Digital Libray [40], the ACM Digital Library [1], SpringerLink [78], and Elsevier's ScienceDirect [30]. However, after starting the process of composing the search strings, there was a lot of difficulty in maintaining the same degree of specificity from one library to another. To maintain coherency, it was decided that this review would not include SpringerLink, as the inability to browse the articles by their abstract would make the degree of specification too disparate through different sources.

The chosen strings (which can be found in annexe II) firstly searched, in the articles' abstracts, for "UML". They also looked in the abstracts for "education" (as well as variants of the word, synonyms, and related words) and "simulation" (again, also variants, synonyms, and other related words). Lastly, the full text was searched for expressions that would connect the terms related to UML (or models) to the ones related to simulation or execution. Some examples of expressions like these are: "UML simulation", "animation of models", "generate code", etc. Some search strings are not as complete as others, because some of the libraries' search engines had some limitations and restrictions in terms of search terms.

The next steps are collecting the articles retrieved from the submission of the search strings to the search engines, as well as filtering results that do not fit the definition of research articles, and then merging the results from the different libraries and finally removing duplicates.

The last step is to apply the selection and exclusion criteria presented next (3.2.2.3) to the remaining documents. After a final selection of studies is obtained, the process of snowballing may be performed, searching the bibliography of the selected studies for

other possibly relevant studies that may cover information not already available in the already gathered studies.

3.2.2.3 Quality Assessment

This step focuses on the selection and exclusion criteria for the papers that were retrieved from the initial search. For these articles, the following inclusion criteria were used:

I1: Studies about the use of one or more UML simulation tools in education.

I2: Studies about one or more UML simulation tools that are recommended or proposed for education.

I3: Peer-reviewed studies (research articles, conference papers, book chapters)

The following exclusion criteria were also defined:

E1: Studies that do not refer to any tool.

E2: Studies that do not refer to a tool's capability to simulate UML, i.e., the ability to animate or execute a UML model in some way or produce executable code.

E3: Studies that do not refer to any educational character for the tool's use (ongoing, proposed, intended, etc.).

E4: Studies not in English.

The tools mentioned by these articles will also be referred to in our analysis. However, no formal criteria will be defined regarding this tool selection, as the judgement on the capabilities of these tools will come mostly just from the information provided by the primary studies themselves, alongside only a brief complementary online research about each of the tools that showed up. This means that the concrete criteria regarding inclusion or exclusion of the tools are already implicit in the article selection criteria stated above. If necessary, further tool comparison and analysis can be performed in the future.

3.2.2.4 Data Extraction and Synthesis

From the selected studies, the following information was extracted:

1. For the articles:
   a) The study type.
   b) The type of character of the educational use of the tool(s) in question: Ongoing; Past; Proposed; Recommended.
   c) The feedback obtained on the use of simulation tools in UML education.
   d) The academic level in which the tools are used or proposed/recommended for.

2. For the tools:

    a) The type of access/use of the tool: Standalone Application; Plugin for another tool; Web Service.

    b) The type(s) of simulation the tool can perform.

    c) The UML sublanguages that the tool can simulate.

    d) The original purpose of the tool: Education, Industry, or Inconclusive.

Regarding a tool not yet in use educationally but mentioned for such, if it was created for educational use, the use will be referred to as Proposed, and if its original purpose is not education, the use will be referred to as Recommended. If a tool is created for education, but no information whatsoever about its ongoing use, that use will be referred to as Proposed instead of Ongoing.

To synthesise the extracted information, this review proposes a Descriptive synthesis [46], presenting this information tabulated in a way that: eases the process of answering the research questions; showcases the similarities and differences between both the selected set of studies and the UML simulation tools retrieved; relates studies with the respective tools; relates the studies with the quality criteria (namely **I1** and **I2**).

3.2.3 Results

The aim of this subsection is, as mentioned, to present the extracted information tabulated according to the process of Descriptive synthesis.

These tables take into account the information gathered from the final result of all the filtering: 7 studies (presented in annexe I), selected from the original 79[1]. These were the studies that passed the exclusion criteria, matched the inclusion criteria and provided the necessary information to properly analyse the matter in question and the research questions proposed.

One iteration of snowballing was performed but, due to the small number of finally selected studies (some with very scarce bibliographies), there were no findings in terms of new studies which referred to an educational context for UML simulation tools that were not already in the existing list.

Table 3.1: Types of educational use across studies.

Educational use	Studies
Ongoing	S3, S5, S7
Proposed	S1, S2
Recommended	S4, S6

[1] This number refers to the collection of studies gathered through the use of the search strings shown in annex II to the corresponding databases as of 30/Dec/2020. This number discounts duplicates

The first two tables (3.1 and 3.2) allow us to further analyse the content of the selected primary studies.

Table 3.1 relates somewhat closely to table 3.4 because, for the studies that do not report on ongoing use, their classification for the type or character of the educational use relates directly to the "Original purpose" value associated with those studies. This means that, for the tools whose original purpose was education, the corresponding articles refer either to a "proposed" or "ongoing" use of those tools. As for the tools related to the industry, their use is, then, either "recommended" or "ongoing".

The exception to this is regarding the tools whose original purpose could not be determined ("inconclusive"). This happened with ArgoUML, although it was not a problem as the tool only appeared in a single article that included another tool originally targeted at industrial activity. Therefore, that study (S4) was considered to refer to a recommendation instead of a proposition.

The results regarding table 3.1 were balanced and, regarding the understanding of which type of use is predominant, rather inconclusive (by force of the small sample size). As referred in the introductory chapter (1), early research revealed the existence of very little information when transitioning from educational UML tools to specifically simulation-related tools, and also when transitioning from UML simulation tools to their use or recommendation regarding the educational field in specific. Despite this, this first two tables (3.1 and 3.2) can help us tackle the main research question (**RQ1**), as there are in fact some tools being used, proposed and recommended for education.

Table 3.2 also clarifies RQ1.2 (and subquestions) and RQ1.3. The main conclusion about the first is that feedback, when present, tended to be positive and to suggest improvement in the students' ability to understand modelling. There is, despite this, a downside: with a small sample (even smaller regarding reliable feedback) and no external studies cited, the conclusions that can be drawn from the information have a considerable risk of bias.

Regarding RQ1.3, the results unveiled the absence of tools targeted at a more advanced level of expertise, which was not surprising, as these tools are referred to in the context of education, and so their level should be somewhat accessible to students, as one of the goals of these tools is to ease the understanding of modelling, not other aspects that may be valued by the industry, like effectiveness, robustness, etc. Despite this, it is relatively hard to correctly evaluate and compare the tools' required knowledge level without further testing and experimenting with the tools. The information compiled regarding the tools' target academic level is therefore obtained mostly directly from the authors of the primary studies.

Regarding its "Quality criteria" column, table 3.2 also relates to table 3.1, as the studies needed to report on "ongoing" tool-use to comply with quality criterion **I1**. Unsurprisingly, the studies that reported on the ongoing use of UML simulation tools also made some type of recommendation or proposition of the tool to justify its use (hence why every study that complied with **I1** also met criterion **I2**).

Table 3.2: Study type and feedback; relation to quality criteria.

Study	Type	Quality Criteria	Feedback	Academic level
S1	Conference Paper	I2, I3	N/A	Final year bachelor students (Intermediate)
S2	Research Article	I2, I3	N/A	Introduction to Programming course (Introductory)
S3	Conference Paper	I1, I2, I3	Student satisfaction survey conducted: Students state that it helped them learning class diagrams and would use it again	Second year course (Intermediate)
S4	Conference Paper	I2, I3	Inconclusive: Student evaluation surveys were conducted but only about UML tools in general, not targeting simulation related tools	Undergraduate students (Intermediate)
S5	Research Article	I1, I2, I3	Study with students revealed positive impact in understanding of system behaviour	Master level course (Intermediate)
S6	Article in Proceedings	I2, I3	N/A	Introductory
S7	Conference Paper	I1, I2, I3	N/A	Introductory

Tables 3.3 and 3.4 will help us respond to research question RQ1.1 and its subquestions. Regarding *rq1.1.1*, the results were very one-sided. As presented in 3.3, seven out of the eight tools allowed for the generation of executable code, while two executed the models with visual aid. The generated code was predominately Java and C variants, but there were other options. Despite the small sample size, we can draw from this information the conclusion that it is rarer that the tools perform the simulation themselves, which leads to the abundance of tools that can translate the model into code that can be executed easily, as the use of Java and C variants is very widespread.

Regarding *rq1.1.2*, we should look at the table at the end of this subsection (table 3.5), which shows the different tools and how they relate to which UML diagram(s) – i.e. which UML sublanguages each tool can simulate. While analysing this table, we encounter two main realities: we find a few tools (ArgoUML and Willert UML Studio) that can simulate a very wide array of the UML diagrams; and then the remaining majority of the tools (apart from IBM Rhapsody), which use both (or a combination of) some Structural Diagrams (Class, Component, Composite Structure) and some Behavioural Diagrams (State-machines, Sequence diagrams, Activity diagrams) for simulation. This hints at a relation to the structure of object-oriented code, as these tools seem to use Structural and Behavioural diagrams to represent class structure and method behaviour, respectively.

In relation to subquestion *rq1.1.3*, we should again analyse table 3.3. We find a

Table 3.3: Tool information and relation to the studies.

Tool	Tool Type	Simulation Type	Studies
ArgoUML [7]	Standalone app.	Generation of C++, C#, Java, PHP 4, PHP 5 and Ruby code	S4
BridgePoint [96]	Standalone app.	Generation of xtUML code that can be translated into Platform-specific code	S1
FIRP tool[2]	Standalone app.n	Generating a interactive prototype from a conceptual model (in the application)	S5
IBM Rhapsody [38]	Standalone app.	Allows viewing animated diagrams	S7
RAPTOR [65]	Standalone app.	Visually execute models on the interface and Java code generation	S2
UMLAnT[2]	Eclipse plugin	Test execution with reports and animation of Sequence and Object diagrams	S6
Umple [86]	Eclipse plugin; Web service; Standalone app.	Generation of Java, Ruby, C++ and PHP code	S3, S4
Willert UML Studio [73]	Standalone app.	Generation of C and C++ optimized code	S7

blatant trend in the use of tools as standalone applications, with the sole exceptions being UMLAnT and Umple. The presence of Eclipse here is not surprising, as the IDE already offers many plugins and possibilities regarding UML and modelling.

Lastly, to respond to *rq1.1.4*, we must look at table 3.4. As stated before, this table is closely related to table 3.1 and the conclusions drawn from it are similarly inconclusive, also by force of the small sample size. Despite this, it is slightly surprising to see so many re-purposed tools from the industrial domain, especially taking into consideration that some authors [48] considered the importance of certain tool characteristics for education specifically (that were generally not sought after by the industry).

Table 3.4: The original purpose of the tools retrieved.

Original purpose	Tools	Studies
Education	BridgePoint, FIRP tool, RAPTOR	S1, S2, S5
Industry	IBM Rhapsody, UMLAnT, Umple, Willert UML Studio	S3, S4, S6, S7
Inconclusive	ArgoUML	S4

In a concluding note, there was relevant information available about UML-simulation tools in education, and some tools were retrieved from the primary studies obtained from the search process. There were some factors, like the original purpose of the tools (whether they were made for education or re-purposed from the industry), for which trends could not be identified, making it harder to sketch out theories from that information. However, some trends were identified, mainly in how these tools used models

[2]The tool does not seem to be available online

to provide simulation or generate code (using mainly diagrams such as class, activity, state machine and sequence diagrams), and how these tools could be used (mainly as standalone applications). Unfortunately, the feedback about the benefits of the studied approach ended up being scarce and rather inconclusive, which is not ideal.

Table 3.5: The tools retrieved in relation to the UML sublanguages.

Type	Diagram	Tool							
		ArgoUML	BridgePoint	FIRP tool	IBM Rhap-sody	RAPTOR	UMLAnT	Umple	Willert UML Studio
Structure	Class	✗	✗	✗		✗	✗	✗	✗
	Component	✗	✗						✗
	Composite Structure							✗	✗
	Deployment	✗							✗
	Object	✗							✗
	Package								
	Profile								
Behaviour	Activity	✗			✗	✗	✗		✗
	State Machine	✗	✗	✗	✗			✗	✗
	Use Case	✗							✗
	Communication	✗							
	Interaction Overview								✗
	Sequence	✗			✗		✗		✗
	Timing								

3.2.4 Identified limitations of the review

The main two types of limitations that can be encountered when performing a process like this relate to the primary studies and then the review itself. Regarding the process of obtaining the final set of studies, it is important to first discuss publication bias. This describes a situation "when the publication of research results depends on the nature and direction of the results. Because of publication bias, the results of published studies may be systematically different from those of unpublished studies" [74]. This means that, as the outcome of research can influence publication, there is the risk that publication is biased towards certain outcomes.

There are many possible limitations and threats to validity involved in the review process, but also many more that could have existed and were mitigated by the use of concise analytical procedures.

The first possible biases that can be easily identified are: firstly, bias related to the place of publication, and secondly bias related to database querying. Regarding the digital libraries, the ones chosen cover a lot of ground in terms of content, and with cross-referencing between libraries it is easier to collect information from many sources. Still, it is plausible that some types of study may be in deficit or excess in a certain database, affecting the preciseness of the data collection. Regarding the process of querying these

databases, there were limitations forcefully imposed by the allowed query structures that did not permit for the search strings to be exactly equivalent between databases.

The usage of search strings as closely alike as possible is a step towards relieving some bias. Regarding the search string choice, it is important to note that even if adjusting the specificity of results can be improved through various iterations and experimentation, it is ultimately very hard to reach a perfect balance between a search string covering enough ground and also avoiding irrelevant results.

Another bias related to the search for studies is language bias. Although these searches were performed through the use of the English language, which is widely accepted as global or universal, there are naturally still studies conducted in other languages that may be of importance and that could be overlooked.

Regarding selection bias, although it can always be relevant, there was an effort to minimise it through the use of inclusion and exclusion criteria. The information to be extracted was predefined as so to try to reduce extraction bias and to keep a balance between extracted information from one study to another.

3.2.5 Conclusions and future work

Although there is still much to be walked, this review takes a step forward towards having a better understanding of how UML-simulation tools take part in the educational world.

Through the presented process, 7 studies were selected from an initial 79. These studies provided us with the empirical evidence that was then extracted and synthesised to ease its analysis.

This review offers some knowledge that can be of use not only to other researchers who wish to further expand on this subject but also for practitioners in search of understanding on the matter. The first conclusion that can be drawn from this review is that there isn't much information on this specific issue, or at least that there is not much information on it that can be gathered without struggling with finding too much unrelated literature. It is possible that a bigger team of researchers with more time on their hands can obtain a better result in terms of gathered studies, especially starting with a bigger (and thus probably richer) selection.

The information gathered identifies another difficulty regarding the matter at hand and the problem presented: **there is a substantial amount of industry-targeted tools being used in education**. As expressed before, some authors [48] underline the need to have specific educationally targeted tools, as the industrial and educational sectors do not have the same characteristic-wise needs. The fact that tools with an originally industrial purpose are being used in education means that something is lacking. Further research is needed to address questions such as: Is this happening because there is not enough demand for specifically educational UML-simulation tools? Is this happening because the existing educationally targeted tools are not effective? Are industrial tools more suited to education than one might think?

Something else that can be further expanded is the validation of the simulation tool approach to UML education. Although the necessity for this approach is known [3, 93], this review has not found enough feedback or cited studies on the matter. This leaves room to either expand the search for this feedback or even for researchers to conduct their own studies to allow the gathering of this missing feedback.

This review offers good insight into how these tools are mainly used and how they work for the most part. It is interesting to notice how these tools incline to the generation of code that can itself be executed. This is possibly because it may reduce the complexity of the tool, but still allows for easy simulation as the code is mainly generated in commonly used languages. These tools also seem to be used in most types of UML-related courses, ranging from first-year bachelor introductory courses to master-level education.

For practitioners, this review offers specifications on a set of tools that are used in education or proposed for that use. It offers limited but useful knowledge on how these tools can be accessed, how much of UML they cover and what output they can ultimately produce.

This review offers guidance to researchers which may take on the task of expanding the knowledge regarding this problem, showcasing which trends can be easily spotted and which have to be further investigated to obtain a more complete understanding of the domain. Besides from expanding this review, further work also involves updating this review, as it can quickly become outdated due to the rapid evolution experienced in the software engineering realm.

3.3 Tools for UML simulation

As mentioned before, there are UML simulation tools for specific sublanguages and some of them can potentially be adequate for use within the context of education. So, there is certainly the need to consider some of these tools for further comparison, to be performed in the future. This is especially true taking into account that the information and tools retrieved from the SLR were scarce and did not cover the UML sublanguages widely. It is important to emphasise that the fact that these tools are not specifically targeted at education might mean that they are unavailable for that purpose (some of these tools might not be open-source technology). The tools that will be referred ahead were found through other forms of manual research and snowballing, and through prior knowledge and also the use of online lists [12, 95] and the Ciccozi et al. SLR [14].

The following table (3.6) quickly describes some tools for the stated purpose of simulating UML models, retrieved by methods other than the SLR (detailed in 3.2).

[3]The tool's website is down

Table 3.6: Other UML simulation tools.

Tool	Tool Type	Simulation Type	UML sublanguages (diagrams)
Cameo [19]	Standalone app.	Activity execution (OMG fUML standard), State machine execution (W3C SCXML standard), running use case classifier behaviour	Activity, State machine, Sequence, Use case
IBM Software Architect [39]	Standalone app.	Java code generation from AUL	State machine, Sequence, Activity
Altova UModel [4]	Standalone app.	Java, C++, C#, and Visual Basic code generation	Class, sequence and state machine diagrams
Moka [29]	Papyrus module	Implementation of fUML 1.4, PSCS 1.2 and PSSM 1.0 standards	Any UML model conforming to the syntactic subset of the union of the 3 standards
Papyrus-RT [28]	Software based on Eclipse	Code generation from UMLRT	All UMLRT diagrams
Quantum Leaps QM [64]	Standalone app.	C and C++ code generation	State machine (hierarchical)
Reactive blocks [3]	Standalone app.	Activity diagram deadlock checking	Activity
Sinelabore [71]	Standalone app.	Generate C, C++, Python, C#, Lua and Swift code from UML	State machine, Activity
Sparx Enterprise Architect [76]	Standalone app.	Execution of executable state machines and dynamic analysis of activity and sequence diagrams	State machine, Activity, Sequence
Syntony [92]	Standalone app. based on Eclipse	Generation of discrete-event simulations from UML	Class, Composite Structure, State machine, Activity
UMLAUT [41]	Standalone app.	Conversion from UML models to executable UML models	Class, State machine, Deployment
UML-based Specification Environment [75]	Standalone app.	OCL interpreter that allows simulation and validation	Class, Object, State machine, Sequence
Yakindu [43]	Standalone app.	Java, C and C++ code generation and also execution in simulation engine	Finite state charts

Chapter 4

Solution

It is now important to provide an overview of the framework that was developed and how it relates to the already existing educational structure (in the context of which it will be put into use). It is equally important to describe the architecture of the framework in question through high-level design.

4.1 Description

As mentioned before, the framework that was created is a UML simulation service that is also integrated into an existing educational platform (even if it can also be integrated into other platforms). The goal of this service is for students to be able to submit models and receive feedback through simulation, making use of model transformations and code generation, as well as other modelling techniques, using additional materials to be provided by the teachers (if necessary, to give some sort of semantical context for these models to be simulated).

This section will describe the service and what are its core functionalities, without going too deep into the implementation process (that is instead done in chapter 5). The next section will provide a view of a high-level architecture for the service, describing its main characteristics and explaining how it relates to the open-source tools utilised and to the platform into which it will be integrated. The tools and technologies to be used in the process of building the service will be detailed ahead (in section 4.3).

Figure 4.1 is a use case model of the service, presenting how the system users relate to the main use cases of the system. As the figure shows, the goal is to have a service that allows students to submit different types of models and receive feedback, obtained through simulation and supported by additional elements provided by the professors when needed.

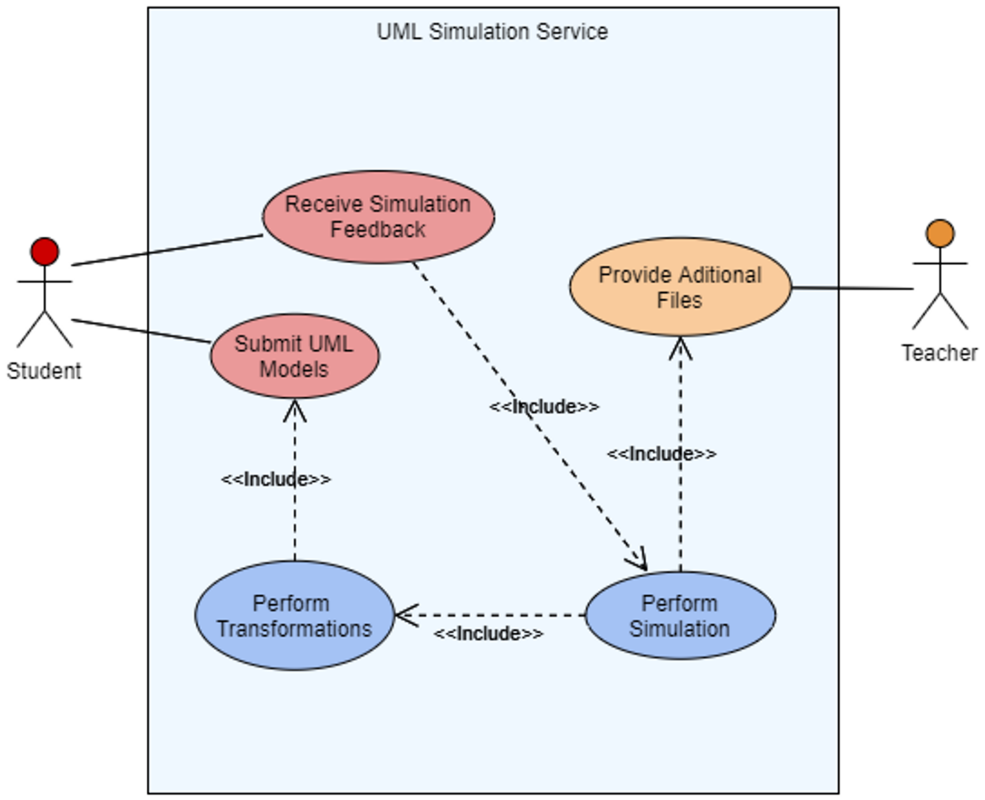


Figure 4.1: Use case diagram representing an overview of the service's functionalities

4.2 Architecture

This section presents a form of high-level architecture for the framework (figure 4.2), regarding how the service interacts with the modules that simulate different UML sub-languages and how it relates to the main platform and, through that, with the users. This section also delves into which tools will be used for simulation in the different modules. It also provides a superficial overview of the input model transformations necessary to provide the chosen tools with the information needed to perform the simulation.

As mentioned before, there is an already existent **UML Education Platform**, used in the context of the Software Engineering course, whose end goal is to provide students and professors with services like project management and monitoring, automatic grading, version control, and, now, simulation. The users (represented by the **Teacher and Student Sessions**) will interact with this platform to access and use the various services it comprises.

Therefore, the Education Platform (or any other that wishes to integrate the framework) has to interact with the **Simulation Service**, to make it available and usable by its users. Like stated in the previous section, there are three main use cases (shown in 4.1),

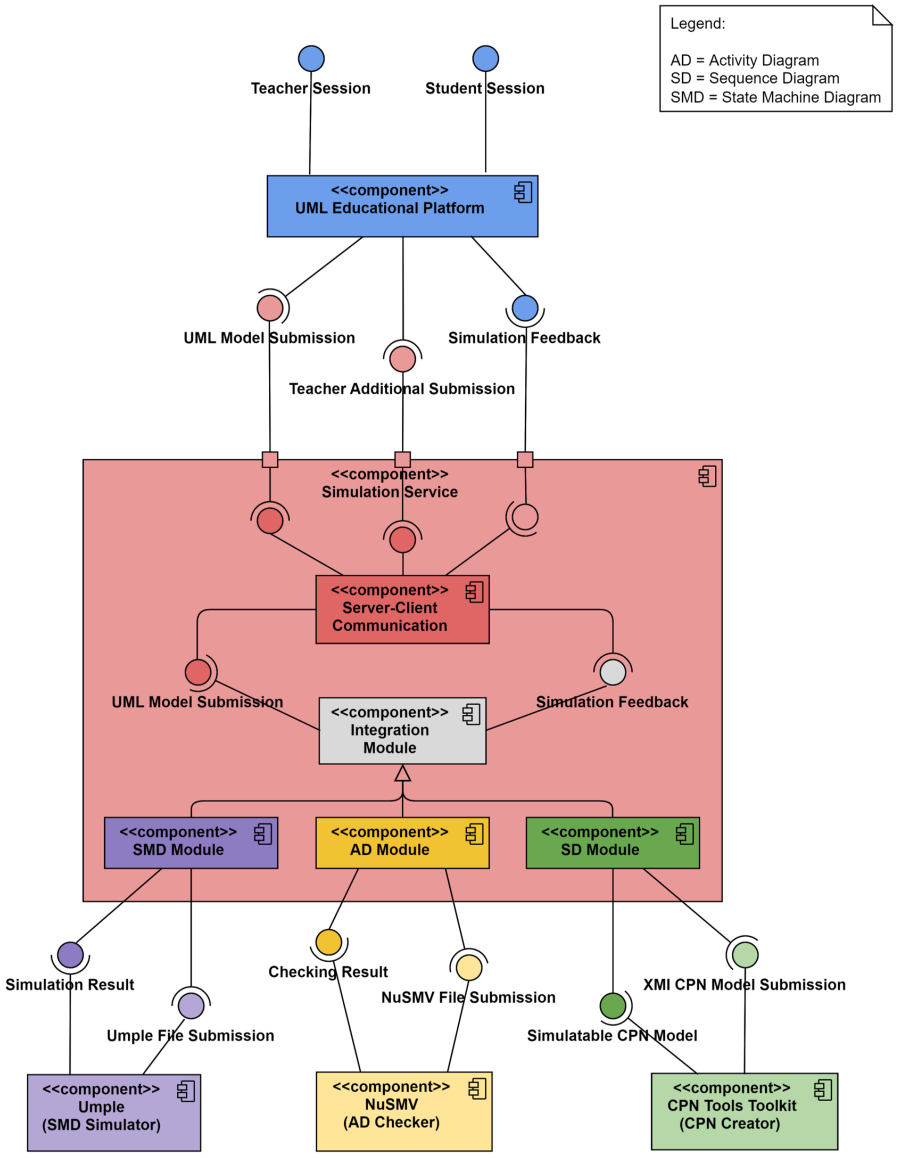


Figure 4.2: Component diagram representing an architecture for the framework: the simulation service in relation to its modules and to the platform responsible for submitting information and receiving feedback.

represented in figure 4.2 by the three interfaces between the Platform and the Simulation Service: **Model Submission**, **Teacher Submission** and **Feedback**. This service makes use of various tools for distinct UML sublanguages, which work in different ways and use different elements: **Umple**, **NuSMV**, and the **CPN Tools Toolkit**. UML models do not have concrete execution semantics, so there is the need to deal with the input model and transform it, mostly through model transformations and code generation, to compose an input for its respective simulator. The main way to do this is by using the open-source Epsilon tools for Eclipse, something which is described further in the next section (4.3). This whole process is represented in figure 4.2 through the **Integration Modules**. Although these modules do more than simulation, to make it easier to understand some aspects of the implementation of the framework, these are also sometimes referred to in this document as *Simulation Modules*. In figure 4.2, the **Server-Client Communication** component represents the part of the Simulation Service which is simultaneously responsible for communicating with the submission platform and the Integration/Simulation Modules.

The UML sublanguages chosen for simulation were selected out of the ones most relevant to the context of the courses related to UML and modelling (e.g. activity, class, component, sequence, state machine, and use case diagrams). Out of these, three Behavioural Diagrams (activity, sequence and state machine) were chosen for simulation, as the Structural Diagrams (class and component), as well as the use case diagram, severely lack semantics and context for simulation.

Therefore, the Service will include a module for each of the chosen diagrams: **activity diagram**, **sequence diagram**, and **state machine diagram**. These behavioural diagrams are widely used in the context of Modelling and Software Engineering classes and their simulation can provide help to the students in the process of understanding how these models work and how they can help while designing and building a system. [93]

Because the different UML sublanguages work differently and have different simulatable elements, there is the need to perform distinct processes for each diagram type. The different external tools (represented in the image below the Service) represent external software that can simulate information extracted from the UML sublanguages. The simulation service interacts with these components, providing information obtained through modification of (or generated from) the models submitted by students, together (in the case of state machine diagrams) with additional elements provided by the teachers. The service then supports the clean-up of the received output of the simulator, composing a simulation report with the aim of transmitting concise and helpful information to the platform responsible for the model submission, for the users to access it and obtain aid in understanding logical and technical aspects of the construction of a UML diagram.

There was therefore the need to choose the tools suited for the simulation of each of the selected UML sublanguages. Firstly, upon comparison, the following subset of tools was chosen from the initial pool of tools (reported in 3.2 and 3.3): Moka [29], RAPTOR [65], UMLAUT [41], Umple [86], and USE [75]. This selection focused only on

open-source tools and there was also the intention to avoid tools that required dynamic model visualisation, as the intention is to provide textual feedback instead of animation or dynamic visualisation of model execution.

Apart from simulation tools, one other way of giving feedback on models is through model checking. Model checking consists of checking if a model meets the specification of a system [9]. In Software Engineering, the specification of a software system may contain rules to avoid crashes, deadlocks, etc. For this purpose, some known tools are NuSMV [53] and Uppaal [88]. This technique is also useful in the case that simulation software is unavailable or insufficient for a specific UML diagram.

The tools mentioned above are mainly standalone applications, but there is no reason to discount web services as candidates for the simulators. Umple, for example, has an online platform [84] besides the standalone application. Various factors will be weighted in the decision of the appropriate tools, such as easy integration, efficacy and usefulness of the information that the simulator outputs.

Further comparison of tools and the final choice for each of the simulated UML sublanguages were made iteratively, as each of the simulators was implemented. The selection was ultimately the following:

- State Machine diagram module: **Umple**. This was the easiest tool to access. It was open-source and could be accessed through a *.jar* file that could be executed with parameters given a *.ump* file. This *.ump* file could be generated from a state machine diagram using Epsilon's code generation tools.

- Activity diagram module: **NuSMV**. This model checker was chosen due to the difficulties presented by the available activity diagram simulation tools (namely Moka) in simulating simple models. Moka required the target activity diagram for simulation to describe extensively and formally the code it was supposed to represent, making use of complex UML elements that steered away from the basic elements that students are taught to use. Therefore, NuSMV was chosen to provide feedback on the construction and inner logic of simpler models. Following a process described by Ul Muram et al. [80] it is possible (using Epsilon's code generation tools) to transform the information contained in an activity diagram into a *.smv* file that can be checked by the tool.

- Sequence diagram module: **CPN Tools** [17]. The tools shortlisted for the simulation of sequence diagrams revealed to be ineffective for reasons related to integration, as well as their nature. RAPTOR, for example, had no way of communicating with external applications or of receiving UML models. The selected approach was to use the process described in [18]. This document details a transformation from UML sequence diagrams into Petri net models which can be serialised using the CPN Tools Toolkit [36] and simulated using CPN Tools (which is a tool for editing, simulating, and analyzing Colored Petri nets). This can be done from a

Java application through various open-source plugins and libraries available online, with aid from the Epsilon's model transformation tools.

4.3 Tools and technologies for solution implementation and integration

Regarding the tools and technologies that were used in the development of the service, there was the decision to use technologies that are not too foreign to the ones already used in the remaining services of the platform and in the Software Engineering course. This implied the use of open-source technology related to UML and modelling, such as Epsilon for Eclipse and its various languages.

4.3.1 Eclipse

Eclipse is a community-powered free and open-source integrated development environment (IDE). [22] Eclipse is a programming environment known for its extensibility and easy use of plug-ins. Its primary use is for Java application development, but it supports a lot of other programming languages.

4.3.2 Spring and Spring Boot

Spring is a framework for programming and configuring Java-based applications on any kind of deployment platform. [77] Spring Boot facilitates the creation of stand-alone Spring based Applications that are easily runnable. It can be used to build simple and effective REST APIs.

4.3.3 Maven

Apache Maven is a software project management and comprehension tool that can be used for building and managing any Java-based project. Based on the project object model (POM) concept, Maven can manage a project's build, reporting, and documentation, from a central piece of information. [49, 50]

4.3.4 Epsilon

Epsilon is a family of Java-based languages targeted at scripting automated tasks regarding model-based software engineering. [21] These tasks include code generation, model transformation and model validation. Epsilon can work together with many types of models, including UML models, and also includes editors and debuggers based on Eclipse. The various Epsilon languages were a very important part of the development of this framework. The most relevant to this work were the following:

- **EOL**: The Epsilon Object Language (EOL) is the core expression language of Epsilon. [27] It is the language foundation for the other task-specific languages for tasks such as model transformation (ETL), code generation from models (EGL) or model validation (EVL). EOL can be used as a model management language regarding tasks that do not fall into the patterns targeted by the task-specific languages mentioned. EOL can be used to define operations related to how certain tasks work. It is used within the context of the integrators, together with the other Epsilon languages.
- **ETL**: The Epsilon Transformation Language (ETL) allows the transformation of an input model into a model of a specific output format (if necessary, in different modelling languages and technologies) in a rule-based and modular manner. [25] ETL is used in the developed service in the context of the integrators, to change the models received from the submission platform into other models that can be used by the simulators for simulation.
- **EGL**: The Epsilon Generation Language (EGL) is a model-to-text transformation language that can be used to transform models into various textual artefacts, like executable code. [20] This can also be useful to generate information that can be input into a simulator.
- **EVL**: The Epsilon Validation Language (EVL) is used to specify and evaluate constraints on models (of various modelling languages and technologies) and, then, produce error messages and executable fixes. [26] It is used in the context of the proposed service to validate the models before simulation.

4.3.5 EMF

The Eclipse Modeling Framework (EMF) is an Eclipse-based framework for modelling and code generation that is used to create applications based on a structured data model. [23] It includes a metamodel (Ecore) that allows the specification of other models. EMF uses model specifications described in XMI and provides the capability to generate Java classes from them. It also supports a vast amount of other tools and plugins that are built upon it.

4.3.6 Papyrus

Papyrus is an open-source UML modelling tool based on Eclipse and based on various standards, especially by OMG [61]. It is a domain-specific tool that has a customisable UML profile and that enables model-based techniques. [24]

4.3.7 Ant

Apache Ant is a Java library and command-line tool. It drives processes described in build files (written in XML) as targets and extension points with dependencies. [5] Ant supplies

several built-in tasks allowing to compile, assemble, test and run Java applications, and can be used to pilot any type of process which can be described in terms of targets and tasks.

4.3.8 REST

REST (representational state transfer) is an architectural style used in the context of the World Wide Web. [34] It is widely used and accepted in software architecture for the process of developing web APIs. REST APIs usually access information making use of HTTP methods with URL-encoded parameters, generally making use of XML or JSON in the process of exchanging data.

4.3.9 JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format based on a subset of JavaScript. [45] It is a language-independent text format, but uses conventions that are familiar to most programmers. JSON is built on an ordered list of values (such as an array or vector) and on a collection of name/value pairs (like a hash table).

Chapter 5

Implementation

This chapter details the process of implementing the architecture described in the previous chapter: the solutions developed, the decisions made, and the inner structure of the service constructed. Firstly, there is an analysis of the service itself: the structure responsible for aggregating the simulation modules and communicating with the platform that requests the model simulation. That is followed by individual descriptions of the implementation process of each of the three modules that are comprised within the framework, targeted at simulation of the three UML diagrams chosen: activity, sequence and state machine.

5.1 The Simulation Service

This section describes the part of the Simulation Service that communicates with the individual modules that perform the transformation and simulation of each of the diagrams, as well as with the UML Educational Platform.

5.1.1 The UML Education Platform

As described in the previous chapter, and shown in figures 4.1 and 4.2, the simulation framework must interact in various ways with the educational platform.

The UML Education Platform communicates with the Simulation Service through REST requests, which provide (via JSON) information like the type of file being sent, and a key for accessing the desired file through Amazon Web Services' database. The service then responds to the Platform's requests with JSON objects containing the simulation feedback. So the Service works as a server that receives client requests from the Educational Platform.

```
final AmazonS3 s3 = AmazonS3ClientBuilder.standard().withRegion(Regions.EU_WEST_2).build();
String bucketName = "myawsbucketupload";

S3Object o = s3.getObject(bucketName, key);
S3ObjectInputStream s3is = o.getObjectContent();
FileOutputStream fos = new FileOutputStream(path);

PrintWriter writer = new PrintWriter(path);
writer.print("");
writer.close();

byte[] readBuf = new byte[1024];
int readLen;
while ((readLen = s3is.read(readBuf)) > 0) {
    fos.write(readBuf, 0, readLen);
}
s3is.close();
fos.close();
```

Figure 5.1: The function that retrieves models from the AWS database, used in the context of the Education Platform's Evaluation Service.

This process can be easily mimicked by other platforms that wish to use the framework. As the service only needs to receive REST requests and for the models to be stored in the AWS database, the access to the framework can even be replicated by manually using AWS and using software such as Postman [63] to build and send requests to the service.

5.1.2 Service implementation

The service itself is a Spring Boot Maven (and therefore Java) application, created with the aid of the Spring Initializr. The project was developed using Eclipse. Its main classes constitute the server portion of a REST API, targeted at receiving client requests from the Platform and responding with the simulation feedback. The Service provides two endpoints: */submit* – for the teacher to submit additional files – and */simulate* – for the student to request the simulation of a specific model.

When using the */submit* request, the service firstly proceeds to call an operation to access the Amazon Web Services' database and retrieve the desired file through the key received in the client's simulation request. After this, based on the "artifact" parameter contained in the received JSON object (which corresponds to the diagram type), the file will be placed in a specific folder with a specific name, for it to be used in simulation in the future.

For the */simulate* request, the service must again call the operation that fetches the desired model from the AWS database, before choosing with which module it will interact. This, again, is done through the "artifact" parameter received through the Platform's JSON request, which details the type of the diagram received, and therefore which module will process the request. This also determines in which folder the model will be placed before the simulation (the State Machine, Sequence, or Activity folder).

The operation that obtains the models from the AWS database uses a "key" parameter

(to fetch the correct model), which is also provided via JSON by the Platform. This Java method had already been written for the UML Education Platform's Evaluation Service, which was developed by a colleague who is also working with the Modelshake project. Therefore, it was also used in the context of this Simulation Service. The method is presented in figure 5.1.

To request the simulation of a model to the Service, one would have to use a JSON object similar to the following:

```
{
"artifact": "sequence",
"project":
    {
    "id":"project1",
    "key":"DENUpKmBGk622Y3s"
    }
}
```

The "id" parameter is required by the Education Platform (as it is necessary for the reply) but is not used in any task by the Service. Therefore, if the service is used with another platform, the value of "id" is not important. As stated before, the "artifact" parameter is used to specify the simulator which should be used (or, similarly, the diagram which must be simulated) and can be one of three values: "activity", "sequence", or "state". The "key" parameter equates to the model's key in the AWS database, where the model must be stored.

5.2 The Simulation Modules

The architectural concept of modules represents, in practical terms, different functions. These methods, when called, make use of the model (placed in a specific folder with a specific name), alongside other files and libraries, to achieve the transformation and simulation of that model. These operations will then return the result of this simulation to the platform responsible for the model submission.

Some aspects that refer to all modules will be discussed in the present section, while the module-specific implementation will be individually described in each of the following sections (5.3, 5.4, and 5.5).

A visual representation of the service's architecture (explained throughout this chapter) is shown in figure 5.2. This diagram describes how the different tools and technologies are integrated into the implementation (and dataflow) of the service.

5.2.1 Epsilon and EMF tools

The process of validating the models and transforming them into the input format of the simulators is done mostly using the Epsilon languages (namely EVL, EGL and ETL).

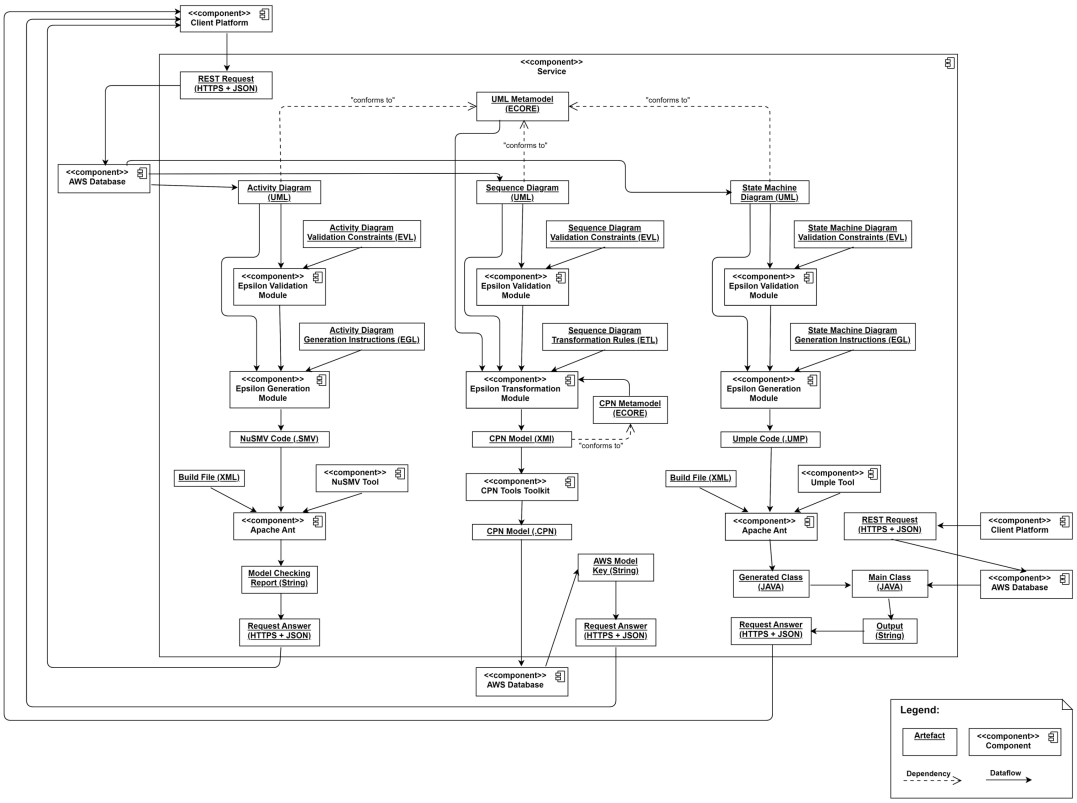


Figure 5.2: High-level architecture of the Simulation Service: the dataflow between its components.

Epsilon is available to be used from Java applications using the Epsilon libraries, which are available online in the *.jar* format, but that can also be accessed from Maven projects using the according dependencies, as these libraries are available in the Maven Central repository [51].

Epsilon, together with EMF, firstly allows model loading, which is necessary so that the models can be used by the different *EOL Module* classes. These are classes provided by the Epsilon libraries that use models and Epsilon files to execute the desired processes.

For example, the *EVL Module* class can parse an EVL file and then receive a loaded model and validate it according to the constraints specified in the EVL file, outputting the errors if there were any.

The *EGL Module* class parses an EGL file and then receives a model, outputting the code generated from it according to the instructions specified in the file.

The *ETL Module* class receives two previously loaded models (together with their metamodel, if it needs to be specified), and converts the source model into the target format, according to the target model's metamodel and taking into account the transformation rules specified in the ETL file.

An example of the model loading process, together with the execution of the validation of that model using an EVL file (and the *EVL Module*), is shown in the code snippet

below.

```
        UmlModel model = new UmlModel();
        model.setModelFile("UmlModel.uml");
        model.load();

        EvlModule module = new EvlModule();
        module.getContext().getModelRepository().addModel(model);

File fileEvl = new File("EvlFile.evl");
        module.parse(fileEvl);

        module.execute();

        Collection<UnsatisfiedConstraint> unsatisfiedList =
        module.getContext().getUnsatisfiedConstraints();
```

5.2.2 Model validation

Before discussing the implementation process of the simulation modules, it is important to refer to the process that occurs just before that. As said above, a model first has to be retrieved from the database and placed in a specific folder, according to its diagram type. But before performing the operations needed to transform the model into the input format of the simulator, there should first be an analysis of the model structure and construction.

UML is very vast and complex and, for that reason, the models must conform to a tighter subset for the service to be able to process them and simulate them. Each of the following sections will discuss the subset chosen and how the compliance to it was guaranteed by the use of model validation (using Epsilon's model validation language - EVL). Besides subset compliance, model validation is also used to guarantee that the model is well built and has no construction errors that would make it unfit for processing and simulation.

For each diagram type, a different EVL file was written to check the elements of that specific UML sublanguage, as well as to comply with the functionalities and elements supported by the transformation processes and by the simulation engines. The different validations will be explored in further detail in the following sections.

In the EVL file, various constraints will be defined according to the construction rules settled and taking into account the UML subset with which a model must comply. Using these constraints will allow checking the model elements and sending messages if the constraint is not met.

In the code snippet below, an example of a constraint (written in the EVL language) for a transition in a state machine diagram is shown:

```
context Transition{
        constraint HasValidTarget{
                guard : self.target.isDefined()
                check : self.target.type().name.toString
                            .equals("State") or
                        self.target.type().name.toString
                            .equals("Pseudostate")
                message : 'The source of a transition
                           must be a state or pseudostate'
        }
}
```

The *constraint* shown above is used to verify transitions (which is why the *context* of the constraint is the Transition element). The constraint is used to *check* whether the target of that transition (if it exists, hence the *guard*) is a State or Pseudostate. In the case that the constraint is not guaranteed, a *message* will be produced.

5.3 The State Machine Module

The state machine simulation module receives a state machine diagram and uses it to generate a representation of it in Umple code, which can be used to generate a Java class for that state machine, allowing it to be simulated, if provided instructions. This section first explores the state machine UML sublanguage itself, then further explains the Umple tool, and finally describes the whole transformation and simulation process.

5.3.1 The state machine diagram

The UML Reference Manual [66] states that a state machine serves the purpose of modelling the possible life histories of an object of a class. It contains states connected by transitions. Each state corresponds to a period of time in that object's lifetime during which it satisfies certain conditions. Events will then trigger transitions, which themselves can have certain effects.

States are a type of node, which can have *entry*, *do* and *exit* activities, which are actions that will be performed when entering the state, while in the state, or when exiting it (respectively). A transition contains a target state and source state, as well as a trigger event, defining when it happens. It can also contain a guard, detailing some condition that must be met for the transition to happen, and may also have an effect: an action that is a consequence of the transition being triggered. State machines can also contain nested

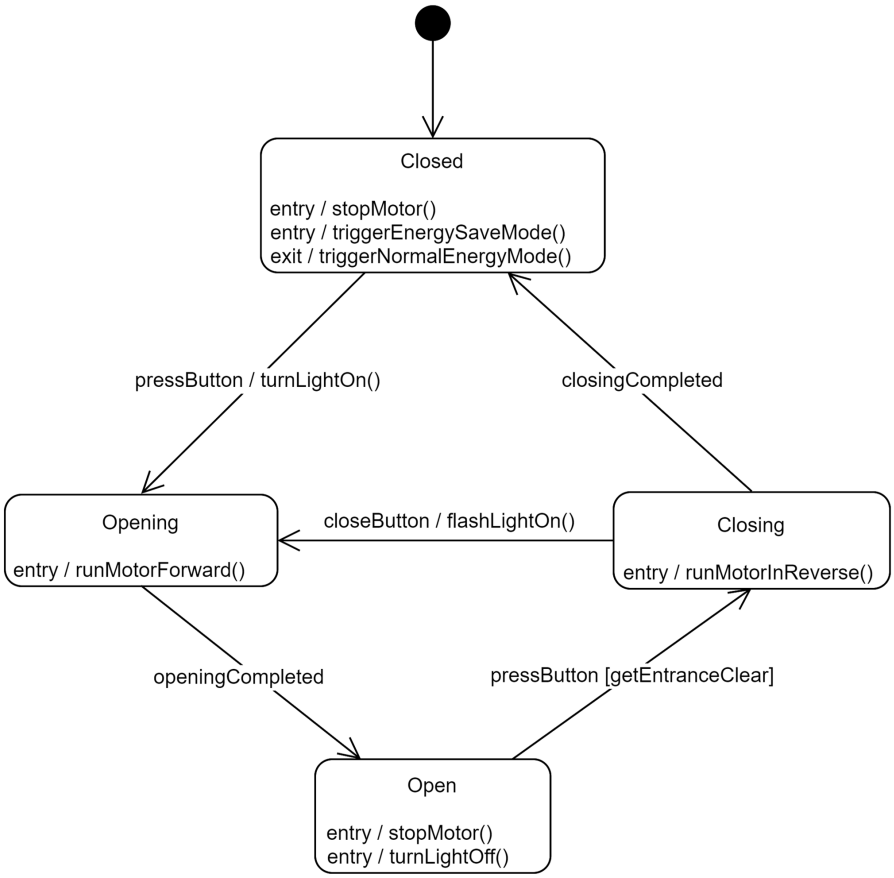


Figure 5.3: Example of a simple state machine diagram (inspired by the Umple examples [85]) representing a garage door.

states (or hierarchical state machines). This means that normal states can contain other substates inside of them, working as a sort of state machine themselves.

A state machine can also contain other special states, called pseudostates. These range through a wide variety, including history states, which are used to recover the state of a state machine when it was interrupted. Another pseudostate is the initial state, which corresponds to the first state in the chain of states and transitions. Some other pseudostates are the Junction and Choice pseudostates, used, respectively, to create decisions between transitions and to chain transitions together. A state machine can also contain concurrent regions, where multiple sequences of transitions can happen simultaneously. States may also contain entry and exit points, which, respectively, allow entering and exiting sub-state machines at a certain midpoint.

Figure 5.3 shows an example of a simple state machine containing some of the most commonly used elements, such as states, transitions, activities, triggers, guards and effects. An initial state is also required.

5.3.2 Umple in more detail

As mentioned in chapter 4, the tool finally chosen for state machine simulation was Umple. As described by Umple's website [86], Umple is a modelling tool and programming language family to enable Model-Oriented Programming. It adds abstractions derived from UML to object-oriented programming languages such as Java.

Umple can be accessed in a variety of ways [87]: it offers some plugins, an online application (UmpleOnline [84]) and a command-line based compiler. The last option was chosen due to its ability to be used from an Ant build (XML) file, which allows the Umple tool to be used together with other processes, in sequence (this will be further detailed below).

Umple can, given a textual representation of a state machine diagram, generate from it a Java class that allows simulation. Therefore, there is the need to transform the received UML model into the textual *.ump* format.

5.3.3 Subset compliance and validation of model construction

The state machine diagram is already a subset of the UML language, but to make sure that the generated Umple code can be processed by the Umple tool, it is important to make sure it complies with Umple's implied state machine subset. Umple accepts a wide array of UML state machine diagram elements. It supports sub-state machines, as well as states (containing entry, do, and exit activities), transitions (containing trigger events, guards, and effects), and pseudostates such as the initial state, as well as the history states (deep and shallow) and the final state (which is a subtype of state).

Figure 5.7 shows the metamodel of the subset of UML that is supported by the module.

The compliance to the subset can for the most part be done in the following stage – Umple code generation (subsection 5.3.4) – because the Epsilon Generation Language can access only the desired element types. However, it is still necessary to use the Epsilon Validation Language to guarantee that the generated code will be processable by Umple. This includes having names defined for elements such as the state machine itself, states and pseudostates, as well as effects, trigger events, guards and activities. In the case of a transition, it needs to have a state or pseudostate as its target and source. Furthermore, an initial state cannot have an incoming transition and a final state cannot have an outgoing transition.

5.3.4 Generating the Umple code

The majority of the implementation of the solution focused on this phase. After the state machine diagram's construction is validated, the model will be used to generate the Umple code.

Umple's textual format follows a hierarchical structure, starting with the main state machine and going down into its states and sub-states. For each state there is a portion

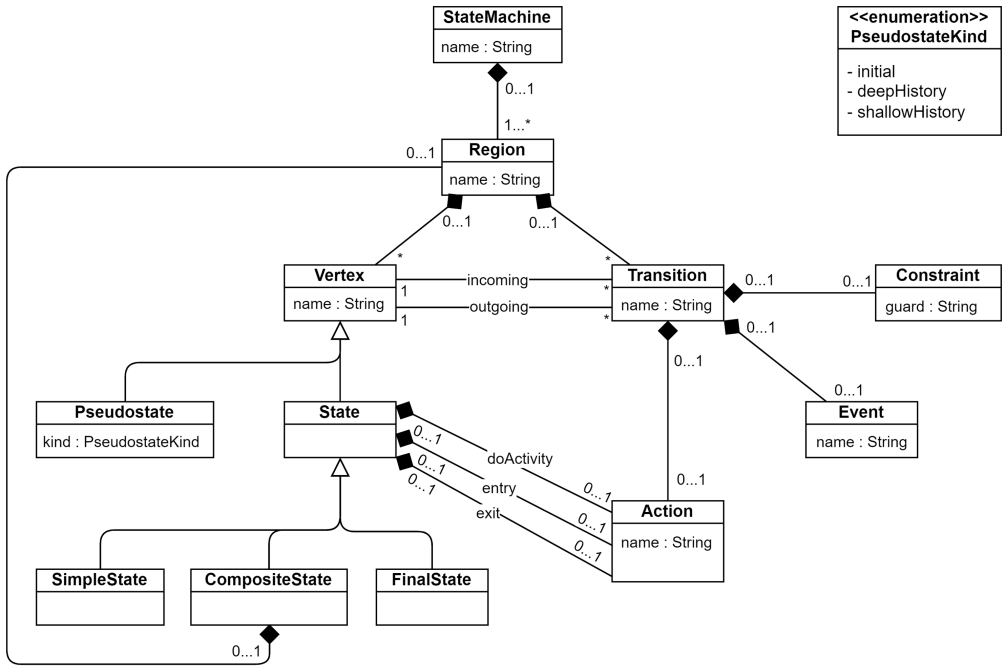


Figure 5.4: The metamodel of the supported subset of UML and state machine diagram.

of code detailing its outgoing transitions (and corresponding target state), as well as the entry, do, and exit activities. Therefore, transitions are detailed (by their trigger) in the code portion describing their source state. The transition's guards and effects are also stated there, with guards being written within square brackets and effects being preceded by a forward slash. State activities are preceded by the *entry/*, *do/*, and *exit/* keywords.

History states are described by adding *.H* (for shallow history) or *.HStar* (for deep history) to the end of a state when it is the target of a transition. The initial state is simply assumed by Umple to transition into the first state presented in the code (in textual order). Final states are described by the "Final" keyword.

Therefore, the EGL file that was written, detailing the generation of Umple code, focuses on generating portions of code for every state contained directly inside the state machine.

These portions (similar to the ones shown in figure 5.5) are composed using the information provided by each state element in the UML model. The state's activities (entry, do and exit) are added to the code with the matching keyword, as well as all of the state's outgoing transitions. As said, transitions are represented by their trigger, and point into the state which, in the model, is the target of that transition. If that state is a history or final pseudostate, the according keyword is added. The guards and effects retrieved from the transition UML-element are also added to the code.

For the hierarchical part, recursion is used. So, if a state has sub-states, the same

```
class Garage {
  Boolean entranceClear=true;
  GarageDoor {
    Closed {
      entry/{stopMotor();}
      entry/{triggerEnergySaveMode();}
      exit/ {triggerNormalEnergyMode();}
      pressButton -> /{turnLightOn();} Opening;
    }
    Opening {
      entry/{runMotorForward();}
      openingCompleted -> Open;
    }
    Open {
      entry/{stopMotor();}
      entry/turnLightOff();}
      pressButton [getEntranceClear()] -> Closing;
    }
    Closing {
      entry/{runMotorInReverse();}
      closingCompleted -> Closed;
      pressButton -> /{flashLightOn();} Opening;
    }
  }

  boolean stopMotor() {
     System.out.println("Stopping motor");
    return true;
  }

  .
  .
  .

  boolean runMotorInReverse() {
     System.out.println(
       "Running motor in reverse");
    return true;
  }
```

Figure 5.5: Umple code for the simple state machine diagram shown in 5.3 (coming originally from the Umple examples [85]).

operation that was applied to every state contained directly inside the state machine is also applied to every sub-state contained directly inside that state.

Figure 5.5 is an example of the Umple code for the garage door case (shown in 5.3). In this figure (below the code portions for each state and sub-state) there is an excerpt of the code portions necessary for the definition of the operations present in the guards, effects and activities. This is because, for the generated Java code to be runnable, Umple needs the transition effects and state activities to be valid code, and the guards to be able to be read as boolean expressions. This is the biggest nuisance about Umple and means that additional information is needed (more than what is provided by a regular UML model). Therefore, variables need to be declared and operations must be defined if they are referred to in a guard, effect, or activity (i.e. in the first part of the Umple code).

The workaround for this is that the student must add this information (declare variables and define operations) in the UML model by attaching a Comment element with this information to the top State Machine element.

This is a bit harder for the students, but it's easier if they use simple code lines for the

transition effects and state activities, instead of operations (which then need to be defined below). A more robust example of this is shown in the case study described in section 6.2.

Therefore, besides the first portions of code (for the state hierarchy), the EGL must also add this information (which, as mentioned, is expressed through a UML Comment element) to the Umple code.

Through these processes, Epsilon's generation engine can generate the Umple code equivalent to a UML state machine and can also output it to the correct Umple (*.ump*) file.

5.3.5 Generating Java code using the Umple tool

As stated before, the approach chosen for running the Umple tool was to use it through the command line. The Umple Tools section of the Umple user manual [87] states that this is typically done from an Ant script, running the Umple jar like follows: *"java -jar umple.jar *.ump"*, where **.ump* will be the file containing the generated Umple code. Using the *-path* option, an output file can also be specified.

To run the Ant *build.xml* file from the Service's Java application, there is the need to recur to a Java class called ProjectHelper, provided by the Apache Ant Java API. It allows to find and run an Ant file if given its path.

The Ant build file which is run by the ProjectHelper class contains instructions to run Umple, as stated before, but also to run the Java code (that was previously generated by Umple) in that same execution sequence. That part will be detailed in the next subsection.

The Java code generated by Umple is a class that represents a state machine and which has functions for each of the events that trigger transitions. For example, using the garage door case, if there is a trigger "pressButton" from the Closed state to the Opening state, the Java class will have the public method *pressButton()* to allow the user to control the flow of the diagram and the state changes.

Therefore, taking as an example the garage door case, the student would have to attach a comment to the state machine in the UML diagram with the definition of all of the functions used in the guards, activities and effects: stopMotor(), turnLightOn(), etc.

5.3.6 Running the generated Java code

After Umple generates the Java class, this class needs instructions for the simulation to occur. As every model will be different, there is no way to use the generated class simply from the Service's code.

Because this simulation service is targeted at Software Modelling courses, which will include assignments that most likely will require the student to construct models following specific guidelines, the workaround to this problem is to have the teachers provide a Main class that will call the methods from the generated Java class.

What this means is that if a teacher assigns the student to construct a model representing a garage door, whose state changes are triggered by specific events (pressButton,

openingCompleted, etc.), the students should include these transition triggers in their models, for the generated Java code to contain the according methods. Therefore, because they assigned the project and defined the desired triggers, the teachers can write a simple *Main* Java class that calls these generated methods.

This simple class can just create an object of the generated Java class, and call the available functions, which match the transition triggers present in the UML model. This means that if a transition in the model has the trigger *t1*, then the generated Java class will have the public method *t1()*.

If the operations defined in the Umple code produce any kind of output, the Main class should also output that; if the Umple code, for example, just changes the value of variables, the Main class can also print the changing values of those variables. For example, if trigger *t1* causes the effect *x=3*, the Main class will have available the method *t1()* and, because it contains a variable called *x*, it will also have a method called *getX()*, to retrieve the variable's value;

If any variables were declared in the Umple code, they can now be initialised when constructing the object of the generated Java class. So, in the case that *x* is the only variable declared, the constructor for the generated Java class would receive as a parameter the desired initial value of *x*.

Therefore, the Main class for this simple case would be something like:

```
public class Main {
        public static void main() {
                GeneratedStateMachineClass s =
                new GeneratedStateMachineClass(0);

                s.t1();
                System.out.println("x = " + s.getX());
        }
}
```

If, as said, trigger *t1* causes the effect *x=3*, the function *t1()* will have changed the value of *x* to 3, which will be the output of the print operation.

For the case of the Garage door (5.3 and 5.5), which does not include variables and already contains the code to print certain messages detailing the actions that occur during the simulation, this Main class would even be simpler, such as something like the following code snippet:

```
public class Main {
        public static void main() {
                GeneratedGarageDoorClass sm =
                new GeneratedGarageDoorClass();

                sm.pressButton();
```

```
        }
}
```

The result of running this class would be to print the messages corresponding to entering the Closed state and then to exiting the Closed state and entering the Opening state (because that is the transition which triggered by the *pressButton()* event while being on the Closed state, which is the first state reached). This means that, firstly, messages would be output for the *stopMotor()* and *triggerEnergySaveMode()* functions, as these are activated when entering the Closed state. Then the same would happen regarding *triggerNormalEnergyMode()* (because it is activated when exiting the Closed state) and *runMotorForward()* (which will occur when entering the Opening state).

The submitted Main class is compiled and run by the same Ant file that ran the generation of the Java class by the Umple tool. The *build.xml* file executes the following sequence: generate Java code from the Umple code (previously generated via EGL); compile the generated Java class and the Main class provided by the teachers; execute the Main class (which uses the generated Java class).

5.3.7 Returning a simulation report

The Ant *build.xml* file allows the use of a Recorder which outputs to a specific file any information that would have been printed to the console. Because of this, after using the ProjectHelper class to run this Ant file, the State Machine Module can read this file (with the result of the recording) and output its contents to the communication part of the service. As mentioned before, the architectural concept of modules is represented in the code by different methods that the service's main code can call. Therefore, the state machine method will return a string with the contents of the file which recorded the execution of the Ant XML file.

The service will then use this simulation report to respond to the submission platform's client request, through a JSON object containing the contents of the report in String format.

5.4 The Sequence Module

This section describes the module developed for handling the input of sequence diagrams and the processes it uses to convert these diagrams into XMI representations of Colored Petri Nets, which can then be serialised and converted into a *.cpn* file that can be simulated by CPN Tools. Before that, there is a description of the structure of sequence diagrams, as well as an overview of the CPN Tools framework.

5.4.1 The sequence diagram

The sequence diagram is a behavioural UML diagram which is part of a subset called the *interaction diagrams*. These describe the exchange of messages between system elements.

[66] The sequence diagram is a two-dimensional chart and, although it does not describe exact time intervals, its vertical axis corresponds to the passage of time. The horizontal axis is used to represent the different elements that take part in the interaction.

The passage of time in an object is represented by a Lifeline. Messages are exchanged between lifelines. Messages can be of various types. Asynchronous messages and Asynchronous Reply messages are used when a process does not require a sent message to be replied to. When that is not the case, one can use a Synchronous message, which will include a mandatory Reply. There are also Create and Destroy messages, which can be used to, respectively, start or end lifelines.

Lifelines can contain objects such as execution specifications, which detail the execution of a process for a certain period of time.

Combined fragments are elements that can span throughout different lifelines for a period of time. They can be used to control the flow of the sequence. There are twelve fragments [82] that can control this flow in different ways: the Alternative fragment (*alt*) defines behaviours that happen alternatively based on some condition; the Optional fragment (*opt*) describes behaviour which is optionally triggered if some condition is fulfilled; the Loop fragment (*loop*): specifies behaviour that can repeat a different number of times based on some condition; the Break fragment (*break*) can be enclosed inside another fragment, where it is used to represent special behaviour that, in the case that some condition is met, will take place instead of the rest of the enclosing fragment; the Parallel fragment (*par*) details different behaviours that occur in parallel; the Strict Sequencing fragment (*strict*) is used to express different behaviours that must happen in a fixed order; the Weak Sequencing combined fragment (*seq*) is similar to the previous but defines a sequence which has to be maintained solely between specifications that are in the same lifeline; the Critical Region fragment (*critical*) defines a region whose behaviour must be treated atomically, which means that, for example, it cannot be run in parallel. There are more fragments which can simply be listed as they are very rarely used: the Ignore (*ignore{m,s}*), Consider (*consider{m,s}*), Assertion (*assert*), and Negative (*neg*) fragments.

To the different portions of a Combined Fragment (which specify these distinct behaviours that are influenced), we call Interaction Operands.

Other elements also exist such as State Invariants, used to define constraints that may apply to elements involved in an interaction.

Figure 5.6 shows a simple sequence diagram including lifelines, execution specifications, the *alt* combined fragment, and different types of messages.

5.4.2 Coloured Petri Nets

Coloured Petri Nets (CPNs or CP-nets) is a graphical modelling language targeted mainly at the construction and analysis of concurrent systems. [44] A coloured Petri net is a type of high-level Petri net (which is an ISO/IEC standard [42]) and is suitable for building

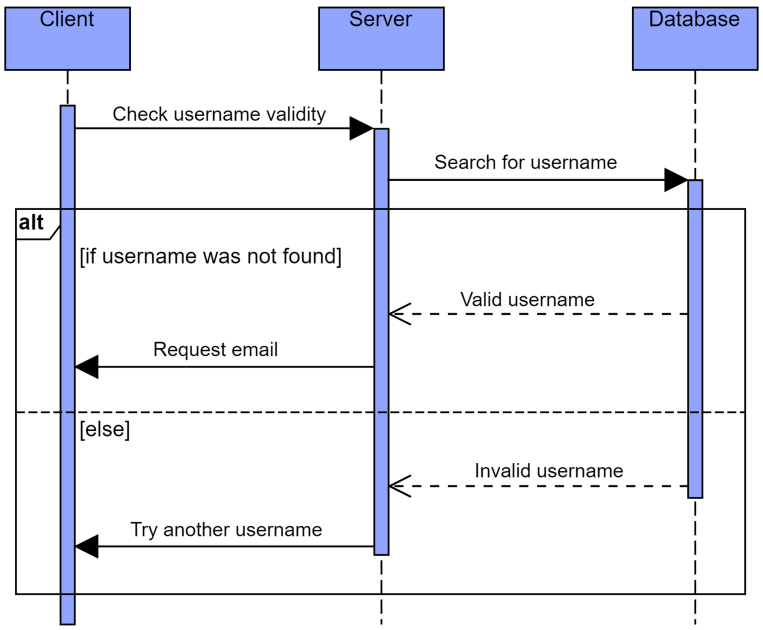


Figure 5.6: A simple sequence diagram representing the exchange of messages in the context of a client-server application.

compact and parameterised models. CPNs are also an extension of Petri nets (or place-transition nets): a mathematical modelling language composed of states and transitions connected through arcs.

5.4.3 CPN Tools in more detail

CPN Tools is a tool that can be used to edit, simulate and analyse CPNs. [17] It includes support for syntax checking, code generation, and simulation. It allows the generation and analysis of full and partial state spaces, and can report on boundedness and liveness properties.

The tool works as a standalone application with a graphical user interface, but it also offers a plugin (or a collection of plugins) for integration with Java environments, called Access/CPN [16].

5.4.4 Subset compliance and validation of model construction

The process described in [18], which was used in the context of this module, detailed the ETL code used to transform sequence diagrams, but did not include any information about validation of the input model. However, this ETL code (annexed to the cited work), implied a subset of the UML sequence sublanguage. Therefore, EVL code was

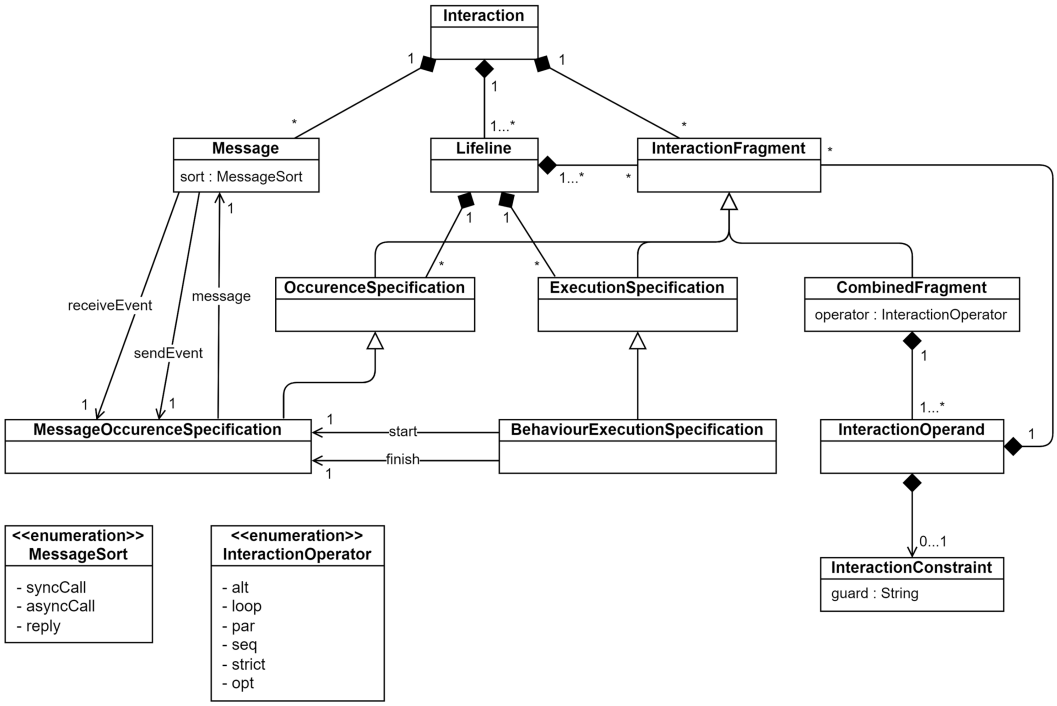


Figure 5.7: The metamodel of the supported subset of UML and sequence diagram.

implemented to test the model with respect to its construction and to the conformance to this implied subset.

The metamodel represented in figure 5.7 depicts the subset of UML that is supported by this module and derives from the ETL code presented in [18].

There was the need to firstly implement EVL constraints that guaranteed that all the supported model elements (lifelines, messages, combined fragments and specifications) had names.

The combined fragment types supported by the EGL code supplied by [18] were the following: *alt*, *loop*, *opt*, *par*, *seq*, and *strict*, so there was a need to check whether the received model complied with this. For the fragments that required guards (namely the alternative fragments), EVL code was added to check if a guard was included in their operands. Combined fragments were also verified with respect to what they cover, i.e. they should contain at least some element from some lifeline.

Messages should always have a sender and a receiver, which both should cover some lifeline. Although this work [18] did not differentiate between message types, to ensure coherency and correct model building, constraints were added to guarantee that synchronous messages always received their required reply.

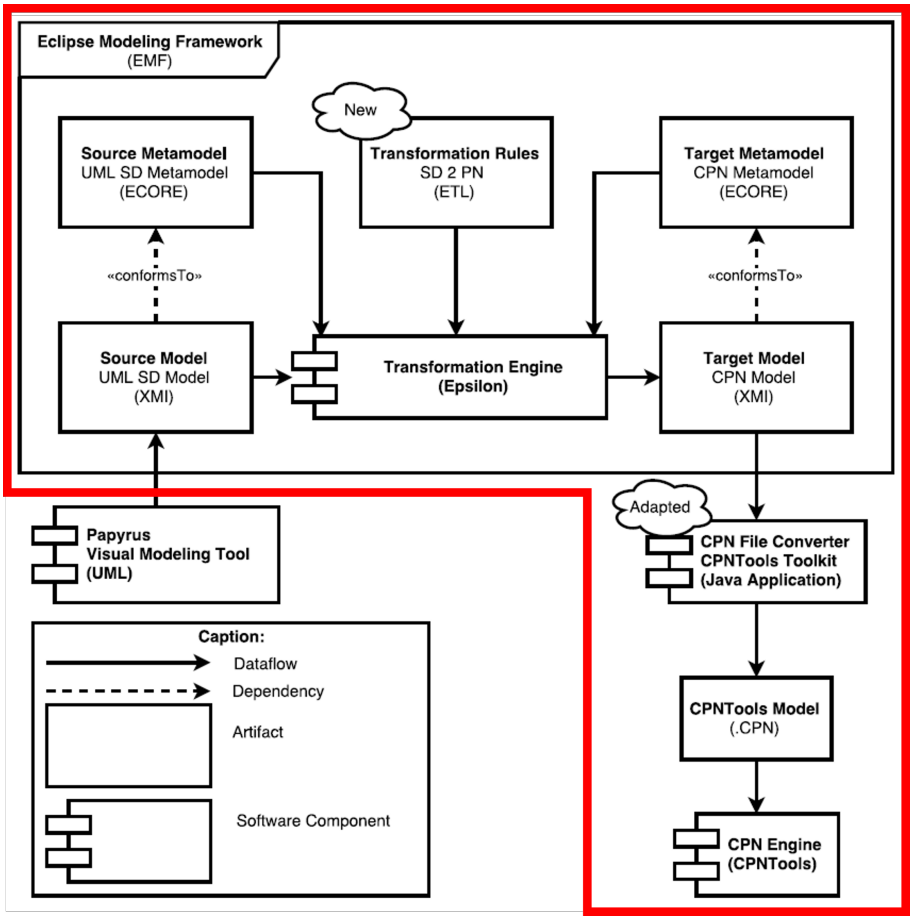


Figure 5.8: The solution design from [18] vs. the processes that could be automated in a Java context (in red).

5.4.5 Transforming the UML model into an XMI CPN model

This subsection and the next one follow the processes presented in [18]. Regarding the process described in subsection 5.4.7, however, the information that existed in the document was scarcer and required adaptation. In contrast, the transformation part of the process (detailed in this subsection) was able to closely follow the information provided by the referred work, which was almost complete in terms of the ETL code (which only had some auxiliary operations missing and some minor changes required to the code due to differing versions of libraries).

This ETL code uses the information contained in the elements of the UML sequence diagram to generate the CPN structures – places, transitions, and arcs (and variables, if needed) – and populate them with the necessary information to make the CPN equate to the diagram. The document uses a set of rules to accomplish this goal, which are shown in annexe V.

As it can be seen in figure 5.8, to define the transformation from UML to an XMI CPN model, besides from the ETL file (to define the transformation rules), Epsilon's

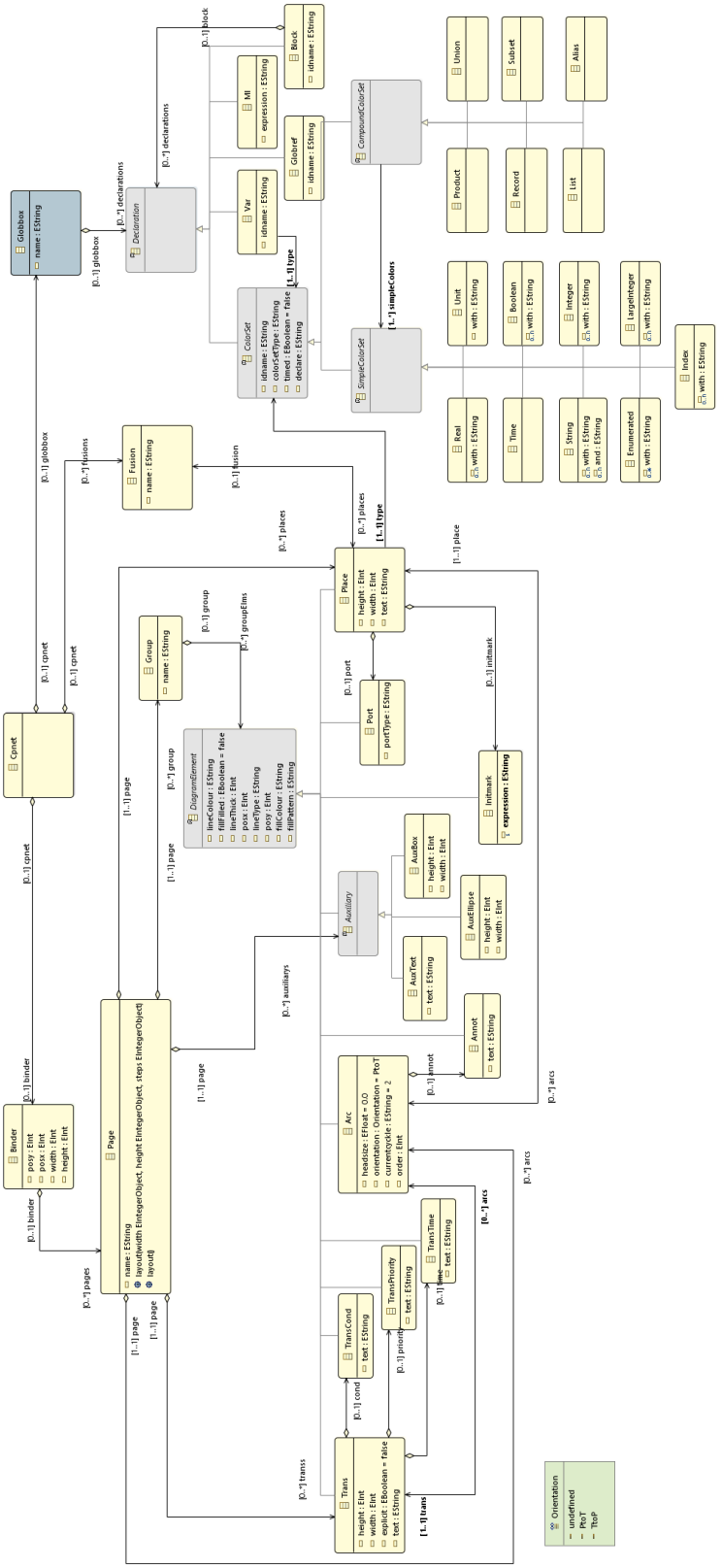


Figure 5.9: The XMI Coloured Petri Net metamodel provided by the CPN Tools Toolkit [36].

tools require a source model (the model submitted through the Platform) and metamodel (already implicitly defined in the case of a UML model). Epsilon also needs a target XMI model and a metamodel. These can both be defined using, respectively, a *.model* file and a *.ecore* file. These files are used to support modelling in Eclipse and are part of the Eclipse Modelling Framework [23].

The target model should logically be empty, while the target metamodel should define the structure of an XMI CP-net. For this purpose, the CPN Tools Toolkit Ecore metamodel was used. CPN Tools Toolkit [36] is a set of plugins that allows the serialization of XMI CPNs into the *.cpn* format (which is used by CPN Tools). For that purpose, the plugins contain an Ecore metamodel that defines the structure of XMI CPNs, which can be used as the metamodel of the target model of this transformation. This metamodel is presented in figure 5.9. With this information, Epsilon is finally capable of generating an XMI representation of a coloured Petri net from the submitted UML sequence diagram.

5.4.6 Converting the XMI CPN model into the CPN Tools format

For the process of converting the generated XMI model into the CPN Tools input format (*.cpn*), there is a set of plugins available online called the CPN Tools Toolkit [36]. These plugins are available through GitHub [35]. Because they only depend on EMF, the plugins can be imported into the Maven project, by having it depend on a local Maven repository that contains the jars for these libraries.

This allows Java to use the classes made available by the plugins to serialise the XMI file. Using the EMF libraries together with these plugins and following the instructions described in the test plugin and in the documentation provided by the CPN Tools Toolkit, the serialisation of the XMI file can be achieved through the following code:

```
URI u = URI.createFileURI("src/main/sequence/cpnModel.model");
ResourceSet resourceSet = new ResourceSetImpl();

resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
put("model", new XMIResourceFactoryImpl());

resourceSet.getPackageRegistry().
put(CpntoolsPackage.eNS_URI, CpntoolsPackage.eINSTANCE);

Resource resource = resourceSet.getResource(u, true);
resource.load(Collections.emptyMap());
Cpnet net = (Cpnet) resource.getContents().get(0);

CpnToolsBuilder builder = new CpnToolsBuilder(net);

builder.serialize(new FileOutputStream(
```

```
new File("src/main/sequence/cpnModel.cpn")));
```

Although in figure 5.8 the serialisation process is not part of the EMF context, in the Simulation Service this process is implemented in Java, in an automated way, in sequence with the remaining steps highlighted in the figure in red.

5.4.7 The CPN Tools solution for simulation in a Java context

The process described by [18], upon which this module is built, does not include automated and integrated simulation of the CPN Tools format. However, for integration into Java environments, CPN Tools provides a set of plugins called Access/CPN [16]. By manually installing the jar files for each of the required plugins into a local Maven repository, the plugins can be imported into Maven by adding the corresponding dependencies to the framework's Maven project.

However, Access/CPN has close to no documentation, and although it includes some test classes (in the *model.test* plugin), it was hard to understand how to properly use Access/CPN to simulate *.cpn* files. Not only did unexpected errors constantly occur when processing and simulating the generated files, but these errors also showed up when running Access/CPN's own test models. This is possibly related to the fact that Access/CPN was last updated in 2011 and does not seem to be maintained, which might cause problems with more recent Java Development Kits (JDKs).

Although the intention of this module was to automate the whole process highlighted in figure 5.8, because of these technological difficulties the module can only reach the penultimate step of the highlighted sequence of processes. However, to still offer the students the capability to understand more deeply the construction and logic of their sequence diagrams, the decision was taken to adapt this solution instead of removing it.

CPN Tools makes available a simulation engine capable of simulating CPNs with real-time visual aid. It simultaneously offers the option to produce a simulation report, which might be easier for students to understand and is more in line with the concept of the other simulation modules and the framework in general. For this reason, it is still relevant to offer the students the ability to perform the process sequence highlighted in 5.8, even if it is not in a fully automated manner. For that reason, the decision was taken to return to the user the generated CPN model that can then be simulated using the CPN tools framework. Therefore, the students can still submit their sequence diagrams (which do not have any execution semantics) and receive in return a model that is easily simulatable and that can provide great help in understanding the dynamic aspects implied by their model construction.

5.4.8 Returning a CPN model

Having decided to return a CPN model to the client platform as an indirect form of model feedback, it was then necessary to develop this extra functionality for the service.

Maintaining the client-server logic, supported by the AWS database, the solution to this problem is to upload the model to the database with a randomly generated key, and then supply the client platform with that key, for it to obtain the model. This equates fundamentally to the reverse of the process used to retrieve the models, mentioned in subsection 5.1.2 (and shown in figure 5.1).

The code snippet below exemplifies the code used for this purpose:

```
String key = randomStringGenerator.nextString();

final AmazonS3 s3 =
AmazonS3ClientBuilder.standard().build();

        String bucketName = "bucket";
        File file = new File("model.cpn");

        s3.putObject(bucketName, key, file);
return key;
```

5.4.9 CPN model simulation with CPN Tools

After the platform which communicates with the service retrieves the CPN model and returns it, the student can then simulate it in CPN Tools by simply importing it into the tool and dragging the simulation menu into the workspace. As said, CPN Tools offers runtime visual feedback of the simulation and, if the generated model is a bit too complex, there is always the possibility to produce a simulation report to analyse the simulation in simpler terms. These processes will be further detailed in the case study presented 6.3.

Taking all of this into account, despite not being given completely direct feedback on the construction of their sequence diagrams, students can use this module to receive a model which is promptly simulatable and that can easily supply them with the feedback they need to further understand their models, as well as the Interaction and Sequence sublanguages of UML.

5.5 The Activity Module

The activity diagram model-checking module uses an activity diagram to generate code (in the NuSMV language) representing a state-transition graph, which can then be model-checked through the use of additional specifications. This section showcases the elements of the activity diagram and then details the process of generating the NuSMV code and checking it using the NuSMV tool.

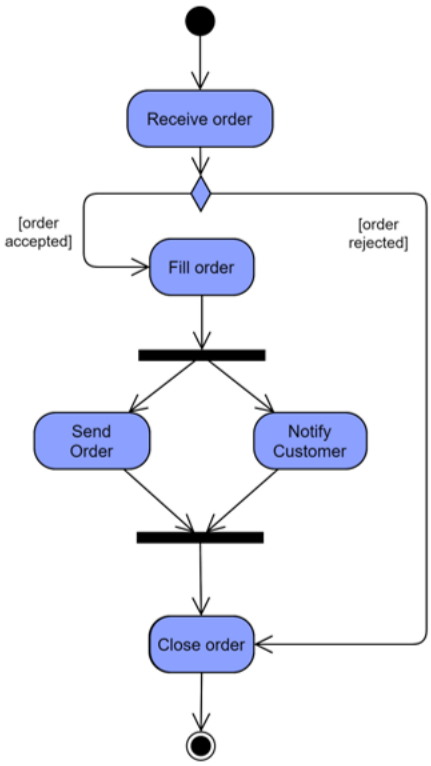


Figure 5.10: A simple activity diagram representing an order being processed.

5.5.1 The activity diagram

An activity represents behaviour in the form of a flow of actions. [81] This flow is defined through activity nodes connected by activity edges. These nodes can represent actions and objects, and can also be control nodes. There are two types of edges: control flow (which details the sequence of execution) and object flow (which shows the flow of objects between actions or activities). [91]

Control nodes are used to control the flow between other nodes. Similarly to state machine diagrams, activity diagrams include initial and final nodes, which are two types of control nodes. Other types include the flow final node and the decision, fork, join, and merge nodes. The flow final node is used to end only a specific flow, while the remaining control nodes relate to the control of the interactions between multiple activity edges.

The decision node is a control node with multiple outgoing edges, but the flow is restricted to only one of them (a token cannot traverse multiple outgoing edges). The fork node is similar but it supports parallelism, meaning that tokens arriving at them can follow through multiple flows simultaneously. The join node has multiple incoming edges and one outgoing edge and is used to synchronise concurrent flows, which means it needs to receive all of the incoming flows to be able to proceed. The merge node is similar, but it simply joins multiple incoming flows into a single outgoing flow (not needing all of them to arrive for the flow to continue). Control flow edges that come out of decision

nodes must have guard conditions.

Activity partitions are divisions or swim-lanes that can express attributes, instances or classifiers which can constrain the behaviour which is being represented throughout their extension.

Figure 5.10 shows an example of a simple activity diagram, using actions and control flow edges interacting through the use of the Decision, Fork and Join activity nodes.

5.5.2 Model checking vs. simulation

Model checking and model simulation are two approaches that can be used to aid modelling education. Although model checking provides a more complete form of verification [90], simulation can be used in education to ease the understanding of the inner logic of models, especially regarding behavioural models [93].

However, as was explained in section 4.2, the available activity diagram simulation tools (which, in the open-source realm, was mainly Moka) could not simulate simple models. Moka could only simulate activity diagrams whose structure basically represented robust code in model-form, using a lot of normally unused and complex UML elements which did not match the goal of this framework and did not fit the ultimate target audience (undergraduate students).

A model checker is therefore a way of providing feedback on the model construction and inner logic, even if it is a less personalised way of offering that feedback. As mentioned in the previous chapter, Ul Muram et al. [80] provide a process for transforming activity diagrams into NuSMV code that can be checked by the tool.

5.5.3 NuSMV in more detail

NuSMV is a tool capable of representing finite-state systems and checking them through the use of LTL and CTL specifications [54]. LTL and CTL are *temporal property-specification languages*, which means that they are languages that use temporal-logic expressions to specify properties. In temporal-logic model checking, one can check a labelled state-transition graph (representing a finite state system) by testing if it satisfies temporal-logic formulas that specify certain properties. [89]

In the Linear Temporal Logic (LTL), formulas are built using propositions and the regular boolean connectors ("and", "or", "not", etc.), in tandem with some additional temporal connectors. These connectors are G, F, X, and U, which respectively mean "always" (or "globally"), "eventually" (or "in the future"), "next", and "until". The Computation Tree Logic (CTL) adds the path quantifiers E and A to LTL. For example, the connector EF means that there exists a computation in the future which guarantees a certain property, while the connector AF means that all computations must eventually guarantee that property. Figure 5.11 can help visualise the meaning of the various temporal logic connectors. To the superset including both LTL and CTL, we call CTL*.

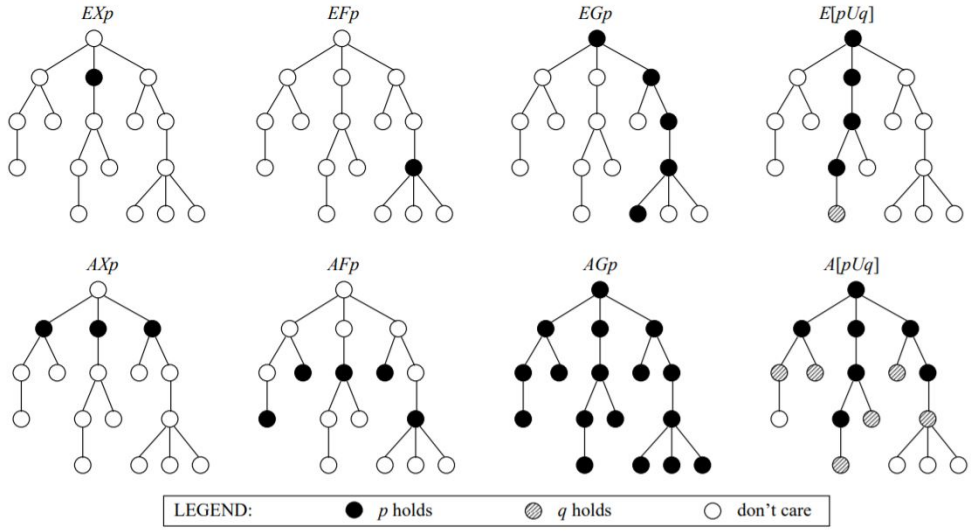


Figure 5.11: CTL* (LTL + CTL) connectors tree logic [13].

In addition to LTL and CTL specifications, the NuSMV language allows the description of state-transition graphs. NuSMV code is composed of various sections, which all start with capitalised keywords. In a simplified way, the MODULE section names the graph (and optionally defines parameters); the VAR section declares state variables; the ASSIGN section assigns initial values to the variables and also defines the value of variables in the next states (i.e. after each transition) given their value in the current states. The structure of NuSMV code will be further discussed in subsection 5.5.5.

NuSMV can be used through its source code or its binaries (both available on NuSMV's website).

5.5.4 Subset compliance and validation of model construction

The paper [80] which offered insight into the conversion of an activity diagram into NuSMV code (representing a labelled state-transition graph) does not cover activity partitions, nor does it cover object nodes, object flow edges, or flow final nodes. Despite this, the article covers the most widely used elements of activity diagrams.

Although the paper does not cover the case where there are various control nodes in succession (which could create problems, especially in the case of decision nodes), that part was added in the process of implementing in EGL the logic presented by the paper (which is described in subsection 5.5.5). However, to make the addition of these cases possible (regarding successive control nodes), there was the necessity to change the model checking rules, which lead to the need to make a small adjustment to the subset to allow only one activity final node.

The metamodel in figure 5.12 shows the subset of the UML metamodel that is supported by the module.

The validation of the model construction and compliance with the subset stated above was, again, guaranteed through the use of the Epsilon Validation Language. The EVL file that was created includes constraints targeted at checking aspects of the model's construction such as always having names for actions and control nodes. These names should also follow an input format that uses only letters, numbers, dashes, underscores, and spaces (and the latter will even be removed later). This is because NuSMV can mistake some characters and symbols for logical elements of the NuSMV language, causing unwanted problems.

The EVL also checks for the number of initial and final nodes (which should be only one for both cases) and checks if the remaining types of control nodes have the required amount (multiple or single) outgoing or incoming edges, according to the node type. It also verifies whether edge guards meet the required input text format (mentioned above) and are between square brackets.

The verification process also includes the task of checking if there are no control node types that are not supported by the subset (e.g. flow final nodes). There are also constraints to confirm whether an activity edge always has a target node and a source node and if these are not the same node.

There is also the verification of some specific cases that help to avoid errors in the following stages. For example, although this notation is not customary in activity diagrams, the UML language allows Decision node and Fork node elements to have multiple incoming edges. This means that before these nodes there is an implicit Join node. Even if this structural decision is uncommon, as a safeguard, these implicit Joins should be avoided (or replaced by explicit Join nodes) when submitting models to the generation and model checking process.

With the activity diagram verified with respect to construction correctness and subset compliance, NuSMV code (representing a labelled state-transition graph and detailing temporal logic rules) can finally be generated from the model.

5.5.5 Generating the NuSMV code

The NuSMV code was generically described in subsection 5.5.3, however, it is important to further explain its structure and commonly used elements.

In the VAR section, variables are defined. These are either boolean or can take on a fixed set of values (which also need to be stated in this section of the code).

In the ASSIGN section, there must be a code portion for the various variables: firstly, detailing a variable's initial value (a boolean value for boolean variables, or a value out of the previously defined for the variables with specific value sets); then, detailing a variable's future value given the current values of the graph's variables. Using the TRUE value for the current state, one can define the default next value, in the case that none of the other cases is met. This next value can be a single value or a set of permitted values.

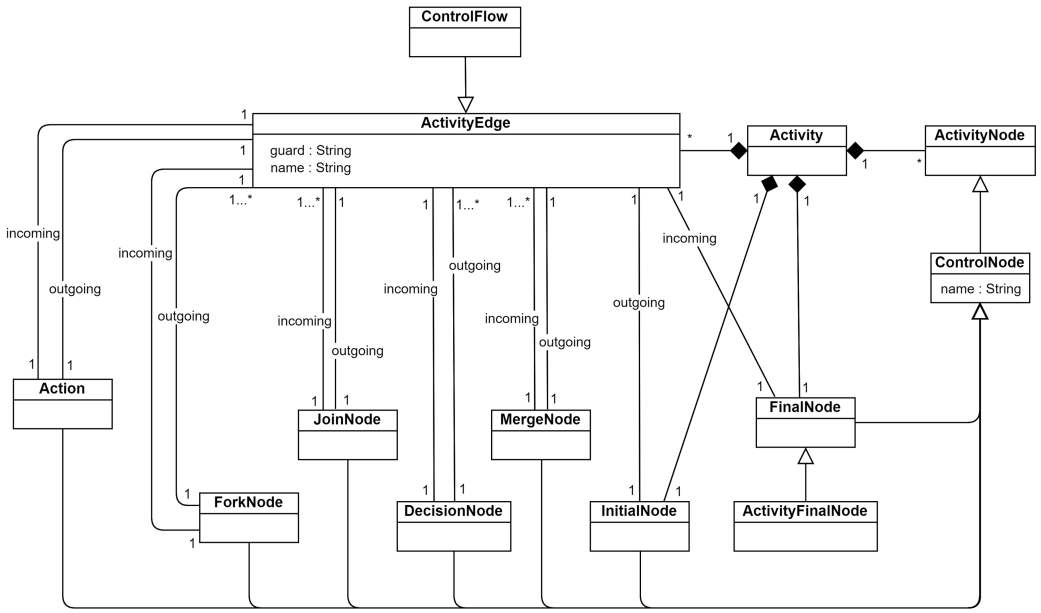


Figure 5.12: The metamodel of the supported subset of UML and activity diagram.

Lastly, the SPEC section contains the rules for checking the graph that was constructed using the steps described above. It uses LTL and CTL rules to check logical conditions. This means, for example, that one might test whether a variable y being TRUE implies (or not) that x will have some specific value in all future computations. This would be expressed like $y = TRUE$ –> $AF(x = value)$.

Below is a simple example of NuSMV code (inspired by the NuSMV User Manual [54]) that can help to further understand the concepts explained above.

```
MODULE main
VAR
    y : boolean;
    x : {ready, busy};
ASSIGN
    init(x) := ready;
    next(x) := case
                   (x = ready) & (y = TRUE)    : busy;
                   TRUE                        : {ready, busy};
               esac;

SPEC
    AG(y -> AF x = busy)
```

The code shown above starts by defining two variables. Variable y can take the values

TRUE and FALSE, while x can take the values ready and busy. Nextly, it is determined that variable x will start as ready. Then, it is determined that, if x was ready and y was TRUE, in the next step x will take on the busy value. If not, x is allowed to take any value in the next step (out of its value-set). The specification used to test the construction of this model has the purpose of verifying whether, in all cases, the occurrence of y (i.e. $y = TRUE$) will always imply that x takes on the busy value in the future.

Regarding the implementation of the Service, the process used to transform the UML activity diagrams into NuSMV code was adapted from the previously mentioned Ul Muram et al. approach [80]. Regarding the VAR code section, this approach firstly mentions how actions and almost every control node can be represented by boolean variables. The exception are decision nodes because they offer optional paths based on conditional values. Therefore, the variables representing decision nodes should have a value set composed of these guard values. Then, Epsilon's generation engine produces NuSMV code for the VAR section taking into account the activity node elements present in the UML model.

For the ASSIGN section, the process followed was very similar to the one detailed in the Ul Muram et al. paper. One of the materials provided there is a table showing visual examples of the logic behind the generation of code for each type of activity node. This table is presented in annexe III. Taking into account the generation logic presented in the paper, EGL code was written to produce the desired code from the respective UML elements.

The SPEC section was adapted from the information showcased in aforementioned paper. The starting point for defining the different generation cases was present in the paper and included another table, which is shown in annexe IV. However, as mentioned before, the paper did not support models which had multiple decision nodes in sequence. Consequently, the desired NuSMV code was different (and more complex) than the one described in the paper, and the generation of the LTL and CTL specifications that was created had to include recursion to accommodate this new case. This SPEC section uses the activity edge model elements together with the activity node elements to represent the control flow through these rules.

As was also mentioned before, in the generation of NuSMV code via the EGL file, the spaces in the model element names were removed. This is because NuSMV only accepts single-word variable names. Because of this, all the spaces were substituted by underscores.

5.5.6 Running the NuSMV code

After the NuSMV code (generated through EGL) is output to a specific *.smv* file, it can be run. As stated in subsection 5.5.3, the tool can be used through its source code and binaries. Using the binaries, it is possible to run NuSMV through the command line, which means that the approach mentioned in 5.3.5 and 5.3.7 can be used again in this

case. What this means is that it is again possible to use the Apache Ant's ProjectHelper Java class to execute an Ant build file containing a task for running NuSMV, as well as a Recorder to register the output. The default output produced is a list of the LTL and CTL specifications that were created and whether they were true or not, together with counterexamples for the cases that were wrong. However, the *-dcx* option was used to remove the counterexamples from the output. [54] This option was included to make the output less extensive and easier for the students to process.

The Ant task for this process is dependant on the operating system because the NuSMV binaries and executable differ based on that factor. The validation of the framework (chapter 6) was made in the Linux virtual machine where the UML Education Platform is deployed, therefore using the NuSMV Linux binaries. However, the framework was also tested with the NuSMV Windows binaries. The code that follows was used to test this module with Linux (to test with Windows, the binary folder would be different and the executable parameter would be *NuSMV.exe* instead of *NuSMV*):

```
<exec dir="\${basedir}/NuSMV-2.6.0-Linux/bin"
executable="NuSMV" resolveexecutable="true">
    <arg line="-dcx \${basedir}/code.smv"/>
</exec>
```

This task (called the *<exec>* task) is used with the NuSMV executable and includes as arguments the aforementioned "dcx" option, as well as the target *.smv* file.

5.5.7 Returning a simulation report

The NuSMV output (recorded from the Ant execution) contains a lot of information, which created the need to clean it up and make it more direct and understandable. The file to which the execution was output is scanned to extract only the specifications that were not true. If no incorrect specifications are found, a small sentence is added to the report to state that the model passed the model checking process. Otherwise, a sentence is added reporting that the model failed the process and detailing the specific specifications that were deemed to be false by the NuSMV tool. This output can be further understood through the case study presented in 6.4.

Although the students are expected to understand computational logic, they may not be familiar with the LTL and CTL connectors. This, however, is not a big problem, as these connectors have relatively straightforward meanings. Therefore, a small legend is added to the output, explaining in simple and understandable terms the logical meaning of the connectors which are used.

The Activity Module method will then return this output to the service's main code (responsible for communication), which will return it to the client platform through a JSON object.

Chapter 6

Design Evaluation

This chapter presents the methods used to evaluate the framework produced [33] in accordance with the Design Science paradigm [37, 94].

6.1 Methodology

As referred above, this chapter details the validation that was conducted within the parameters of the Design Science methodology (detailed in subsection 1.6.1), using an Experimental approach of Simulation.

To test the operations performed by the service, multiple models (representing common student input) were constructed and fed to the system, targeted at testing the various steps of the simulation and transformation process, as well as the possible weaknesses of the service. The original idea was to use a model dataset composed of the submissions of students from the last edition of the Software Engineering course. This, however, turned out to be impossible, as none of the models that were supplied passed the part of the process where the service validates model construction and subset compliance. This meant that no models would be simulated, as they would not comply with the requirements to do so.

The service was therefore tested with the input of various UML models of the three supported diagram types (sequence, state machine and activity). Section 6.5 details the nature of several models used for this purpose. Furthermore, from these tests, three case studies were derived, one for each simulator. These case studies are targeted at showing, for each simulation module, their respective transformation and simulation processes applied to specific cases that simulate reality, representing the input of students and how the framework uses it to achieve its goal of simulation. These case studies are detailed in the following three sections (6.2, 6.3, and 6.4).

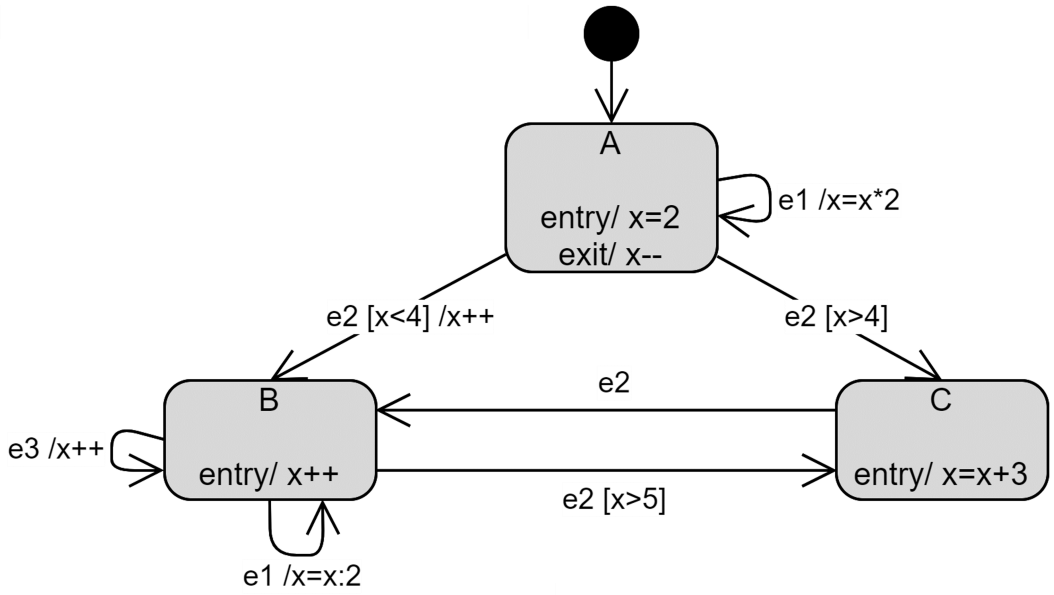


Figure 6.1: State machine model from a lab exercise from the 2020 Software Engineering course.

These sections focus on showing, through the use of specific cases that mimic real input, the processes that were developed. Instead of focusing on showing the range of feedback that can be given, these are instead targeted at making the whole process more tangible and understandable. Therefore, the case studies do not include overly complex input diagrams, to facilitate the comprehension of each of the three modules.

6.2 Case Study 1: State Machine Diagram

This case study aims at showcasing the transformation and simulation processes conducted by the service (described in 5.3), starting with a UML state machine model and ending on the feedback of the simulation that was produced.

For this case study (regarding the simulation of state machine diagrams by the framework), the choice was to use an example model which is included in an exercise from the lab classes of the Software Engineering course. This model is shown in figure 6.1.

As the simulation service is primarily intended to be used in the context of this course, it makes sense to, if possible, validate the simulation service through the use of an example that would show up in the course. This model includes some simple elements of the state machine diagram which are commonly used by students. Beyond the ever-present states, transitions, and initial state, the diagram includes transition triggers, as well as guards and effects for some transitions. It also includes entry and exit activities.

In this case, this diagram will therefore be used as our artificial student input, to showcase the whole process. The model submitted by the student would be a UML file

```
class GeneratedStateMachineClass{
  Integer x;
  Sm {
    A {
      entry/{x=x+2;}
      exit/{x--;}
      e2 [x<4] /{x++;} -> B;
      e2 [x>=4] -> C;
      e1 /{x=x*2;} -> A;
    }
    B {
      entry/{x++;}
      e2 [x>5] -> C;
      e1 /{x=x/2;} -> B;
      e3 /{x++;} -> B;
    }
    C {
      entry/{x=x+3;}
      e2 -> B;
    }
  }
}
```

Figure 6.2: The Umple code generated through EVL for the model presented in 6.1

with a UML state machine element containing the UML elements corresponding to those represented in the figure, with all of the same relationships and attributes.

As referred before, because of the limitations of Umple, students should have extra care with some aspects of the construction of their state machine diagrams if they want to submit them for simulation. As stated in section 5.3, guards, activities and effects must equate to valid code. Extra code such as auxiliary functions or variable declarations should also be added to the constructed state machine via a UML Comment.

After a platform sends a simulation request to the Simulation Service and the service fetches the UML model from the AWS database, the model will be validated using EVL. If the model fails the validation, the student will receive as feedback a list of messages reporting the constraints that were not satisfied. If the model is successfully validated using the Epsilon tools, the service will proceed to generate the Umple code. For the model used in this case study, the generated Umple code is as shown in figure 6.2.

As explained in subsection 5.3.4, this Umple code corresponds to a textual representation of the state machine elements (a class with a state machine with the elements inside). The initial state is implied to transition into the first state in textual order (hence why state *A* is listed first). For each state, its outgoing transitions are represented by their trigger and include the target state after the arrow. Guards and effects are also represented in their corresponding transitions. As the integer variable x is used in the model (and therefore specified in the model file using a UML comment), it will be declared in the Umple code before representing the state machine element.

The next step in the process conducted by the state machine simulation module is to

use the Umple tool to generate Java code from the generated Umple code. This generated class is a bit more complex and is not really meant to be modified, so it would be a bit confusing and futile to explain it here, therefore it is only important to explain which methods it offers.

So, as the triggers used in the Umple code are *e1*, *e2* and *e3*, there will be functions in this class that allow these events to be triggered, which will change the value of the variable *x* in respect to the rules and structure derived from the Umple code (which itself derives them from the UML model). The, the generated Java class will include the following methods: *e1()*, *e2()*, *e3()*, and *getX()* (because the integer variable *x* is defined in the Umple code).

As mentioned in 5.5.6, this simulator needs the teachers to provide the simulation instructions via a *Main* Java class which they would have previously submitted to the service and that would already be placed in the correct location for the service to use it.

The original exercise from where the example model of this case study is taken from suggests the event chain *{e1, e2, e2, e2}* for the students to analyse the outcome when applied to the state machine diagram. Therefore, the teachers can provide a main class that contains the corresponding set of instructions. The idea would be for the teachers to inform the students that the simulator would trigger that specific event chain, and then students can submit their models to the platform and see the results of triggering these events on their model.

An example of a main class that could be submitted by the teacher is the following Java code:

```
public class Main {
        public static void main(){
                GeneratedStateMachineClass s =
                new GeneratedStateMachineClass(0);

                s.e1();
                System.out.println("x = " + s.getX());
                s.e2();
                System.out.println("x = " + s.getX());
                s.e2();
                System.out.println("x = " + s.getX());
                s.e2();
                System.out.println("x = " + s.getX());
        }
}
```

For this Main, the service's output would be the following:

```
[exec] x = 4
[exec] x = 6
```

```
[exec] x = 7
[exec] x = 10
```

Consequently, given the event sequence supplied by the teacher (*{e1, e2, e2, e2}*) and the UML state machine diagram submitted, students can obtain feedback on the simulation of their model and on the way the value of the variable *x* would change, taking into account the event-chain provided.

This should help the students to understand how the construction of their models can impact the changing of these values. Furthermore, if teachers provide the students with the expected evolution of the value of *x* in the model (given the execution of the selected event sequence), students have a very easy way of concluding whether their models are correctly built or not. This simulation report presents a useful asset to the students, because even if the submitted state machine diagram passes the model validation phase (or other external verification tools), it might still produce unexpected and unwanted changes to *x*. This would mean that the model's inner logic ended up not corresponding to what the student intended it to be.

6.3 Case Study 2: Sequence Diagram

This case study describes the process used by the service to transform sequence diagrams into Coloured Petri Nets that can easily be simulated using CPN Tools, as described in 5.4.

Using the approach to return a CPN model to the students has a downside: as the CPN models generated from a sequence diagram are slightly more complex than their UML counterpart, when using overly complex sequence diagrams, the generated CPN will be even more complex (despite it gaining in being able to be simulated). Therefore (and similarly to the other case studies presented) the sequence diagram used in this section is relatively simple, to prioritise easing the understanding of the module over showing its whole range.

The model chosen for the case study uses one of the six supported combined fragments, the Optional fragment, and includes messages and lifelines. It is shown in figure 6.3.

The first step, after the model is submitted and obtained from the database, is the model validation phase. As said in 5.4.4, students must make sure that they name all elements of the diagram, and that they only use valid combined fragment types, among other things. If this is not the case, as mentioned in the previous section, the students will receive feedback on the infringed constraints, which they will then have to amend.

The next step is to transform the sequence diagram into an XMI version of a Coloured Petri Net, using the Epsilon Transformation Language. Applying the rules described in [18] and listed in V, Epsilon first sets up the initial places of the CPN and creates places for each of the events that occur in each of the existing lifelines. This is, in this case, the

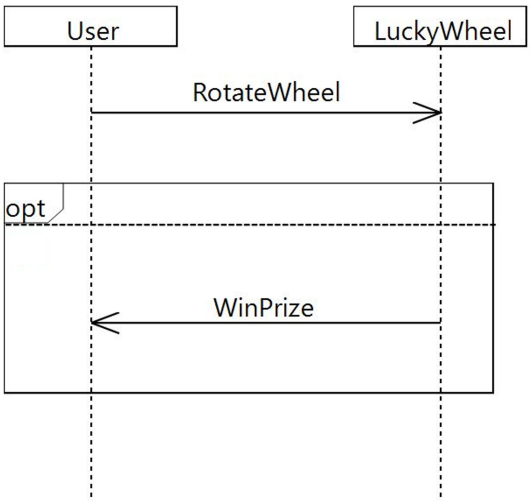


Figure 6.3: A relatively simple sequence diagram representing a luck-based game.

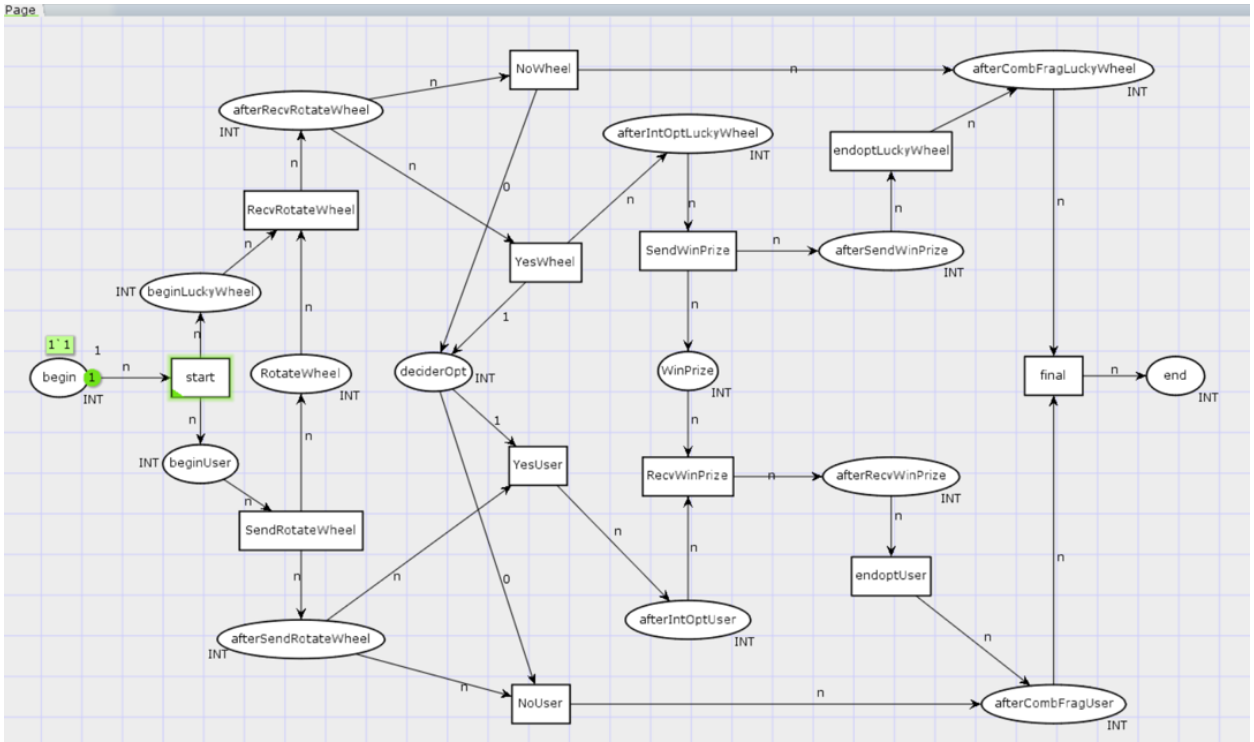


Figure 6.4: The CPN Tools model generated from the sequence diagram shown in figure 6.3.

SendEvent and the *ReceiveEvent* UML elements that are created whenever two lifelines exchange a message.

The following step is to process the Optional combined fragment, creating two paths in the CPN, one where the message inside the optional fragment (*WinPrize*) is sent and received, and one where it never happens. The remaining step before the final transformation and addition of the final place is to create places and transitions to represent the

first message (*RotateWheel*), which is mandatory.

After the XMI CPN model is generated, it is converted into the *.cpn* format using the CPN Tools Toolkit (as described in subsection 5.4.6). After that, the model is stored in the database and the access key is returned to the platform that made the simulation request. After the students receive the CPN model from the service, they can simulate it using CPN Tools. Although this is not a direct part of the module or the service, it is relevant to show how the students can use this model, to further explain and detail the capabilities of the service that was implemented.

The generated CPN model is shown in figure 6.4. Although this model is complex, the students can still derive useful feedback from it, using the CPN Tools Simulation tool, together with the option to generate a simulation report[1]. This report is easy to understand because it only contains the transitions triggered, which are the elements in the simulated CPN model that correspond to the original UML elements.

While running the simulation tool, multiple outputs can be generated to the report file. In this case, the simulation can produce the case where both messages are sent (i.e when *WinPrize* happens) and also the case when only *RotateWheel* happens.

The following code snippet shows the output in the case when both messages occur:

```
1       0       start @ (1:Page)
2       0       SendRotateWheel @ (1:Page)
3       0       RecvRotateWheel @ (1:Page)
4       0       YesLuckyWheel @ (1:Page)
5       0       YesUser @ (1:Page)
6       0       SendWinPrize @ (1:Page)
7       0       endoptLuckyWheel @ (1:Page)
8       0       RecvWinPrize @ (1:Page)
9       0       endoptUser @ (1:Page)
10      0       final @ (1:Page)
```

This step description shows that, firstly, *RotateWheel* was sent and received, and then the *opt* fragment was triggered (in both lifelines). Then, the *WinPrize* message was sent and received, and then, after the interaction operand was over in both lifelines, the execution reached the end of the CP-net.

The code below details the information that is output to the report file when the second message is never exchanged:

```
1       0       start @ (1:Page)
2       0       sendRotateWheel @ (1:Page)
3       0       RecvRotateWheel @ (1:Page)
4       0       NoQuiz @ (1:Page)
5       0       NoUser @ (1:Page)
```

[1] This can be achieved by ticking the option: *Options -> Simulation Report -> Save Report*.

```
6          0          final @ (1:Page)
```

In this step description, the *RotateWheel* message (which always must happen), is sent and received. After that, the *opt* combined fragment is not triggered, and so the execution reaches the end of the CPN, without ever sending or receiving *WinPrize*.

Therefore, even without complete automation, the students can still submit sequence diagrams and receive CPN models which can be simulated readily. This can help them to understand how sequence diagrams work, especially regarding how the different combined fragments influence the flow of information between lifelines.

6.4 Case Study 3: Activity Diagram

This case study describes the processes of transformation and model checking of activity diagrams (detailed in 5.5).

Because proper simulation had to be discarded regarding this module and replaced with model checking, this case study will be slightly different. To demonstrate the type of rules that can be broken by a model, this case study will have as its subject two different versions of the same model: one with a construction error, and another which is correctly constructed.

These simple models were constructed using Papyrus [24] and represent a very simple starting menu for some game. These models are presented in figure 6.5.

The difference between the two models is the node that gathers the flows from the two game-mode options, before confirming the game mode selection. In diagram **A**, the flows are merged by a Merge node, while in diagram **B** they are joined by a Join node. The problem with the second is that it will need both flows to be true to proceed. However, because the game mode choice is done through a Decision node, which only flows through one of its optional edges, the Join node will never receive both incoming flows and the final node will never be reached.

After a simulation request is received and the model is obtained, the first step in the process is to validate it with respect to its construction. In this case, both models would pass that stage. If that was not the case, a report of the failed validations would be output to the user.

As explained in section 5.5, after the validation phase, NuSMV code will be generated. This code firstly includes the declaration of variables that equate to activity nodes. Then, it includes (as explained in 5.5.3) the definition of the nodes' initial values and what their values will be after a transition (and given the current node values). These sections will be very similar in both models because they are not affected by the inner logic of the model.

However, the third part of the code – the CTL* specifications – is affected by the different inner logic of the models caused by the use of the different node types. The NuSMV code is too extensive to detail in the context of this case study, but, anyhow, most of it is very similar between the two models. Therefore, only the CTL* specifications will be

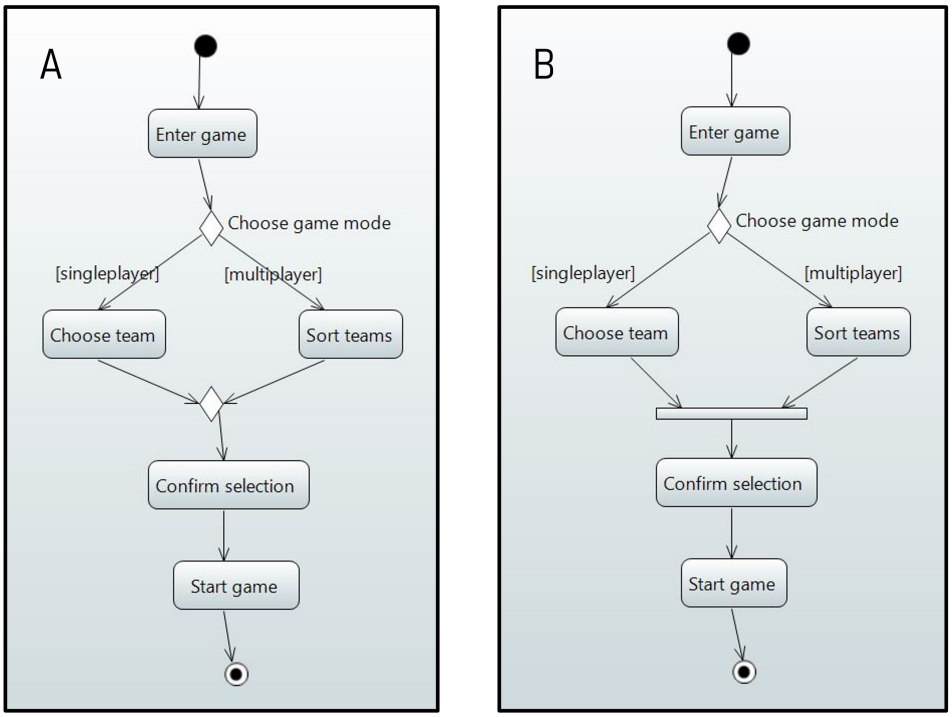


Figure 6.5: Properly constructed (**A**) and wrongfully constructed (**B**) activity diagrams for a simple game menu.

shown for each of the models.

Following is the code for the correct model (**A**):

```
LTLSPEC G (InitialNode -> F Enter_game)
LTLSPEC G (Start_game -> F ActivityFinalNode)
LTLSPEC G (Confirm_selection -> F Start_game)
LTLSPEC G (Enter_game -> ( (X X Chose_team) xor (X X Sort_teams)))
LTLSPEC G (Chose_team | Sort_teams -> F Confirm_selection)
CTLSPEC EF ActivityFinalNode
```

Below is the NuSMV code regarding the incorrect model (**B**):

```
LTLSPEC G (InitialNode -> F Enter_game)
LTLSPEC G (Confirm_selection -> F Start_game)
LTLSPEC G (Start_game -> F ActivityFinalNode)
LTLSPEC G (Enter_game -> ( (X X Chose_team) xor (X X Sort_teams)))
LTLSPEC G (Chose_team & Sort_teams -> F Confirm_selection)
CTLSPEC EF ActivityFinalNode
```

As it can be seen from the code snippets shown above, this portion of the code differs between models in one specification: the penultimate, which is highlighted. As the

difference between these two models is the use of a Merge or Join node, the translation of the UML structure into the NuSMV code also differs, which is reflected by the difference in the boolean operands used between the two incoming flows to this node (the Merge or Join node). In model **A**, which uses the Merge node, the two incoming flows are related through an **"or"** operator (represented by a vertical bar), while in the case of model **B**, which uses the Join node, the flows relate through the use of an **"and"** operator (represented by the ampersand symbol).

When the NuSMV code is run, the NuSMV tool will assert the veracity of the different specifications, in relation to one another and to the graph defined by the code portions that precede the CTL* specifications.

The result of model-checking the code that corresponds to model **A** is that the model is true, and no specification is violated. When running the code corresponding to the activity diagram **B**, however, NuSMV reports that specification *"EF ActivityFinalNode"* was deemed false. Considering the behaviour of the CTL* connectors (which is explained in some detail in 5.5.3), it is relatively straightforward to derive the meaning of this specification. It states that the Final node must happen in the future in some execution, i.e. that at least one computational path must reach it, or even simpler, that the graph's Final node must be reachable.

This serves to show that NuSMV can understand that the flow of the activity diagram **B** can never achieve the Final node, which is a logical misconstruction. Using the result of the model checking process, the service can then report to the student that the Final node can never be reached in the case of the incorrectly constructed model (**B**). Similarly, the service can inform the student that model **A** passed the model checking process, as all specifications were true. Therefore, this module can be helpful to students, as it can provide the user with feedback regarding problems related to the inner logic of the model, even if the model can, for example, pass a validation procedure that is targeted at the more basic principles of model construction.

6.5 Further tests

Apart from the models used in the previously described case studies (sections 6.2, 6.3, and 6.4), more models were used to test other aspects of the service and its modules.

This was especially important in the validation phase, where multiple independent constraints had to be tested. For this purpose, extensive test models were created. These models included, if possible, all the verifiable elements, and were modified throughout various iterations targeted at testing each of the different validation constraints. Incorrectly modified models were used to test if the validation errors would show up when expected, and correct models were tested to see if they passed the validation process as intended.

Chapter 7

Conclusions and Future Work

This chapter provides a retrospective of the work done and offers some closing remarks. It also analyses the contributions that can be drawn from the work, and what can be made in the future to improve, complement, or repurpose it.

7.1 Conclusions

After evaluating the framework with respect to the Design Science approach, it is important to now look at it as a whole and draw the appropriate reflections.

Considering the current paradigm of the use of modelling tools in education (analysed in chapter 3), this framework is an improvement on the different standalone tools that can currently be used to simulate UML models. Because this service integrates various tools, each with its own distinct nature and context, it can offer a wider variety of simulation techniques, applied to various types of UML models. Therefore, this framework offers aggregation of multiple simulation technologies into a single service that can be easily accessed and used.

This work contributes not only with a complete working framework [33] but also with the integration and transformation processes that were developed (described in 5). The integration of the CPN Tools Toolkit, as well as the Umple and NuSMV tools into a Java context, together with the automatic transformation of models to create an input for these tools, is something that is of interest and that can be built upon. Not only can it aid in creating similarly education-targeted services, but it can also be of use to anyone who wishes, for industrial or educational purposes, to integrate these tools in the context of Java and Maven projects.

The systematic review that was conducted and is detailed in 3.2 can also be relevant to those who wish to delve into the subject of applying UML and modelling tools to

the educational world. It offers valuable insight into the existing assets, as well as the deficiencies in terms of specifically education-targeted modelling tools.

However, the main point of interest of this work is the framework that was produced, for its educational value. This is what initially motivated this work, and it was many times the deciding factor behind the difficult decisions that always have to be taken in this type of project. The goal of this work was to help students and to offer them, in the form of a centralised service, what they already possessed, albeit in a very fragmented and incomplete way. This framework integrates tools that do not originally support UML models as their input and, therefore, the simulation capabilities of these tools are now made available for students of UML in the form of a service that is whole, and easy to understand.

The case studies presented in 6 served to assert that, regarding the selected UML sublanguages, this framework can give the students valuable feedback on the inner workings of their models. This framework can help students in understanding the dynamic implications of their static models, in a way that cannot be achieved by simply validating their construction with respect to the rules of the UML language. Students can therefore obtain useful feedback on their UML models in an automated and simple way.

7.2 Future Work

Although this work presents a working product and a complete methodology, there is always work that can be built upon what was made.

One way to start is to look within the framework. There are certainly some aspects which can be improved, and which were stated in the course of this document, regarding technological limitations, time, etc. One of these cases was explained in section 5.4: the difficulties encountered in trying to use CPN Tools from a Java and Maven application. Perhaps, with more time and aid from the developers of Access/CPN (because of the severe lack of documentation), it can be possible to automate the totality of the processes described in 5.4.

It is also possible that the framework can be expanded. The 3 diagrams chosen out of the total of 14 are the product of combining two factors: the importance of the sublanguage in education (which removed a good portion of the candidates) and having the necessary context and semantics to make it possible to extract from the model something that can be simulated (which removed most of the widely used structural diagrams from the equation). However, if there are reliable ways to obtain that execution context from some structural models, or if some other behavioural models gain wider use in education, it would be possible to extend the framework to other UML sublanguages.

Regarding the Modelshake project, and taking into account the institutional context of the framework (mentioned in 1.2), it is also possible to develop new UML-related services such as the one presented in this work, as long as they are required or can have a positive impact in the context of UML-related courses.

Although this work mentions several times the UML Education Platform into which this service will be integrated, because it was developed in that context, it is important to underline that the framework produced is independent and can be (in great part because of the use of HTTP requests for access) applicable to a wide range of other educational platforms. Although integration should be easy when using the current form of the framework, it might also be easy to adapt it to, if necessary, have it function in an even wider array of platforms.

Bibliography

[1] ACM. *ACM Digital Library*. URL: https://dl.acm.org/ (visited on 11/24/2021).

[2] *Action Language for Foundational UML (Alf)*. OMG. 2017.

[3] S. Akayama, B. Demuth, T. Lethbridge, M. Scholz, P. Stevens, and D. Stikkolorum. "Tool use in software modelling education". In: *CEUR Workshop Proceedings* 1134 (Jan. 2013).

[4] Altova. *UModel UML Modeling Tool*. URL: https://www.altova.com/umodel (visited on 08/24/2021).

[5] ApacheAnt. *The Apache Ant Project*. URL: https://ant.apache.org (visited on 08/24/2021).

[6] M. Ardis, D. Budgen, G. W. Hislop, J. Offutt, M. Sebern, and W. Visser. *SE 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. Vol. 48. 11. 2015. DOI: 10.1109/MC.2015.345.

[7] ArgoUML. *Welcome to ArgoUML*. URL: https://argouml-tigris-org.github.io/ (visited on 02/17/2021).

[8] J. Banks, J. Carson, B. Nelson, and D. Nicol. *Discrete-Event System Simulation*. July 2009. ISBN: 9780136062127.

[9] F. U. Berlin. *Model checking*. URL: https://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2009-10_WS/L_19548_Model_checking/index.html (visited on 02/19/2021).

[10] A. Bolderston. "Writing an Effective Literature Review". In: *Journal of Medical Imaging and Radiation Sciences* 39 (June 2008), pp. 86–92. DOI: 10.1016/j.jmir.2008.04.009.

[11] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Vol. 1. Sept. 2012. DOI: 10.2200/S00441ED1V01Y201208SWE001.

[12] J. Cabot. *The most complete list of Executable UML tools*. URL: https://modeling-languages.com/list-of-executable-uml-tools/ (visited on 01/21/2021).

[13] G. Ciardo, R. III, R. Marmorstein, A. Miner, and R. Siminiceanu. "SMART: Stochastic Model-checking Analyzer for Reliability and Timing." In: Jan. 2002, p. 545. DOI: 10.1109/QEST.2004.1348056.

[14] F. Ciccozzi, I. Malavolta, and B. Selic. "Execution of UML models: a systematic review of research and practice". In: *Software Systems Modeling* 182 (June 2019). DOI: 10.1007/s10270-018-0675-4.

[15] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. Jan. 2001. ISBN: 978-0-262-03270-4.

[16] CPNTools. *Access/CPN*. URL: https://cpntools.org/access-cpn/ (visited on 11/17/2021).

[17] CPNTools. *CPN Tools*. URL: https://cpntools.org/ (visited on 10/25/2021).

[18] J. A. Custódio Soares. "Automatic Model Transformation from UML Sequence Diagrams to Coloured Petri Nets". MA thesis. Portugal: Faculdade de Engenharia da Universidade do Porto, 2017.

[19] DassaultSystèmes. *Cameo Simulation Toolkit*. URL: https://www.nomagic.com/product-addons/magicdraw-addons/cameo-simulation-toolkit (visited on 02/17/2021).

[20] Eclipse. *Code generation (EGL)*. URL: https://www.eclipse.org/epsilon/doc/egl/ (visited on 02/22/2021).

[21] Eclipse. *Eclipse Epsilon*. URL: https://www.eclipse.org/epsilon/ (visited on 02/22/2021).

[22] Eclipse. *Eclipse IDE*. URL: https://www.eclipse.org/eclipseide/ (visited on 02/22/2021).

[23] Eclipse. *Eclipse Modeling Framework (EMF)*. URL: https://www.eclipse.org/modeling/emf/ (visited on 11/18/2021).

[24] Eclipse. *Eclipse Papyrus*. URL: https://www.eclipse.org/papyrus/ (visited on 02/22/2021).

[25] Eclipse. *Model transformation (ETL)*. URL: https://www.eclipse.org/epsilon/doc/etl/ (visited on 02/22/2021).

[26] Eclipse. *Model validation (EVL)*. URL: https://www.eclipse.org/epsilon/doc/evl/ (visited on 02/22/2021).

[27] Eclipse. *Object Language (EOL)*. URL: https://www.eclipse.org/epsilon/doc/eol/ (visited on 02/22/2021).

[28] Eclipse. *Papyrus Real Time*. URL: https://www.eclipse.org/papyrus-rt/ (visited on 02/17/2021).

[29] EclipseMarketplace. *Papyrus Moka*. URL: https://marketplace.eclipse.org/content/papyrus-moka (visited on 02/17/2021).

[30] Elsevier. *ScienceDirect*. URL: https://www.sciencedirect.com/ (visited on 11/24/2021).

[31] G. Engels, J. Hausmann, and S. Sauer. "Teaching UML Is Teaching Software Engineering Is Teaching Abstraction". In: vol. 3844. Oct. 2005, pp. 306–319. ISBN: 978-3-540-31780-7. DOI: 10.1007/11663430_32.

[32] A. Evans, P. Sammut, and J. S. Willans, eds. *Metamodelling for MDA: First International Workshop, York, UK, November 2003, Proceedings*.

[33] M. Fernandes. *UML Simulation Framework*. URL: https://github.com/miguel-fernandes/uml-simulation-framework (visited on 11/24/2021).

[34] R. T. Fielding and R. N. Taylor. "Architectural Styles and the Design of Network-Based Software Architectures". AAI9980887. PhD thesis. 2000. ISBN: 0599871180.

[35] GitHub. *GitHub*. URL: https://github.com/ (visited on 11/18/2021).

[36] A. Gómez. *CPN Tools toolkit*. URL: https://github.com/abelgomez/cpntools.toolkit (visited on 10/25/2021).

[37] A. R. Hevner, S. T. March, J. Park, and S. Ram. "Design Science in Information Systems Research". In: *Management Information Systems Quarterly* 28 (Mar. 2004), pp. 75–.

[38] IBM. *IBM Engineering Systems Design Rhapsody - Developer*. URL: https://www.ibm.com/pt-en/marketplace/uml-tools (visited on 02/17/2021).

[39] IBM. *IBM Rational Software Architect Designer*. URL: https://www.ibm.com/developerworks/downloads/r/architect/index.html (visited on 02/17/2021).

[40] IEEE. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/Xplore/home.jsp (visited on 11/24/2021).

[41] IRISA. *Simulation of UML descriptions with UMLAUT Simulator*. URL: https://www.irisa.fr/UMLAUT/UserManual/simulator/overview.html (visited on 02/17/2021).

[42] *Systems and software engineering — High-level Petri nets — Part 1: Concepts, definitions and graphical notation*. Standard. Technical Committee : ISO/IEC JTC 1/SC 7. Aug. 2019.

[43] itemis. *YAKINDU Statechart Tools*. URL: https://www.itemis.com/en/yakindu/state-machine/ (visited on 02/17/2021).

[44] K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3642002838.

[45] JSON. *Introducing JSON*. URL: https://www.json.org/json-en.html (visited on 10/28/2021).

[46] B. Kitchenham and S. Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. Keele University and Durham University Joint Report, 2007.

[47] T. Kühne. "Matters of (Meta-) Modeling". In: *Software Systems Modeling* 5 (2006), pp. 369–385. DOI: https://doi.org/10.1007/s10270-006-0017-9.

[48] G. Liebel, O. Badreddin, and R. Heldal. "Model Driven Software Engineering in Education: A Multi-Case Study on Perception of Tools and UML". In: *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE T)*. 2017, pp. 124–133. DOI: 10.1109/CSEET.2017.29.

[49] Maven. *Apache Maven Project*. URL: https://maven.apache.org/what-is-maven.html (visited on 08/24/2021).

[50] Maven. *Maven – Introduction*. URL: https://maven.apache.org/index.html (visited on 08/24/2021).

[51] Maven. *Maven Repository: Central*. URL: https://mvnrepository.com/repos/central (visited on 11/18/2021).

[52] NOVA-LINCS. *NOVA LINCS*. URL: https://nova-lincs.di.fct.unl.pt/ (visited on 02/23/2021).

[53] NuSMV. *NuSMV: a new symbolic model checker*. URL: https://nusmv.fbk.eu/ (visited on 02/22/2021).

[54] *NuSMV 2.6 User Manual*. FBK-irst. 2010.

[55] OMG. *About the semantics of a foundational subset for executable UML models specification version 1.5 beta*. URL: https://www.omg.org/spec/FUML/About-FUML/ (visited on 02/22/2021).

[56] OMG. *About the Unified Modeling Language specification version 2.5.1*. URL: https://www.omg.org/spec/UML/About-UML/ (visited on 01/29/2021).

[57] OMG. *About the XML metadata interchange specification version 2.5.1*. URL: https://www.omg.org/spec/XMI/About-XMI/ (visited on 01/29/2021).

[58] OMG. *MDA - The architecture of choice for a changing world*. URL: https://www.omg.org/mda/ (visited on 01/21/2021).

[59] OMG. *Metaobject Facility*. URL: https://www.omg.org/mof/ (visited on 01/29/2021).

[60] OMG. *Object Management Group*. URL: https://www.omg.org/ (visited on 01/21/2021).

[61] OMG. *The OMG specifications catalog*. URL: https://www.omg.org/spec/ (visited on 02/22/2021).

[62] OpenLearn. *An introduction to software development: 6 Modelling and the UML - OpenLearn - Open University - M813$_1$*. URL: https://www.open.edu/openlearn/science-maths-technology/introduction-software-development/content-section-6 (visited on 02/19/2021).

[63] Postman. *Postman API Platform*. URL: https://www.postman.com/ (visited on 11/18/2021).

[64] QuantumLeaps. *About QM*. URL: https://www.state-machine.com/qm/ (visited on 02/17/2021).

[65] RAPTOR. *Welcome to the RAPTOR home page*. URL: https://raptor.martincarlisle.com/ (visited on 02/17/2021).

[66] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[67] D. A. Ryan. *Class diagram of the 13 types of diagrams of the Unified Modelling Language 2.0*. URL: https://upload.wikimedia.org/wikipedia/commons/7/74/Uml_diagram.svg (visited on 11/14/2021). License: Creative Commons CC BY-SA 2.5; Released into Public Domain.

[68] D. C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58.

[69] E. Seidewitz. "What models mean". In: *Software, IEEE* 20 (Oct. 2003), pp. 26–32. DOI: 10.1109/MS.2003.1231147.

[70] Simul8. *What is simulation?* URL: https://www.simul8.com/what-is-simulation (visited on 02/19/2021).

[71] SinelaboreRT. *Generate production ready source code from UML state – and activity diagrams!* URL: https://www.sinelabore.de/doku.php (visited on 02/17/2021).

[72] Y. Singh and M. Sood. "Models and Transformations in MDA". In: *2009 First International Conference on Computational Intelligence, Communication Systems and Networks*. 2009, pp. 253–258. DOI: 10.1109/CICSYN.2009.52.

[73] Sodius. *Willert Embedded UML Studio*. URL: https://www.willert.de/software-tools/modellgetriebene-softwareentwicklung/willert-embedded-uml-studio/ (visited on 02/17/2021).

[74] F. Song, S. Parekh, L. Hooper, Y. K. Loke, and J. Ryder. "Dissemination and publication of research findings : an updated review of related biases". In: *Health Technol Assess* 14 (2010), p. 8. DOI: https://doi.org/10.3310/hta14080.

[75] SourceForge. *USE: UML-based Specification Environment*. URL: https://sourceforge.net/projects/useocl/ (visited on 02/17/2021).

[76] SparxSystems. *Enterprise Architect*. URL: https://sparxsystems.com/ (visited on 02/17/2021).

[77] Spring. *Spring Framework*. URL: https://spring.io/projects (visited on 08/24/2021).

[78] Springer. *Springer Link*. URL: https://link.springer.com/ (visited on 11/24/2021).

[79] SpringerLink. *Metamodel*. URL: https://link.springer.com/referenceworkentry/10.1007%5C%2F978-0-387-39940-9_898 (visited on 10/26/2021).

[80] F. Ul Muram, H. Tran, and U. Zdun. "Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking". In: *Electronic Proceedings in Theoretical Computer Science* 147 (Apr. 2014). DOI: 10.4204/EPTCS.147.7.

[81] uml-diagrams.org. *Activity Diagrams*. URL: https://www.uml-diagrams.org/activity-diagrams.html.

[82] uml-diagrams.org. *Combined Fragment*. URL: https://www.uml-diagrams.org/sequence-diagrams-combined-fragment.html (visited on 11/17/2021).

[83] uml-diagrams.org. *UML 2.5 Diagrams Overview*. URL: https://www.uml-diagrams.org/uml-25-diagrams.html (visited on 01/21/2021).

[84] Umple. *UmpleOnline*. Feb. 22, 2021. URL: https://cruise.umple.org/umpleonline/.

[85] Umple. *Basic State Machines*. URL: https://cruise.umple.org/umple/BasicStateMachines.html (visited on 10/28/2021).

[86] Umple. *Model-Oriented Programming - Umple.org*. URL: https://cruise.umple.org/umple/ (visited on 02/17/2021).

[87] Umple. *Umple Tools*. URL: https://cruise.umple.org/umple/UmpleTools.html (visited on 10/28/2021).

[88] Uppaal. *Uppaal*. URL: https://uppaal.org/ (visited on 02/22/2021).

[89] M. Vardi. "Branching vs. Linear Time: Final Showdown". In: vol. 2031. Mar. 2001, pp. 1–22. ISBN: 978-3-540-41865-8. DOI: 10.1007/3-540-45319-9_1.

[90] S. Verma, P. Lee, and I. G. Harris. "Error Detection Using Model Checking vs. Simulation". In: *2006 IEEE International High Level Design Validation and Test Workshop*. 2006, pp. 55–58. DOI: 10.1109/HLDVT.2006.319964.

[91] VisualParadigm. *What is Activity Diagram?* URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/.

[92] I. Wagner, F. Dressler, V. Schmitt, and R. German. "SYNTONY: network protocol simulation based on standard-conform UML 2 models". In: Jan. 2007, p. 21. DOI: 10.1145/1345263.1345290.

[93] B. Westphal. "Teaching Software Modelling in an Undergraduate Introduction to Software Engineering". In: *Proceedings of the 22nd International Conference on Model Driven Engineering Languages and Systems*. IEEE Press, 2019, pp. 690–699. ISBN: 9781728151250.

[94] R. J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Germany: Springer, 2014. ISBN: 978-3-662-43839-8. DOI: 10.1007/978-3-662-43839-8.

[95] Wikipedia. *List of Unified Modeling Language tools*. URL: https://en.wikipedia.org/wiki/List%5C_of%5C_Unified%5C_Modeling%5C_Language%5C_tools (visited on 01/21/2021).

[96] xtUML. *BridgePoint*. URL: https://xtuml.org/ (visited on 02/17/2021).

ANNEX I

Annexe 1: The selected Primary Studies of the Systematic Literature Review

Study	Title	Authors	Year
S1	Executable and Translatable UML – How Difficult Can it Be?	Burden, H. et al.	2011
S2	Raptor: a visual programming environment for teaching object-oriented programming	Carlisle, M. et al.	2009
S3	Teaching UML using umple: Applying model-oriented programming in the classroom	Lethbridge, T. et al.	2011
S4	Model Driven Software Engineering in Education: A Multi-Case Study on Perception of Tools and UML	Liebel, G. et al.	2007
S5	Assessing the influence of feedback-inclusive rapid prototyping on understanding the semantics of parallel UML statecharts by novice modellers	Sedrakyan, G. et al.	2017
S6	UMLAnT: an Eclipse plugin for animating and testing UML designs	Trong, T. et al.	2005
S7	Software engineering in a nutshell for Electrical Engineering students	von Schwerin, M. et al.	2014

Table I.1: The selected Primary Studies of the Systematic Literature Review.

ANNEX II

Annexe 2: The search strings used in the SLR for each digital library

Digital Library	Search String
ACM (https://dl.acm.org/search)	Abstract:(UML) AND Abstract:(simulat* OR execut* OR ani-mat* OR generat* OR debug*) AND Abstract:(educat* OR teach*OR “students” OR academi* OR learn* OR understand*) AND Fulltext:(“model simulation” OR “model simulator” OR “models simulators” OR “UML simulation” OR “UML simulator” OR“UML simulators” OR “model execution” OR “UML execution”OR “simulate models” OR “execute models” OR “simulation of models” OR “simulation of UML” OR “execution of models” OR “execution of UML” OR "model animation"OR "code generation"OR "UML animation"OR "animation of models"OR"animation of UML"OR "generate code"OR "animate models"OR"animate UML"OR “debugging models” OR “model debugging”OR “debugging of models” OR “debug models” OR “debugging UML” OR “UML debugging” OR “debugging UML” OR “debug UML”)
Elsevier (https://www.sciencedirect.com/search)	Find Articles with these terms: “model simulation” OR “UML simulation” OR “model execution” OR “UML execution” OR“simulation of models” OR “execution of models” OR “model animation” OR “code generation” OR “UML animation”; Title,abstract or author-specified keywords: UML AND (“simulation”OR “execution” OR “animation”) AND (“education” OR “educational” OR “students” OR “learning” OR “teaching”); Article types: Research Articles
IEEE (https://ieeexplore.ieee.org/Xplore)	(("Abstract":"UML") AND (("Abstract":simulat*) OR ("Abstract":execut*) OR("Abstract":animat*) OR ("Abstract":generat*)) AND (("Abstract":educat*) OR ("Abstract":teach*) OR ("Abstract":"students") OR ("Abstract":"academic") OR ("Abstract":"academically") OR ("Abstract":"academia") OR ("Abstract":learn*) OR ("Abstract":"understand") OR ("Abstract":"understanding")) AND (("Full Text Only":"model simulation") OR ("Full TextOnly":"model simulator") OR ("Full Text Only":"model simulators") OR ("Full Text Only":"UML simulation") OR ("FullText Only":"UML simulator") OR ("Full Text Only":"UML simulators") OR ("Full Text Only":"model execution") OR ("FullText Only":"UML execution") OR ("Full Text Only":"simulate models") OR ("Full Text Only":"execute models") OR ("Full TextOnly":"simulation of models") OR ("Full Text Only":"simulation of UML") OR ("Full Text Only":"execution of models") OR ("FullText Only":"execution of UML") OR ("Full Text Only":"model animation") OR ("Full Text Only":"code generation") OR ("FullText Only":"UML animation") OR ("Full Text Only":"animation of models") OR ("Full Text Only":"animation of UML") OR ("Full Text Only":"generate code") OR ("Full Text Only":"animate models") OR ("Full Text Only":"animate UML"))

Table II.1: The search strings used in the SLR for each digital library.

ANNEX III

ANNEXE 3: TRANSFORMATION OF UML STRUCTURES INTO SMV DESCRIPTIONS

UML constructs	Transformation Rules

Initial Node *a*	`init(a) := TRUE;`
	`next(a) := case`
	`  a : FALSE;`
	`  TRUE : a;`
	`esac;`

Final Node, Action, Fork Node, Join Node *a*	`init(a) := FALSE;`
	`next(a) := case`
	`  incoming_1 & incoming_2 & ... & incoming_n : TRUE;`
	`  a : FALSE;`
	`esac;`

Decision Node *a*	`init(a) := undetermined;`
	`next(a) := case`
	`  incoming_1 & incoming_2 & ... & incoming_n$ : {outgoing_1, outgoing_2, ...,`
	`  outgoing_n};`
	`  a != undetermined : undetermined;`
	`esac;`

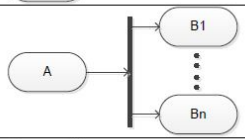
Merge Node	`init(a) := FALSE;`
	`next(a) := case`
	`  incoming_1 | incoming_2 | ... | incoming_n : TRUE;`
	`  a : FALSE;`
	`esac;`

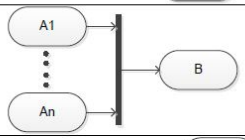
Figure III.1: Transformation of UML structures into SMV descriptions (this table is presented in the Ul Muram et al. paper *Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking* [80]).

ANNEX IV

ANNEXE 4: REPRESENTATION OF UML ACTIVITY DIAGRAM ELEMENTS USING LTL PRIMITIVES

Description	Modeling Notation	LTL Primitive
Sequence: A set of actions executed in sequential order. E.g., the action A1 will be performed before A2.		`G (A1 -> F A2)`

Fork Node: The execution of a fork node leads to the parallel execution of subsequent actions (B1...Bn).		`G (A -> (F B1`
		`& F B2`
		`& ...`
		`& F Bn))`

Join Node: The execution of two or more parallel actions leads to the execution of Join Node.		`(G (A1`
		`& A2`
		`& ...`
		`& An) -> F B)`

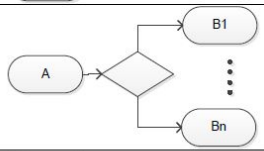
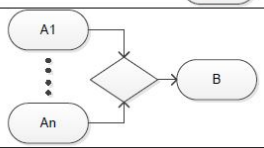
Decision Node: The execution of a Decision Node eventually followed by the execution of one and only one action among the available set of actions based on the decision guard.		`G (A -> (F B1 xor F B2` `xor ... xor F Bn))`
Merge Node: At least one action among a set of alternative execution of actions will lead to the execution of Merge Node.		`G (A1 | A2 | ... | An ->` `F B)`

Figure IV.1: Representation of UML activity diagram's elements using LTL primitives (this table is presented in the Ul Muram et al. paper *Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking* [80]).

Annexe 5: Sequence-to-CPN transformation rule set

Rule	Name	Transformed element	Preceed. rules
R1	Initial transformation	-	-
R2	Lifelines to initial places	Lifeline	R1
R3	Events to after places	MessageOcurrenceSpec.	R1
R4	Weak sequencing combined fragments	CombinedFragment	R2, R3
R5	Strict sequencing combined fragments	CombinedFragment	R2, R3
R6	Parallel combined fragments	CombinedFragment	R2, R3
R7	Optional combined fragments	CombinedFragment	R2, R3
R8	Alternative combined fragments	CombinedFragment	R2, R3
R9	Loop combined fragments	CombinedFragment	R2, R3
R10	Transformation of messages	Message	R4-R9
R11	Final transformation	-	R10

Table V.1: Transformation rule set (this table is presented in the Custódio Soares thesis *Automatic Model Transformation from UML Sequence Diagrams to Coloured Petri Nets* [18]).